Correctness Testing of Loop Optimizations in C and C++ Compilers

van Veen, Remi; Beemster, Marcel; Grelck, C.U.

Publication date
2018

Document Version
Final published version

Published in
11th Seminar on Advanced Techniques and Tools for Software Evolution (SATToSE 2018)

License
Other

Correctness Testing of Loop Optimizations in C and C++ Compilers

Remi van Veen
University of Amsterdam
The Netherlands
remi.vanveen@student.uva.nl

Marcel Beemster
Solid Sands BV
The Netherlands
marcel@solidsands.nl

Clemens Grelck
University of Amsterdam
The Netherlands
c.grelck@uva.nl

Abstract

Test coverage is often measured by the number of source lines of code that is executed by tests. However, compilers can apply transformations to the code to optimize the performance or size of a program. These transformations can remove parts of the original code, but they can also add new code by creating specialized copies and additional conditional branches. This means that while at source code level it seems as if all code would be tested, it is quite possible that the actually executed machine code is only partially tested. This project investigates how confidence in the correctness of a compiler's optimizations can be improved. To this end we create a suite of programs that explicitly trigger compiler optimizations and test their correctness with coverage on machine code level. Once confidence into the correctness of the compiler has been established, it becomes sufficient to test application software with high source code coverage only.

1 Introduction

In the safety-critical domain, software is tested according to safety standards such as the ISO 26262 [9] functional safety standard for automotive software. This standard requires safety-critical software to be tested with MC/DC coverage (*modified condition/decision coverage*) [8]. While compilers are not in the car itself, the ISO standard describes that confidence should be created in the correctness of tools such as compilers because of their significant role in the creation of executable code.

Proceedings of the Seminar Series on Advanced Techniques and Tools for Software Evolution SATToSE 2018 (sattose.org). 04-06 July 2018, Athens, Greece.

Much research has been done on theoretically proving the correctness of compiler optimizations [4, 12, 13]. However, a theoretical proof of an algorithm does not guarantee a correct implementation of the algorithm as errors could be introduced at implementation level. Therefore, while theoretically proving compiler optimizations to be correct is essential, testing the correctness of their implementation within a compiler is just as important.

To create confidence in the correctness of a compiler, producers of safety-critical software use compiler test suites. One such example is *SuperTest* [17], a C/C++ test suite developed by Solid Sands. These test suites consist of large collections of unit tests that test the correctness of compilers according to the language specification.

A crucial quality metric for any test suite is test coverage. Usually, test coverage is measured by the fraction of source code lines executed by tests. As long as compilers more or less directly translate source code to machine code, this is an accurate metric. However, compilers typically apply optimizations aiming at improving non-functional properties of the resulting code, such as runtime performance, code size or energy consumption. These transformations may remove parts of the original code, but they may likewise add new code by creating specialized copies and additional conditional branches. Consequently, test coverage of source code and test coverage of compiled code may differ.

The example in Listing 1 illustrates this phenomenon. The compiler transforms the original loop by loop unrolling, reducing the number of jumps executed by the machine code. A new conditional branch is introduced below the loop, which handles the remaining loop iteration if n is not a multiple of 2. For the original loop any $n > 2$ would result in 100% code coverage when testing. For the optimized code, however, any n that is a multiple of 2 would cause the newly introduced conditional branch not to be tested.

```
// Original loop
for(int i = 2; i < n-1; i++) {
    a[i] = a[i] + a[i-1] * a[i+1];
}

// After unrolling
for(int i = 2; i < n-2; i += 2) {
    a[i] = a[i] + a[i-1] * a[i+1];
    a[i+1] = a[i+1] + a[i] * a[i+2];
}
if((n-2) % 2 == 1) {
    a[n-1] = a[n-1] + a[n-2] * a[n];
}
```

Listing 1: Loop unrolling example [2].

Other examples of compiler optimizations that introduce new conditional branches to the generated machine code are loop vectorization and strip mining [2]. These optimizations introduce multiple conditional branches to the machine code, including a remainder branch like in the loop unrolling example.

Since loop optimizations are performed by the compiler, the newly introduced conditional branches are only present in the generated machine code. Consequently, no test inputs to cover these branches can be deduced from the source code. While at source code level it seems like all code is tested, the actually executed machine code is only partially tested. In other words, whereas source code is shown to be correct by tests, the introduced compiler optimizations are not.

