Testing a Formally Verified Compiler

David Monniaux, Léo Gourdin, Sylvain Boulmé, Olivier Lebeltel

Testing a Formally Verified Compiler*

David Monniaux[0000−0001−7671−6126], Léo Gourdin,
Sylvain Boulmé[0000−0002−9501−9606], and Olivier Lebeltel.

Univ. Grenoble Alpes, CNRS, Grenoble INP, Verimag.

Abstract. We report on how we combine tests and formal proofs while developing extensions to the CompCert formally verified compiler.

Keywords: Software Engineering, Formal Proofs, Experimental Validation.

1 Introduction

CompCert is a formally verified compiler, which compiles a large fragment of the C programming language to assembly code. "Formally verified" means that there is a machine-checked proof (here with the Coq proof assistant) that if *compilation succeeds*, the possible executions of this assembly code match those of the C source [11]. Hence, CompCert's formal proof aims at forbidding *miscompilation*, i.e., compiled programs with unexpected behaviors.

Miscompilations being potentially very unsafe for final users, eradicating them is highly desirable. However, according to [27], in mainline compilers—without formal proof—such as GCC and LLVM, more than half of the bugs in optimization passes are miscompilations. This indicates that eradicating them is nonetheless very difficult with standard software engineering methods.

While developing new features for CompCert, we realized that formal proofs were not enough, and that testing was required. We thus greatly extended the testing system present in the GitHub repository of CompCert. This paper describes the challenges that we experienced and the solutions we found.[1]

Here are some incorrect behaviors not prevented by CompCert's formal proof and that we experienced. (1) **Unexpected compilation failure** (compiler internal error)**:** the formal proof assumes that compilation succeeds; always failing would trivially satisfy this criterion. Failing when incorrect behavior occurs protects against miscompilation, as detailed in §2. (2) **Compilation timeout:** compilation may loop forever or take prohibitively long. (3) **Error during assembling or linking:** reasons for this range from details in the syntax comment of certain assemblers to the use of short branch instructions.[2]

The above behaviors are not miscompilations, but compile-time errors. In contrast, the following may lead to miscompilation, revealing issues in the Trusted

* Work partially supported by the LabEx PERSYVAL-Lab (ANR-11-LABX-0025-01).

[1] Get our code on https://gricad-gitlab.univ-grenoble-alpes.fr/certicompil/Chamois-CompCert.

[2] A short branch and its target must be close, which may be false on large functions.

Computing Base (TCB) which the formal proof relies on. (4) **Source semantics mismatch:** the C language is surprisingly complicated, and its semantics as formally defined in CompCert may diverge from the informal one defined in the standard, or in CompCert's manual. (5) **Assembly semantics mismatch:** the semantics of assembly language, plus platform-dependent peculiarities (e.g., how to access global symbols), may contain unexpected pitfalls (such as out-of-range operands resulting in a wrap-around behavior). Furthermore, some instructions present in CompCert's "assembly" languages are actually macros expanded by trusted (unverified) OCaml code. Some of these macros were inexactly specified, for instance by forgetting a clobbered register—this went unnoticed as long as the compiler did not take advantage of the value in that register being preserved [13]. (6) **Assembly language mis-expansion or misprinting:** we also found rare miscompilations in the expansion or printing of macros-instructions. For instance, a rarely selected instruction was misprinted; our macro-instruction for stack frame allocation did something incorrect but that almost always worked.

Some of these issues had not been detected for months, which indicated that testing was insufficient. We thus gradually developed a testing and continuous integration framework, which we ported to the other targets supported by CompCert, leading to new discoveries [13].

Here is an outline of the paper. Section 2 explains how we deliberately rely on compiler failures—issue (1) above—in order to simplify our formal proofs. Indeed, compiler failures seem a lesser evil than other issues, and testing is anyway necessary for these other issues. The remainder of the paper examines methods for identifying all these issues. Section 3 introduces an extended suite of tests. When a new bug is triggered, we describe in Section 4 how to produce a minimal example. CompCert's TCB is intended to be as small as possible, but it is almost inevitable to rely on trusted parts for both ends of the pipeline: the C and assembly semantics, as well as the expansion of assembly macro-instructions. Details about these sensitive parts and possible solutions are provided in Section 5. Another important aspect of testing is the benchmarking: measuring the performance of the generated code faithfully is not trivial, and require a rigorous strategy. Section 6 covers this topic, and presents our evaluation toolkit to compile, execute, and analyze representative benchmarks with multiple compilers, including CompCert. Finally, Section 7 concludes. Some complements are given in Appendix A.

