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

**Remi van Veen**
remi.vanveen@student.uva.nl

July 11, 2018, 36 pages

**Supervisor:** Dr. Clemens Grelck
**Host organisation:** Solid Sands B.V.
**Host supervisor:** Dr. Marcel Beemster

Universiteit van Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Master Software Engineering
http://www.software-engineering-amsterdam.nl

# Contents

# 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. Hence, especially in the safety-critical software domain, utilizing compiler optimizations without creating confidence in their correctness introduces significant risk. 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 at machine code level. We base these test programs on existing loop optimization benchmarks, which are designed to trigger specific compiler optimizations, but do not test them for correctness. By compiling the benchmarks and transforming the optimized compiled code back to a human-readable programming language using a decompiler, we can analyze the code introduced by compiler optimizations and select test inputs that cover them at machine code level. To create robust test cases that do not only cover the optimizations applied by one specific compiler, we extract recognizable patterns from the behavior of multiple compilers and select test inputs that cover these patterns. This way our test suite is designed to be used with any C or C++ compiler. Once confidence into the correctness of the compiler has been established, it becomes sufficient to test application software with high source code coverage only.

# Chapter 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 condition/decision 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.

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 Solid Sands. The test suite consists of a large collection of test programs that test the correctness of a compiler based on the C/C++ language standard. This way, confidence in the conformance of a compiler with the language standard can be established.

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.

The example in Listing 1.1 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 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.

```
1  // Original loop
2  for(int i = 0; i < n; i++) {
3      a[i] = a[i] + 1;
4  }
5
6  // After unrolling
7  for(int i = 0; i < n-1; i += 2) {
8      a[i] = a[i] + 1;
9      a[i+1] = a[i+1] + 1;
10 }
11
12 if(n % 2 == 1) {
13     a[n-1] = a[n-1] + 1;
14 }
```

Listing 1.1: Loop unrolling example [BGS94].

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.

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. However, based on the source code of a program it is impossible to select test inputs that fully cover the optimized machine code. Analysis of the generated machine code would be required instead, which is hard as compiler optimizations can significantly increase the size and complexity of the generated code. 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 this thesis, we investigate how such a test suite can be designed and implemented. We do this in collaboration with Solid Sands, as this could help improve the test coverage of compiler optimizations by SuperTest. 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, so it 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 programs is needed.

The following questions guide our research:

- How can we design test programs 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?

Our approach is as follows. We use small test programs that target specific loop optimizations and compile them at different optimization levels. We analyze the resulting binaries to find what code is introduced by compiler optimizations. From this, we select test inputs that ensure that the code introduced by compiler optimizations is fully covered, such that we test the applied optimizations at machine code level. By doing this for a broad range of loop optimizations, a compiler test suite can be created that can be used to create confidence in the correctness of the loop optimizations that a compiler can apply.

In the next chapter, we provide a motivating example for our research by demonstrating the gap between test coverage on source code level and test coverage on machine code level. Next, we describe our methodology for creating compiler optimization tests in Chapter 3. We demonstrate the implementation of our test suite in Chapter 4, followed by an evaluation of our test suite in Chapter 5. Limitations and future work are discussed in Chapter 6 and 7. We discuss related work to our project in Chapter 8. Finally, conclusions are drawn in Chapter 9.

# Chapter 2

# Motivating Example

To illustrate why it is important to create confidence in the correctness of compiler optimizations, in this chapter we demonstrate the gap between test coverage on source code level and test coverage on optimized machine code level. We do this for the simple C function shown in Listing 2.1. To achieve full source code coverage on this function, a single input `n > 0` is sufficient. We use `n = 1` as an example. For this input all statements are executed and the loop branch is taken both ways. However, we will demonstrate that this coverage does not hold for the executed machine code when compiler optimizations are enabled. We do this by compiling the function at different optimization levels using the LLVM-based compiler Clang for X86-64 architecture and analyzing the generated machine code.

```
1 int f(int n) {
2     int total = 0;
3     for (int i = 0; i < n; i++) {
4         total += i & n;
5     }
6     return total;
7 }
```

Listing 2.1: Source code of a simple C function that can be optimized by a compiler.

## 2.1 Optimization level -O0

At optimization 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 our test input. The generated machine code consists of 19 instructions and 1 conditional branch. The corresponding assembly code is listed in Listing 2.2. To the left of each instruction, a coverage indicator shows whether the instruction was executed for our selected test input. As full coverage is achieved, all instructions are marked by a "+".

```
+:  push   rbp
+:  mov    rsp,rbp
+:  mov    edi,-0x4(rbp)
+:  movl   0x0,-0x8(rbp)
+:  movl   0x0,-0xc(rbp)
+:  mov    -0xc(rbp),eax
+:  cmp    -0x4(rbp),eax
+:  jge    0x40051b <f+0x3b>
+:  mov    -0xc(rbp),eax
+:  and    -0x4(rbp),eax
+:  add    -0x8(rbp),eax
+:  mov    eax,-0x8(rbp)
+:  mov    -0xc(rbp),eax
+:  add    0x1,eax
+:  mov    eax,-0xc(rbp)
+:  jmpq   0x4004f5 <f+0x15>
+:  mov    -0x8(rbp),eax
+:  pop    rbp
+:  retq
```

Listing 2.2: Assembly code corresponding to the function of Listing 2.1 compiled by Clang at optimization level -O0, with coverage indicators for input `n = 1`.