One may argue that to overcome this problem producers of safety-critical software could simply disable all compiler optimizations when building their binaries. However, non-functional properties of software, namely execution speed and (thus) power consumption, are crucial for the usability and, hence, commercial success of software as well. Compiler optimizations significantly contribute to this. Slower software would require better (stronger microprocessors, more cores) hardware to still meet soft or hard performance targets. Better hardware is more expensive and consumes more energy. To summarize, disabling compiler optimizations is usually not an option as it severely diminishes the competitiveness of the product.

Alternatively, producers of safety-critical software could test their software at machine code level instead of at source code level, but this is hard and could introduce redundancy, as the same optimization might be applied multiple times. Instead, if the compiler itself is tested for correctly implementing optimizations, confidence in the compiler is created such that it is (indeed) sufficient to test a software project at source code level instead of at machine code level.

In the remainder of this paper we investigate how such a test suite can be designed and implemented. For the time being we focus on loop optimizations [2], but our ultimate goal is a testing methodology that can be applied to other compiler optimizations just as well. A particular challenge for the design of a test suite that fully covers the optimizations performed by some compiler is that we cannot derive any test cases from the language specification. The problem here is that any language specification merely specifies the functional behavior of code, but does not state anything about the implementation of the compiler, its internal processes or potential optimizations. Therefore, a novel method for creating appropriate compiler test suites is needed.

The following questions guide our research:

- How can we design tests that target specific compiler optimizations?
- How can we identify conditional branches introduced by compiler optimizations, such that test inputs that fully cover the machine code can be selected?
- How can we measure test coverage of a program at machine code level?
- How large is the gap between test coverage at source code level and test coverage at machine code level?

2 Related work

CompCert is a C compiler that is claimed to be formally verified using machine-assisted mathematical proofs [14]. The compiler does not apply loop optimizations, which means that the produced machine code can be significantly slower than machine code produced by other compilers. Furthermore, using the SuperTest compiler test suite a semantics bug in machine code produced by the compiler was detected [3]. This shows that even for a compiler that focuses on formally proving its correctness, externally testing it using a systematic test suite could detect previously unknown bugs in its implementation.

To find bugs caused by incorrect compiler optimizations, Yang et al. use random program generation with their tool Csmith [19]. While the authors found many bugs using this methodology, it is not a systematic approach to testing the correctness of optimizations. As programs are generated randomly, bugs are found by chance and there is not a systematic level of coverage achieved on either the tested compiler or the generated test programs. Thus, this methodology is not suitable to integrate into systematic compiler test suites such as SuperTest [17].

Jaramillo et al. test the correctness of compiler optimizations by comparing the semantics of an optimized program with the semantics of the corresponding unoptimized program at runtime [10]. While the authors introduce an accurate way of testing compiler optimizations, they do not mention what input programs are used for testing or what test coverage is achieved on them.

Similarly, Necula validates optimizations performed by the gcc compiler by verifying the preservation of semantics between the intermediate form of a program before and after each compiler pass [16]. This methodology relies on the intermediate representation structure of gcc and is therefore not a generic methodology that can be applied to any compiler. Also Necula does not mention what input programs are used for testing or what test coverage is achieved.

Other research papers on compiler optimization testing do provide the input programs they use. Callahan et al. provide the Test Suite for Vectorizing Compilers (TSVC), a Fortran benchmark for compiler optimizations [6]. Maleki et al. adapted TSVC for benchmarking C compilers and added additional loops [15]. Both studies, however, only use TSVC to benchmark compiler optimizations and not to test them for correctness.

3 Research methodology

We investigate how test programs can be designed that specifically test compiler loop optimizations for correctness. These test programs should achieve test coverage at machine code level, such that all optimized machine code generated by a compiler is tested. This way, confidence in the correctness of the optimizations applied by the compiler can be created, as for example required by the ISO 26262 functional safety standard for automotive software. In the remainder of this section we discuss the methodology for creating such test programs.

3.1 Test program design

To create loop optimization test programs, code that specifically triggers these optimizations is needed. The Test Suite for Vectorizing Compilers (TSVC) for C by Maleki et al. consists of 151 loops that are designed to trigger various kinds of loop optimizations and therefore are a useful starting point for creating such tests [15]. These benchmarks are only designed to measure the performance of the optimized machine code and do not test it for correctness. Therefore, they do not aim at any particular level of code coverage upon executing. Hence, we adapt these benchmarks to use them for correctness testing instead. We select test variables that cover the optimized machine code of the benchmarks and validate the results of executing the benchmarks.