2 Formally Verified Defensive Tests

CompCert's formal proof assumes compilation success, and states nothing in case of failure. This allows for *Formally Verified Defensive Programming* (FVDP) [1]: complex computations are delegated to efficient functions, called *oracles*, whose implementations are both *untrusted* and hidden from formal proofs; only a *defensive test* of their result is formally verified.

For example, we introduced a hash-consing factory, formally verified using only defensive pointer equality tests [1, §3.3.2][23, §4.4]. We use *translation validation* [16,19,25], in particular by *symbolic execution*, an approach for compiler

testing [6,21]: the same formally verified translation validator is used to check many untrusted transforms [24]. We validate fixpoints, computed using imperative data structures that cannot be easily modeled in Coq, by a verified checker [14].

Such a design induces simpler formal proofs, high modularity, while allowing for arbitrarily complex oracles.

In short, FDVP is a systematic approach for turning miscompilations into compiler failures (behavior (1) of §1). It helps us formally prove the absence of miscompilation (w.r.t. the TCB). Furthermore, such a compiler failure helps in debugging; the uncaught exception raised by the compiler directly points to the issue: either an incorrect oracle or a too coarse defensive test. For this, we still need testing for compiler failures extensively.

3 Test Suite for Compiler Correctness

AbsInt markets a version of CompCert suitable for qualification for safety-critical applications, e.g. nuclear power plants and avionics [9]. To our knowledge, this involves a large test suite, including the standard compliance suite SuperTest.[3] This test suite not being publically available, we extended the regression tests of CompCert's GitHub repository[4] with tests produced by off-the-shelf random generators, a form of *compiler fuzzing* [12], as well as the gcc "torture test" suite. The purpose of adding new tests here is to avoid both compile-time errors (especially failures and timeouts in our translation validators) and any form of miscompilation. Testing can help identify the six bad behaviors mentioned in §1. For each program generator i (items (a) and (b) below), N_i programs are generated by varying the random seed of the generator from 0 to $N_i - 1$, ensuring reproducibility. (a) **Csmith 2.3.0 & YarpGen 1.1:**[5] The produced code—which is supposed to be compilable and devoid of undefined behaviors—is compiled with both CompCert and gcc and run on the target processor or an instruction set simulator (e.g. qemu). The results are then compared (*differential testing*). Yet this code may fail to terminate, thus a timeout is used; the test is considered valid if both programs yield the same value, or fail to terminate within the timeout. The timeout value is large enough to avoid cases where only one program, better optimized, terminates while the other does not, but would with more time. (b) **CCG:**[6] Its programs are not expected to run correctly, so we thus just test that they compile correctly. (c) **gcc 12.2.0 tests:** Finally, we added gcc's C torture tests, both for compilation only and for compilation + execution, except those that relied on gcc-specific extensions (such as SIMD

[3] See https://www.absint.com/ and https://solidsands.com/products/supertest.

[4] See directory "`test`" of https://github.com/AbsInt/CompCert/.

[5] https://github.com/csmith-project/csmith [26] with packed structures (gcc extension) disabled. https://github.com/intel/yarpgen (One random seed value is excluded because on ARM it leads to register allocation causing out of memory. Large auto-generated programs causing resource exhaustion in the compiler is not considered a bug [10].)

[6] https://github.com/Mrktn/ccg. We disabled the generation of ternary conditional operators with omitted middle operand, a gcc extension not supported by CompCert.

vectors), gcc-specific behaviors on undefined or unspecified cases, and those that tested the limits of the compiler (i.e. very large number of declarations).

Each newly added generator or suite triggered new bugs (in our own extensions, or in upstream recent extensions not yet covered by AbsInt's tests). The full test suite, including the three items above, is triggered from continuous integration for a variety of targets.[7] Generally, test cases that triggered bugs had to be reduced (§4) before the bug could be investigated. In addition to making us more confident about the reliability of our validators by detecting compilation failures (§2), the tests helped to discover miscompilations (§5) and intolerable compilation running-times. We have also another framework for evaluating the performances of the generated code, briefly described in §6 (the test suite of this section is not designed to measure performance).