## 2.2 Optimization level -O1

At optimization level -O1, the number of instructions in the generated machine code goes down to 14, 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. For our test input of `n = 1`, 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 and `n > 1` to cover the loop branch both ways, as the loop is transformed into a do-while loop. Simplified assembly code with coverage indicators for test input `n = 1` is provided in Listing 2.3. The "-" indicates an instruction that is not executed, while the "v" indicates a conditional branch instruction that is only covered on the false condition.

```
+:  test   edi,edi
v:  jle    0x4004fe <f+0x1e>
+:  xor    ecx,ecx
+:  xor    eax,eax
+:  nopl   0x0(rax,rax,1)
+:  mov    ecx,edx
+:  and    edi,edx
```

```
+:  add    edx,eax
+:  inc    ecx
+:  cmp    ecx,edi
v:  jne    0x4004f0 <f+0x10>
+:  jmp    0x400500 <f+0x20>
-:  xor    eax,eax
+:  retq
```

Listing 2.3: Assembly code corresponding to the function of Listing 2.1 compiled by Clang at optimization level -O1, with coverage indicators for input `n = 1`.

## 2.3 Optimization level -O2

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. The corresponding assembly code with coverage indicators is listed in Listing 2.4. Now, for our test input of `n = 1` 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 of a function, 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 by the compiler is needed to achieve full machine code coverage. For this very simple 4-line function, machine code coverage already drops from 100% on level -O0 to 18% on 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 a compiler is not tested for correctness of these optimizations, while test coverage at source code level can be high, a significant fraction of the generated machine code remains untested. Hence, especially in the safety-critical software domain, utilizing compiler optimizations without creating confidence in their correctness introduces significant risk.

```
+:  test   edi,edi
v:  jle    0x4004c2 <f+0x12>
+:  xor    edx,edx
+:  cmp    0x7,edi
v:  ja     0x4004c5 <f+0x15>
+:  xor    eax,eax
+:  jmpq   0x4005d0 <f+0x120>
-:  xor    eax,eax
-:  retq
-:  mov    edi,ecx
-:  and    0xfffffff8,ecx
-:  mov    0x0,eax
-:  je     0x4005d0 <f+0x120>
-:  movd   edi,xmm0
-:  pshufd 0x0,xmm0,xmm0
-:  lea    -0x8(rcx),edx
-:  mov    edx,eax
-:  shr    0x3,eax
-:  bt     0x3,edx
-:  jb     0x40051a <f+0x6a>
-:  movdqa 0x17c(rip),xmm1
-:  pand   xmm0,xmm1
-:  movdqa 0x180(rip),xmm3
-:  pand   xmm0,xmm3
-:  movdqa 0x184(rip),xmm5
-:  mov    0x8,edx
-:  test   eax,eax
-:  jne    0x400530 <f+0x80>
-:  jmpq   0x4005a7 <f+0xf7>
-:  pxor   xmm1,xmm1
-:  movdqa 0x14a(rip),xmm5
-:  xor    edx,edx
-:  pxor   xmm3,xmm3
-:  test   eax,eax
-:  je     0x4005a7 <f+0xf7>
-:  mov    ecx,eax
-:  sub    edx,eax
-:  movdqa 0x163(rip),xmm8
-:  movdqa 0x16a(rip),xmm9
```

```
-:  movdqa 0x172(rip),xmm6
-:  movdqa 0x17a(rip),xmm7
-:  nopw   cs:0x0(rax,rax,1)
-:  movdqa xmm5,xmm2
-:  paddd  xmm8,xmm2
-:  movdqa xmm5,xmm4
-:  pand   xmm0,xmm4
-:  pand   xmm0,xmm2
-:  paddd  xmm1,xmm4
-:  paddd  xmm3,xmm2
-:  movdqa xmm5,xmm1
-:  paddd  xmm9,xmm1
-:  movdqa xmm5,xmm3
-:  paddd  xmm6,xmm3
-:  pand   xmm0,xmm1
-:  pand   xmm0,xmm3
-:  paddd  xmm4,xmm1
-:  paddd  xmm2,xmm3
-:  paddd  xmm7,xmm5
-:  add    0xfffffff0,eax
-:  jne    0x400560 <f+0xb0>
-:  paddd  xmm3,xmm1
-:  pshufd 0x4e,xmm1,xmm0
-:  paddd  xmm1,xmm0
-:  pshufd 0xe5,xmm0,xmm1
-:  paddd  xmm0,xmm1
-:  movd   xmm1,eax
-:  cmp    edi,ecx
-:  mov    ecx,edx
-:  je     0x4005dc <f+0x12c>
-:  nopw   0x0(rax,rax,1)
+:  mov    edx,ecx
+:  and    edi,ecx
+:  add    ecx,eax
+:  inc    edx
+:  cmp    edx,edi
v:  jne    0x4005d0 <f+0x120>
+:  retq
```