To validate the results, we use comparison checking, which is based on the work of Jaramillo et al. [10]. This means a test program is compiled with and without optimizations enabled and the results of executing both programs are verified to be the same.

3.2 Loop selection

TSVC consists of 151 loops. In the scope of this project it is not feasible to adapt all loops for correctness testing. All TSVC loops are labeled by the optimization they aim to trigger, and multiple loops aim to trigger the same optimization. Therefore, we make a careful selection of loops that cover a wide range of optimizations while keeping the engineering effort reasonable.

As all TSVC benchmarks are bundled in a single file and use global variables, the benchmarks are first isolated to small and modular test files. Then, the benchmarks are checked for undefined behavior in order to guarantee that they are correctly functioning programs suitable for correctness testing.

3.3 Test input selection

To create test inputs that cover compiler optimizations at machine code level, the newly introduced conditional branches to the machine code of the test programs need to be discovered such that test inputs that trigger these conditional branches can be selected. We investigated the possibility to use concolic execution tools like KLEE [5] to automatically generate test inputs, but such tools work at source code level instead of at machine code level. Upon that, while concolic execution tools are able to find numeric test inputs needed to cover a loop branch, they are not able to find test inputs that cover more complex branches that for example require a pointer address to be aligned by a specific number of bytes.

To perform static analysis on machine code level, Křoustek transforms the machine code back to a human-readable programming language using a decompiler and analyses the resulting code [11]. The decompiler is actively maintained and is open source. Snowman, another actively maintained decompiler by Troshina et al., is also open source and supports the X86-64 instruction set [18].

Because of its X86-64 support we use Snowman, at least for the time being. We compile the test programs at different levels of optimization and decompile the resulting binary file with Snowman. We then analyze the decompiled code to discover what test inputs are needed to trigger all conditional branches introduced by the compiler. While the decompiled code is often complex, it is still structured in if-then-else blocks and while-loops. This considerably facilitates analysis compared to assembly code with jumps to labels.

3.4 Robustness among different compilers

Compiler test suites used in the safety-critical software domain are designed to use for creating confidence in any C compiler. As each compiler can implement optimizations differently, full machine code coverage for a test program with certain input values compiled by one compiler does not guarantee full machine code coverage when using another compiler. For example, different compilers can use different loop unroll factors or select the loop unroll factor dynamically. Sufficient test input values need to be selected to cover the optimizations of compilers in general as widely as possible, so robust tests are created that not only target one specific compiler. We do this by recognizing regular patterns in compiler optimizations and creating test inputs that cover these patterns.

3.5 Machine code coverage measurement

Commonly used test coverage measurement tools, such as Gcov [7], measure test coverage based on the source code of a program. As this project aims to create tests that cover optimized code at machine code level, test coverage needs to be measured at machine code level as well.

To do this, we use the GNATcoverage tool [1]. GNATcoverage allows coverage analysis of machine code on both instruction-level and branch-level. This is done by executing a program through the GNATcoverage tracing environment that keeps track of which instructions and conditional branches are covered during execution. The tool outputs the program's assembly code, marking every instruction with a coverage indicator. Simplified example output is provided in Listing 2. Here, "+" indicates an executed instruction or fully covered branch instruction, "-" indicates an instruction that was not executed, ">" indicates a branch instruction that was only covered on the true condition and "v" indicates a branch instruction that was only covered on the false condition. To calculate coverage metrics, we parse the GNATcoverage results and calculate these metrics using the coverage indicators.

```
+: cmpl   0x0,-0x8(rbp)
>: jge    0x4004cd <f+0x1d>
-: movl   0x0,-0x4(rbp)
v: jne    0x4004e3 <f+0x33>
```

Listing 2: Simplified example output of GNATcoverage machine code coverage analysis.

4 Preliminary results

Since we report on work in progress, we can only present partial results for now. First, we demonstrate the significance of testing compiler optimizations for correctness by showing the gap between source code coverage and machine code coverage at different compiler optimization levels. Next, we provide an example of what test cases are needed to robustly cover an unrolled loop at machine code level.

4.1 Gap between source code coverage and machine code coverage

We illustrate the gap between source code coverage and machine code coverage for the simple C function shown in Listing 3. The function is compiled by LLVM-based compiler Clang for X86-64 architecture at optimization levels -O0, -O1 and -O2.

To achieve full source code coverage, a single input $n > 0$ is sufficient. This way, all statements are executed and the loop branch is taken both ways. At level -O0 no optimizations are performed by the compiler, so the source code is a one-to-one representation of the generated machine code. This means also full instruction and branch coverage at machine code level is achieved for this input. The generated machine code consists of 19 instructions and 1 conditional branch.