4 Reducing Test Cases

Random and application test cases are often too large for the compiler developer to identify bugs. Finding a reduced test case that exhibits the same bug is the first step for understanding what went wrong [4]. Reducing cases by hand is tedious and error-prone; we thus automated this task using C-Reduce.[8]

C-Reduce takes as input a C program (possibly several source or header files) and a predicate (typically a shell script), which must be satisfied by this original program, and gradually reduces the program by removing or simplifying declarations and statements, as long as the predicate is satisfied, until a minimal test case is produced. A timeout may be specified for evaluating the predicate, as well as other parameters (e.g. the number of cores for parallel execution).

On a compiler failure, the predicate is easy to write: simply execute the compiler and check that it crashes with the same error. For a timeout error, run the compiler with a timeout and check that it really reaches the timeout. For assembling (resp. linking) errors, check that the compiler runs successfully, but that assembling (resp. linking) fails with the same error.

When the error is miscompilation, the predicate is much harder to write. The main test is then to compile the program with CompCert and another compiler (gcc or clang), run both versions (on either the target processor or an instruction set simulator), and compare execution results: miscompilation is detected if their results diverge (either they print different values, or the version produced by CompCert crashes while the other does not). However, this criterion works only if the program does not have undefined behavior. Indeed, if a program has undefined behavior, such as reading from an uninitialized variable, it is normal that it gives one result with one compiler, another with another compiler. Unfortunately, if given the opportunity by a lax predicate, C-Reduce will write programs that have undefined behavior: this is what usually happens if one removes statements

[7] x86, x86-64, AArch64, ARMv7 with software and hardware floating-point, 32-bit PowerPC, 64-bit RISC-V, KVX. This even led us to find bugs in qemu for PowerPC.

[8] https://embed.cs.utah.edu/creduce/ [20].

haphazardly from a program, such as statements that initialize variables. The predicate should thus refuse programs that have undefined behavior.

An obvious method is to compile the program using gcc and/or clang and run the resulting program under Valgrind,[9] which checks for undefined behaviors at the object code level, such as accessing data outside currently allocated blocks, or branching (or conditionally moving data) according to a test on uninitialized values. In addition, if both gcc and clang are used, the program should give exactly the same output. Another approach is to compile the program using gcc and/or clang using the "sanitizers", which insert code that detects at runtime that certain undefined behaviors have happened. The categories of bugs that Valgrind and the "sanitizers" can detect overlap, but are not identical.

However, running the program under an instruction set simulator or under Valgrind is costly. It is thus very beneficial to exclude programs that evidently are likely to exhibit undefined behavior prior to running them. We ask gcc and clang to print error (as opposed to warning) messages when they discover certain undefined behaviors such as the use of an uninitialized variable.

Finally, some sanity checks on programs results are performed: for reducing a miscompilation bug that results in the gcc and CompCert versions printing out different checksums (e.g. for programs generated by Csmith), we check that the output indeed includes a line printing out a checksum. See Figure 1 in Appendix.

5 Checking the Trusted Computing Base

Formal C semantics [issue (4) of §1] Two mismatches between the C formal semantics of CompCert and that of the standard were identified: (a) memory allocation is supposed to always return a valid pointer (e.g. not `NULL`), unlike the standard's `malloc` function; (b) storing a bitfield in an otherwise uninitialized structure, then reading from it, yields an undefined (uninitialized) value whereas it should yield the value stored (truncated to the appropriate size); this is because initialized-ness is defined at word level, not bit level.

The second one was identified when using CompCert's reference C interpreter on reduced test cases. Neither can currently cause miscompilation, because CompCert's optimizations do not exploit these shortcuts of the formal semantics.

