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 like all code is tested, it is possible the actually executed machine code is only partially tested. This project investigates how confidence in the correctness of a compiler's optimizations can be created, by creating programs that trigger compiler optimizations and test them with coverage on machine code level. By creating such confidence, it is sufficient to test software generated by the compiler only on source code level.

1 Introduction

In the safety-critical domain, software is tested according to safety standards such as the ISO 26262 [ISO11] functional safety standard for automotive software. This standard requires safety-critical software to be tested with MC/DC coverage (*modified decision/-condition coverage*) [HVCR01]. 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.

Much research has been done on theoretically proving the correctness of compiler optimizations [Ben04, LJVWF02, LMC03]. 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 into a compiler is just as important.

To create confidence in the correctness of a compiler, producers of safety-critical software use compiler test suites [San, Hal]. One such example is *SuperTest*, a C/C++ test suite developed by SolidSands. 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 number of source lines of code that is executed by tests. As long as compilers more or less directly translate source code to machine code, this is an accurate metric, that indicates what fraction of the code is tested. However, compilers typically also 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.

```
// 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 [BGS94].

The example in Listing 1 illustrates this phenomenon. The compiler transforms the original loop by loop unrolling, reducing the number of jumps exe-

cuted by the machine code. A new conditional branch is introduced below the loop, which is executed 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.

Since optimizations like loop unrolling are performed by the compiler, the newly introduced conditional branch is only present in the generated machine code. Consequently, no test input to cover this branch 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 optimization is not.

Achieving full coverage on the optimized machine code of a large-scale software project is hard and could introduce redundancy, as the same optimization might be applied multiple times. Instead, if the compiler itself has been 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 [BGS94], 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

Yang et al. use random program generation with their tool Csmith to find bugs caused by compiler optimizations [YCER11]. 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. Because of that, this methodology is not suitable to integrate into systematic compiler test suites such as SuperTest.

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 [JGS02]. 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 comparing the intermediate form of a program before and after each compiler pass and verifying the preservation of semantics [Nec00]. 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 on them.

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 [CDL88]. Maleki et al. adapted TSVC for benchmarking C compilers and added additional loops [MGG+11]. Both studies, however, only use TSVC to benchmark compiler optimizations and not to test them for correctness. Because of that, these programs are also not designed to achieve a certain level of coverage when executing them.

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 [MGG+11]. These benchmarks are only designed to measure the performance of the optimized machine code and do not test it for correctness. Because of that, they are also not designed to achieve a certain level of coverage when executing them. For this project, these benchmarks are adapted to use for correctness testing instead, by selecting test variables that cover the optimized machine code of the benchmarks and adding code to validate the results of executing them.

To validate the results, predefined expected results or properties of executing a program are compared to the actual results of executing the program. A possible way of doing this is the methodology used by Jaramillo et al. They test the correctness of compiler optimizations by compiling the program with and without optimizations enabled and comparing the final and intermediate results of executing both programs [JGS02].

3.2 Loop selection

As TSVC consists of 151 loops, in the scope of this project it is not achievable to adapt all loops for correctness testing. All TSVC loops are labeled by the optimization they are designed for to trigger, and multiple loops are designed for the same optimization. A selection of these loops is made that covers a wide range of optimizations, while the number of tests programs remains comprehensible.

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. To perform static analysis at machine code, Křoustek and Kolář suggest transforming the machine code back to a human-readable programming language using a decompiler and performing analysis on the resulting code [KK14]. The decompiler developed by these authors 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 [TCD09].

Because of its X86-64 support, for the time being we use Snowman. 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 structured in if-then-else blocks and while-loops, which makes analysis easier than for assembly code.

3.4 Robustness among different compilers

Compiler test suites that are 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 for the same input values when the program is compiled by another compiler. For example, different compilers can use different loop unroll factors. 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 those patterns.

3.5 Machine code coverage measurement

Commonly used test coverage measurement tools, such as Gcov [Gco], 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 [Ada]. 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 of GNATcoverage 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 ∨ indicates a branch instruction that was only covered on the false condition. To calculate coverage metrics, we parse the results produced by GNATcoverage 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. 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.

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.

```
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.

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 maximal machine code coverage, now 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 when compiler optimizations are enabled. If compiler optimizations are not tested, while test coverage at source code level can be high, a significant fraction of the executed code remains untested.

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.

5 Discussion

First results show that the gap between test coverage at source code level and test coverage at machine code level is significant with compiler optimizations 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 aim to do so by adapting the TSVC loop optimization benchmarks that specifically trigger compiler optimizations for correctness testing and covering them at machine code level.

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 developing a methodology that can be used to create a larger collection of tests.

References

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

[Ben04] Nick Benton. Simple relational correctness proofs for static analyses and program transformations. *ACM SIGPLAN Notices*, 39(1):14–25, 2004.

[BGS94] 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.

[CDL88] 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.

[Gco] Gcov - a test coverage program. `https://gcc.gnu.org/onlinedocs/gcc/Gcov.html`.

[Hal] Plum Hall. C and C++ Validation Test Suites. `http://www.plumhall.com/suites.html`.

[HVCR01] K 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.

[ISO11] ISO 26262-6:2011. Road vehicles – Functional safety – Part 6: Product development at the software level. Standard, International Organization for Standardization, Geneva, CH, November 2011.

[JGS02] 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.

[KK14] Jakub Křoustek and D Kolář. *Retargetable Analysis of Machine Code*. PhD thesis, Brno University of Technology, 2014.

[LJVWF02] 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.

[LMC03] Sorin Lerner, Todd Millstein, and Craig Chambers. Automatically proving the correctness of compiler optimizations. *ACM SIGPLAN Notices*, 38(5):220–231, 2003.

[MGG+11] 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.

[Nec00] George C Necula. Translation validation for an optimizing compiler. In *ACM sigplan notices*, volume 35, pages 83–94. ACM, 2000.

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

[TCD09] 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.

[YCER11] 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.