At optimization level -O1, the generated machine code consists of 14 instructions, but an additional conditional branch is introduced by the compiler. This branch provides a shortcut if $n = 0$ and the loop thus does not need to be executed. If based on the source code input $n = 1$ is selected, instruction coverage drops to 93% while both branches are only covered on the false condition. Now to achieve full coverage at machine code level, different test inputs are needed: $n = 0$ to cover the newly introduced branch on the true condition and $n > 1$ to cover the loop condition both ways, as the loop is transformed into a do-while loop.

At optimization level -O2, the resulting machine code grows significantly in size and complexity. In addition to the shortcut introduced at level -O1, loop unrolling and vectorization are applied. The machine code consists of 77 instructions and 9 conditional branches. Now, if based on the source code input $n = 1$ is selected, instruction coverage drops to 18%, 3 branches are only covered on the false condition and 6 branches are not covered at all. To achieve full machine code coverage the function needs to be executed with 5 different inputs.

This example illustrates that based on the source code, it is impossible to select test inputs that achieve a high level of test coverage on optimized machine code level. Instead, analysis of the generated machine code is needed to achieve full machine code coverage. Even for this very simple 4-line function, machine code coverage drops from 100% at level -O0 to 18% at level -O2. As optimizations are applied to commonly used programming structures, machine code coverage is likely to significantly drop in real-world software projects as well. If compiler optimizations are not tested, a significant fraction of the executed code remains uncovered, despite high or even perfect test coverage at source code level.

```
int f(int n) {
    int total = 0;
    for (int i = 0; i < n; i++) {
        total += i & n;
    }
    return total;
}
```

Listing 3: Source code of a simple C function that can be optimized by a compiler.

4.2 Note on branch coverage at level -O2

In the previous example, at level -O2 full branch coverage is impossible to achieve, as the compiler introduces two unsatisfiable conditional branches. One of these unsatisfiable conditions can be explained as follows. The compiler adds a conditional branch for `n & 0xfffffff8 == 0` inside a conditional branch for $n >= 8$. The bitwise AND operation can never result in 0 as it rounds down n to the nearest multiple of 8 while $n >= 8$. The machine code thus still shows room for optimization, as the conditional jump could be replaced with an unconditional jump.

Unsatisfiable conditions like these mean that the test programs that are designed for this project may not always achieve full machine code coverage. However, if it can be proven that the uncovered conditions are impossible to satisfy, this is not a problem, as the untested code can never be executed.

4.3 Loop unrolling coverage

As explained in the introduction, a loop can be optimized by unrolling it to reduce the number of jumps needed to execute the loop. The loop body is replicated a certain number of times and the loop increment is changed to the same number, which is called the unroll factor [2]. To take into account that the number of loop iterations might not be a multiple of the unroll factor, the compiler introduces a branch to handle remaining loop iterations below the unrolled loop.

When a loop is unrolled by factor 2, a single conditional branch is added for remainder handling as there can only be one remaining loop iteration that needs to be taken care of. However, for a larger unroll factor there can be more remaining loop iterations. While some compilers handle this by inserting a single loop branch, other compilers introduce nested conditional branches that explicitly check for every possible remainder value. This means that to robustly cover the remainder handling code of any compiler, test cases that trigger every possible remainder value are needed.

Usually, the maximum unroll factor that a compiler will apply can be retrieved from the compiler's configuration. We configure our test suite accordingly, such that we cover all possible unroll factors up to that value. To robustly cover the machine code of an unrolled loop, we use test inputs $n = [1..UF]$ where UF is the maximum loop unroll factor. For $n = UF$, full loop branch coverage is assured, as for this value the unrolled loop body is executed at least once for every possible unroll factor. The other test inputs ensure that every possible remainder branch is fully covered as well.

In our example demonstrating the gap between source code coverage and machine code coverage, we mentioned that the compiler introduces a shortcut branch for $n = 0$. When doing this, the compiler also transforms the loop into a do-while loop. To take this optimization into account as well, we also need $n = 0$ and $n = 2UF$ as test cases. For $n = 0$ the shortcut branch is covered, while for $n = 2UF$ the loop body is executed at least twice for every unroll factor. As the loop condition of a do-while loop is evaluated below the loop body, only this way full loop branch coverage is achieved. This shows that in order to fully cover a loop optimization, other optimizations also need to be taken into account, as a compiler might apply multiple optimizations to a single loop.