Unit testing of the assembly (macro-)instructions [issues (5) and (6) of §1] The formally verified part of CompCert ends with an abstraction of assembly code: (a) machine instructions are seen as operating over values, with the value datatype defined as a disjoint union of 32- and 64-bit integers, 32- and 64-bit floating-point values, and pointers, whereas in reality they are all bitstrings; pointers are pairs of a block identifier and an offset within this block, whereas in reality they are (in a flat memory model) just integers; (b) memory copy, used either as an intrinsic or for implementing C structure assignment, is a macro whose expansion depends on the block size; (c) memory is seen as independent blocks, and it is

[9] https://valgrind.org/ [17,18,22]

impossible to move from one to another by pointer arithmetic; each stack frame is an independent block; macros implement the creation and destruction of stack frames by expanding into a sequence that saves the previous frame's address and adjusts the stack pointer; (d) certain basic operations may be macro-instructions: taking the address of a global symbol may entail splitting it between high- and low-order bits or more complicated schemes;[10] on a 32-bit platform, adding two 64-bit numbers split in 32-bit registers involves propagating carry bits, which may not be reflected in the assembly semantics; etc.

The expansion of macro-instructions—miscompilation (6) of §1—has been identified as the main source of miscompilation bugs in CompCert (which, however, remain rare) [13], especially if they involve case analyses over arguments (register aliasing, constants): rarely exercised cases could be incorrect. Also, in some cases, wrong but syntactically correct instructions were printed.

Possibly, the definition of the formal semantics of the regular assembly instructions—mismatch (5) of §1—could also feature incorrect corner cases. However, our testing approach currently lacks a complete unit testing system that would match the semantics of abstract (pseudo-)instructions in their formal specification to that on the target processor. Such testing would not only need to check that the result values of the instruction fit the specification, whether these values are in processor registers or in memory, but also that values that the specification lists as unmodified were unaffected. Based on our coverage measurements, our tests do not validate the entire OCaml code of the TCB.

6 Testing the Performance of Generated Code

Developing a new pass require ensuring that its results are not only *beneficial* on every target, but also that it does not *interfere* with existing optimizations [2]. Since most instruction set simulators are not capable of counting cycles, we measure and compare the code performance of various CompCert configurations directly on the target core. Firstly, such measures are often subject to many *subtle biases* [15], among which are the runtime environment, the size of the benchmarks, as well as decisions by the operating system kernel: frequency scaling, migration between cores, etc. We address this by running multiple execution of each test, and by forcing the process to remain on the same core (e.g. with `taskset`), under the same shell environment. Then, we average out the different executions to filter them when the *relative standard deviation* exceeds a certain threshold (*noise elimination*), so that too small or unreliable tests are removed.

Secondly, for the comparison to GCC or Clang to be *fair*, we have to compare compilers on a common basis of applicable transformations. Notably, we disable options that would not be correct in the CompCert semantics—e.g. "fast-math",

[10] E.g., we implemented using symbols in thread-local storage (TLS) because some platforms put some C library globals in TLS even if the program is not multithreaded.

or replacing $ab + c$ by a fused multiply-add,[11] and instruction set extensions that CompCert cannot use—e.g., vector (SIMD) instructions.

For the measure to be *representative* and to avoid concluding on an overfitted subset of benchmarks, we combine several test suites: (a) an extended version of LLVMtest[12] (various concrete applications); (b) the computational oriented PolyBench;[13] the embedded oriented suites (c) MiBench [5] and (d) TACLeBench [3]; (e) and our own test suite with multiple types of tests (from small sorting algorithms to OCaml or GCC whole applications). See details in Appendix A.2.

We developed a performance measuring toolkit,[14] based on a JSON configuration that details, for each compiler to measure, sets of options to compare. Shell scripts then automatically (a) build; (b) copy to the target machine (e.g. via `rsync`); (c) run N times on a fixed core; and (d) gather tests results as CSV files. Finally, a Python/Pandas script filters and analyses CSVs to yield (in text or as a plot) the observed gains w.r.t. a reference compiler with options set pair.

7 Conclusion on Testing Formally Verified Software

Extensive testing on formally verified software is necessary for at least two reasons. First, the formal specification may not guarantee all the properties expected by users, but only *critical* ones (e.g. no miscompilation for CompCert). Second, critical bugs may still remain, because the formal specification might not exactly fit reality. However, in this case, critical bugs are in the—much smaller and simpler—TCB. For example, in our experience, bugs in CompCert's TCB are fixed a few days after their discovery. This seems to contrast with critical bugs in usual optimizing compilers which remain, on average, more than one year before being fixed [27].

Moreover, *formally verified defensive testing*—as introduced in §2—may drastically reduce the overall proof effort. It is also much more powerful for finding bugs of untrusted optimizations than usual testing: it detects all their miscompilation bugs for a given input file, without even running the code they generate. In contrast, in a usual compiler devoid of independent verification of the optimization passes, some of the bugs that we detected because compilation aborted would have resulted in miscompilation, which may have been unnoticed by usual testing.

Acknowledgement. We thank Xavier Leroy for his useful feedbacks on a preliminary version of this paper.

[11] An "fma" rounds differently from a $\times$ followed by a $+$. Replacing the latter by the former thus is a semantic change, which runs afoul of CompCert's soundness criteria.

[12] https://github.com/lac-dcc/Benchmarks

[13] http://web.cs.ucla.edu/~pouchet/software/polybench/

[14] Our benchmarks and toolkit: https://gricad-gitlab.univ-grenoble-alpes.fr/certicompil/chamois-benchs.

References

1. Boulmé, S.: Formally Verified Defensive Programming (efficient Coq-verified computations from untrusted ML oracles). Habilitation à diriger des recherches, Université Grenoble-Alpes (Sep 2021), https://hal.archives-ouvertes.fr/tel-03356701, see also http://www-verimag.imag.fr/~boulme/hdr.html
2. Curtsinger, C., Berger, E.D.: STABILIZER: Statistically Sound Performance Evaluation. In: ASPLOS'2013. pp. 219–228. ACM (2013). https://doi.org/10.1145/2451116.2451141
3. Falk, H., Altmeyer, S., Hellinckx, P., Lisper, B., Puffitsch, W., Rochange, C., Schoeberl, M., Sørensen, R.B., Wägemann, P., Wegener, S.: TACLeBench: A benchmark collection to support worst-case execution time research. In: Schoeberl, M. (ed.) 16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016). OpenAccess Series in Informatics (OASIcs), vol. 55, pp. 2:1–2:10. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2016)
4. GCC bugs, https://gcc.gnu.org/bugs/
5. Guthaus, M., Ringenberg, J., Ernst, D., Austin, T., Mudge, T., Brown, R.: MiBench: A free, commercially representative embedded benchmark suite. In: Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538). pp. 3–14. IEEE, Austin, TX, USA (2001). https://doi.org/10.1109/WWC.2001.990739, http://ieeexplore.ieee.org/document/990739/
6. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976). https://doi.org/10.1145/360248.360252, https://doi.org/10.1145/360248.360252
7. Knoop, J., Rüthing, O., Steffen, B.: Lazy code motion. In: Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation - PLDI '92. pp. 224–234. ACM Press, San Francisco, California, United States (1992). https://doi.org/10.1145/143095.143136, http://portal.acm.org/citation.cfm?doid=143095.143136
8. Knoop, J., Rüthing, O., Steffen, B.: Lazy Strength Reduction. Journal of Programming Languages **1**, 71–91 (1993)
9. Kästner, D., Wünsche, U., Barrho, J., Schlickling, M., Schommer, B., Schmidt, M., Ferdinand, C., Leroy, X., Blazy, S.: CompCert: Practical experience on integrating and qualifying a formally verified optimizing compiler. In: ERTS 2018: Embedded Real Time Software and Systems. SEE (Jan 2018), http://xavierleroy.org/publi/erts2018_compcert.pdf
10. Leroy, X.: Answer to CompCert bug #137, https://github.com/AbsInt/CompCert/issues/137#issuecomment-243353529
11. Leroy, X.: Formal verification of a realistic compiler. Communications of the ACM **52**(7) (2009). https://doi.org/10.1145/1538788.1538814
12. Marcozzi, M., Tang, Q., Donaldson, A.F., Cadar, C.: Compiler fuzzing: how much does it matter? Proc. ACM Program. Lang. **3**(OOPSLA), 155:1–155:29 (2019). https://doi.org/10.1145/3360581
13. Monniaux, D., Boulmé, S.: The trusted computing base of the compcert verified compiler. In: European Symposium on Programming Languages and Systems (ESOP '22). Lecture Notes in Computer Science, vol. 13240, pp. 204–233. Springer (2022). https://doi.org/10.1007/978-3-030-99336-8_8
14. Monniaux, D., Six, C.: Formally verified loop-invariant code motion and assorted optimizations. ACM Trans. Embed. Comput. Syst. **22**(1), 3:1–3:27 (2023). https://doi.org/10.1145/3529507, https://doi.org/10.1145/3529507