5 Conclusion

First results show that the gap between test coverage at source code level and test coverage at machine code level is significant when compiler optimizations are enabled. When compiling a simple function with Clang, instruction test coverage for the same test input drops from 100% on optimization level -O0 to 18% on optimization level -O2 because of loop optimizations.

Instead of testing all software generated by a compiler at machine code level, we investigate how a compiler itself can be tested for correctly implementing its optimizations. We do so by adapting the TSVC loop optimization benchmarks, that specifically trigger compiler optimizations, for correctness testing and covering them at machine code level. Our loop unrolling example demonstrates that we are able to identify patterns in compiler optimizations from which a range of test inputs can be selected that robustly cover such patterns.

What is not covered by such tests, is testing whether compiler optimizations are only applied to code that actually is suitable to optimize. Some loops are for example not suitable for optimization because of data dependencies. As most TSVC benchmarks are specifically designed to be optimized, such compiler errors are not detected by test programs based on these benchmarks. Instead, negative test cases are needed to test this. Also, the collection of TSVC benchmarks is not guaranteed to cover all loop optimizations that a compiler implements. The main goal of this project, however, is to develop a methodology that can be used to create a larger collection of tests.

References

[1] AdaCore. GNATcoverage. `https://github.com/AdaCore/gnatcoverage`.

[2] David F Bacon, Susan L Graham, and Oliver J Sharp. Compiler Transformations for High-Performance Computing. *ACM Computing Surveys (CSUR)*, 26(4):345–420, 1994.

[3] Marcel Beemster. C99 for-loop defined variable gets incorrect scope. `https://github.com/AbsInt/CompCert/issues/211`, 2017.

[4] Nick Benton. Simple Relational Correctness Proofs for Static Analyses and Program Transformations. *ACM SIGPLAN Notices*, 39(1):14–25, 2004.

[5] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI*, volume 8, pages 209–224, 2008.

[6] David Callahan, Jack Dongarra, and David Levine. Vectorizing Compilers: A Test Suite and Results. In *Proceedings of the 1988 ACM/IEEE conference on Supercomputing*, pages 98–105. IEEE Computer Society Press, 1988.

[7] Gcov. `https://gcc.gnu.org/onlinedocs/gcc/Gcov.html`.

[8] Kelly J Hayhurst, Dan S Veerhusen, John J Chilenski, and Leanna K Rierson. A Practical Tutorial on Modified Condition/Decision Coverage. *NASA Report, NASA/TM-2001-210876*, 2001.

[9] ISO 26262-6:2011. Road vehicles – Functional safety – Part 6: Product development at the software level. Standard, International Organization for Standardization, Geneva, Switzerland, November 2011.

[10] Clara Jaramillo, Rajiv Gupta, and Mary Lou Soffa. Debugging and Testing Optimizers through Comparison Checking. *Electronic Notes in Theoretical Computer Science*, 65(2):83–99, 2002.

[11] Jakub Křoustek. *Retargetable Analysis of Machine Code*. PhD thesis, Brno University of Technology, Czech Republic, 2014.

[12] David Lacey, Neil D Jones, Eric Van Wyk, and Carl Christian Frederiksen. Proving correctness of compiler optimizations by temporal logic. *ACM SIGPLAN Notices*, 37(1):283–294, 2002.

[13] Sorin Lerner, Todd Millstein, and Craig Chambers. Automatically Proving the Correctness of Compiler Optimizations. *ACM SIGPLAN Notices*, 38(5):220–231, 2003.

[14] Xavier Leroy. A Formally Verified Compiler Backend. *Journal of Automated Reasoning*, 43(4):363, 2009.

[15] Saeed Maleki, Yaoqing Gao, Maria J Garzar, Tommy Wong, David A Padua, et al. An Evaluation of Vectorizing Compilers. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 372–382. IEEE, 2011.

[16] George C Necula. Translation Validation for an Optimizing Compiler. In *ACM sigplan notices*, volume 35, pages 83–94. ACM, 2000.

[17] Solid Sands. SuperTest. `https://solidsands.nl/supertest`.

[18] Katerina Troshina, Alexander Chernov, and Yegor Derevenets. C Decompilation: Is It Possible? In *Proceedings of International Workshop on Program Understanding, Altai Mountains, Russia*, pages 18–27, 2009.

[19] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and Understanding Bugs in C Compilers. In *ACM SIGPLAN Notices*, volume 46, pages 283–294. ACM, 2011.