15. Mytkowicz, T., Diwan, A., Hauswirth, M., Sweeney, P.F.: Producing Wrong Data Without Doing Anything Obviously Wrong! In: ASPLOS'2009. pp. 265–276. ACM (2009). https://doi.org/10.1145/1508244.1508275
16. Necula, G.C.: Translation validation for an optimizing compiler. In: Programming Language Design and Implementation (PLDI). pp. 83–94. Association for Computing Machinery (2000). https://doi.org/10.1145/349299.349314, http://doi.acm.org/10.1145/349299.349314
17. Nethercote, N., Seward, J.: How to shadow every byte of memory used by a program. In: Krintz, C., Hand, S., Tarditi, D. (eds.) Proceedings of the 3rd International Conference on Virtual Execution Environments, VEE 2007, San Diego, California, USA, June 13-15, 2007. pp. 65–74. ACM (2007). https://doi.org/10.1145/1254810.1254820, https://doi.org/10.1145/1254810.1254820
18. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. In: Ferrante, J., McKinley, K.S. (eds.) Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007. pp. 89–100. ACM (2007). https://doi.org/10.1145/1250734.1250746, https://doi.org/10.1145/1250734.1250746
19. Pnueli, A., Siegel, M., Singerman, E.: Translation validation. In: Tools and Algorihtms for the Construction and Analysis of Systems (TACAS). Lecture Notes in Computer Science, vol. 1384, pp. 151–166. Springer (1998). https://doi.org/10.1007/BFb0054170, https://doi.org/10.1007/BFb0054170
20. Regehr, J., Chen, Y., Cuoq, P., Eide, E., Ellison, C., Yang, X.: Test-case reduction for C compiler bugs. In: Vitek, J., Lin, H., Tip, F. (eds.) ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012. pp. 335–346. ACM (2012). https://doi.org/10.1145/2254064.2254104
21. Samet, H.: Compiler testing via symbolic interpretation. In: Gosden, J.A., Johnson, O.G. (eds.) Proceedings of the 1976 Annual Conference, Houston, Texas, USA, October 20-22, 1976. pp. 492–497. ACM (1976). https://doi.org/10.1145/800191.805648
22. Seward, J., Nethercote, N.: Using Valgrind to detect undefined value errors with bit-precision. In: Proceedings of the 2005 USENIX Annual Technical Conference, April 10-15, 2005, Anaheim, CA, USA. pp. 17–30. USENIX (2005), http://www.usenix.org/events/usenix05/tech/general/seward.html
23. Six, C., Boulmé, S., Monniaux, D.: Certified and efficient instruction scheduling: application to interlocked VLIW processors. Proc. ACM Program. Lang. **4**(OOPSLA), 129:1–129:29 (2020). https://doi.org/10.1145/3428197, https://hal.archives-ouvertes.fr/hal-02185883
24. Six, C., Gourdin, L., Boulmé, S., Monniaux, D., Fasse, J., Nardino, N.: Formally Verified Superblock Scheduling. In: Certified Programs and Proofs (CPP '22). Philadelphia, United States (Jan 2022). https://doi.org/10.1145/3497775.3503679, https://hal.archives-ouvertes.fr/hal-03200774
25. Tristan, J.B., Leroy, X.: Formal verification of translation validators: A case study on instruction scheduling optimizations. In: Proceedings of the 35th ACM Symposium on Principles of Programming Languages (POPL'08). pp. 17–27. ACM Press (Jan 2008), http://xavierleroy.org/publi/validation-scheduling.pdf
26. Yang, X., Chen, Y., Eide, E., Regehr, J.: Finding and understanding bugs in C compilers. In: Programming Language Design and Implementation (PLDI). pp. 283–294. Association for Computing Machinery (2011). https://doi.org/10.1145/1993498.1993532

27. Zhou, Z., Ren, Z., Gao, G., Jiang, H.: An empirical study of optimization bugs in gcc and llvm. Journal of Systems and Software **174**, 110884 (2021). https://doi.org/https://doi.org/10.1016/j.jss.2020.110884, https://www.sciencedirect.com/science/article/pii/S0164121220302740

A Appendix

A.1 An example of reduction predicate

```
if ! aarch64-linux-gnu-gcc -fno-sanitize-recover -fsanitize=
    address -fsanitize=undefined -Wall -Werror=format -Werror
    =implicit -Werror=uninitialized -Werror=return-type -
    Werror=main -Werror=missing-prototypes -Werror=int-
    conversion -I . source.c -o source.gcc-sanitized.aarch64
    2> source.gcc-sanitized.err ;
then exit 1; fi
if ! ASAN_OPTIONS='detect_leaks=0' qemu-aarch64 -L /usr/
    aarch64-linux-gnu ./source.gcc-sanitized.aarch64 2>&1 >
    source.gcc-sanitized.out ;
then exit 2 ; fi
if ! grep -q checksum source.gcc-sanitized.out ;
then exit 3 ; fi
if ! gcc source.c -o source.gcc.amd64 ;
then exit 5; fi
if ! valgrind --error-exitcode=42 ./source.gcc.amd64 ;
then exit 6 ; fi
if ! ccomp -fstruct-passing -fbitfields -fno-cse2 -fno-cse -
    fno-cse3 -I . source.c -o source.ccomp.aarch64 ;
then exit 4 ; fi
qemu-aarch64 -L /usr/aarch64-linux-gnu ./source.ccomp.aarch64
     2>&1 > source.ccomp.out
! cmp source.gcc-sanitized.out source.ccomp.out
```

Fig. 1. Example of a predicate for reducing test cases produced by Csmith

A.2 Experimental evaluation of our CompCert version

We compared performance of generated code with GCC -01 (11.3.0) and our CompCert version versus the mainline CompCert (3.12). Our version includes several additional optimizations not present in the mainline CompCert: superblock prepass scheduling [24], CSE3 (Common Subexpression Elimination) [14], a variant of LCM (Lazy Code Motion) & LSR (Lazy Strength-Reduction) [7,8], and loop-unrollings (here, we only activated the first loop iteration unrolling).

The execution time gain is computed using the formula $gain(C) = ((R - C)/C) \times 100$ where C is the new compiler (e.g. GCC or ours) and R the reference one (e.g. mainline CompCert); a higher result indicates a better performance. To obtain representative and not biased measurements, we filtered results whose relative standard derivation was exceeding 2% over five runs on the target machine (as explained in §6). All our benchmarks were run on a SiFive U740 core (HiFive

Table 1. GCC and our fork versus mainline CompCert on RISC-V — Higher is better

Setup	gcc-O1	**Our CompCert fork**
LLVMtest/fpconvert	+24.22%	+17.15%
LLVMtest/matmul	+15.9%	+144.11%
LLVMtest/nbench_bf	+74.58%	+24.51%
MiBench/jpeg	+27.75%	+24.75%
MiBench/sha	+92.43%	+51.73%
MiBench/stringsearch	+133.34%	-10.15%
PolyBench/*	+64.05%	+46.23%
TACLeBench/bsort	+49.04%	+33.16%
TACLeBench/deg2rad	+56.75%	+50.28%
TACLeBench/md5	+42.18%	+47.93%

Unmatched), a dual-issue, in-order RISC-V processor, using the toolkit and test suites described in §6.

Globally, our version is 20.7% faster than the mainline CompCert on the LLVM test suite; 14.7% on MiBench; 22.83% on TACLeBench; and 42% on our own suite (the fifth one of §6); results for the full PolyBench suite are at line "PolyBench/*" in Table 1. A sample of results is provided in Table 1, illustrating the performance variability depending on the input: GCC is far more performant on some benchmarks, but way slower on some others, and the same applies to CompCert. For instance, the poor result on MiBench/Stringsearch comes from the loop-unrolling, which lengthens the code and makes other optimizations dramatically increasing register pressure. On this benchmark, if we disable the first loop iteration unrolling, our CompCert version yields a gain of 39.4% w.r.t. mainline CompCert. Nonetheless, except on rare tests like this one, unrolling is usually very beneficial in terms of performance. Overall, we improved the performance of about 30% in average on RISC-V comparing to the mainline version, which is a great step towards closing the gap with GCC.