Listing 2.4: Assembly code corresponding to the function of Listing 2.1 compiled by Clang at optimization level -O2, with coverage indicators for input `n = 1`.

# Chapter 3

# Research Methodology

In this chapter, we discuss our methodology for the development of our test suite. First, we discuss how we design test programs that target specific loop optimizations. Next, we discuss how we analyze test coverage at machine code level and how we select test inputs that lead to full machine code coverage of an optimized program. Finally, we explain how we test an optimized program for correctness.

## 3.1 Loop sources

To create loop optimization test programs, we need code that specifically triggers such optimizations. The Test Suite for Vectorizing Compilers (TSVC) is a collection of Fortran loop benchmarks that are designed to do that [CDL88]. Maleki et al. translated the benchmarks to C and expanded the collection to a total of 151 loops [MGG+11]. Therefore, these benchmarks are a useful starting point for our test suite. However, they are only designed to measure the performance of the optimized machine code and not to test it for correctness. Hence, they are also not designed to achieve a certain level of coverage when executing them. For this project, we adapt these benchmarks to use for correctness testing instead, by selecting test variables that cover the optimized benchmarks at machine code level and validating the results of executing them.

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

Besides using the TSVC benchmark suite as a source for our test loops, we also use loops from papers and online resources on compiler optimizations [BJL12, CGS+17, Sar15]. These sources are used to cover optimizations that are not triggered by any of the TSVC benchmarks.

## 3.2 Adapting benchmarks for correctness testing

As most of our test programs are based on TSVC benchmarks, in this section we demonstrate how we adapt these benchmarks for correctness testing. All TSVC benchmarks are bundled in a single file, so we first isolate the benchmarks to separate test files. Next, we make adjustments to the code such that they can be used for correctness testing.

Listing 3.1 shows an example of a TSVC loop benchmarking function. First of all, we remove code surrounding the benchmarked loop, such that the remaining function only contains the loop that is being tested. We also remove the outer loop that is only used to execute the benchmarked loop multiple times to get more accurate benchmarking results, as it is not needed to trigger the targeted optimization.

Next, instead of using global variables, we parameterize all loop variables. This allows for easier testing of the loop, as we can simply call the loop function with different test values. Furthermore, this way the loop iteration count and array dimensions are unknown at compile time, which means that the compiler needs to add runtime checks to the machine code. This can be explained using the

loop unrolling example from Chapter 1. If a loop is unrolled by factor 2, and the loop iteration count is known to be a multiple of 2, the compiler does not have to add the branch that is executed when the iteration count is not a multiple of 2. By leaving the loop iteration count unknown at compile time, the compiler needs to take into account that the iteration count is not always a multiple of 2 and thus adds the conditional branch that checks for this. These checks are of interest to our test suite, as they are examples of conditional branches that can not be deduced from the source code of the tested loop.

Finally, we use a `TYPE` macro to define the type of the pointer parameters passed to the tested function. This way, instead of only testing the loop with `float` pointers like TSVC does, we can test it with various data types by changing the macro value. As different data types need different CPU instructions, for example floating point instead of integer arithmetic instructions, this way we are able to test a broader range of optimizations by only changing a single macro value.

The resulting test function is displayed in Listing 3.2. Test loops that we collect from other sources than the TSVC benchmarks are formatted similarly. The test function is saved in a separate file, which is compiled at different optimization levels for correctness testing. The function is called from another file containing the `main()` function required to execute a C program. This file is not optimized by the compiler, such that no optimization errors can be introduced to the code that is responsible for the initialization of the input variables, calling of the tested function and freeing of allocated memory. This way, only the code from the tested function itself can be changed by any compiler optimization passes.

```
1  #define ntimes 200000
2  #define len 32000
3
4  float X[len], Y[len];
5
6  int s000() {
7      init("s000");
8      start_t = clock();
9
10     for (int nl = 0; nl < 2*ntimes; nl++) {
11         for (int i = 0; i < len; i++) {
12             X[i] = Y[i] + 1;
13         }
14
15         // Dummy function to make the computations appear required, such that
16         // the outer loop is not removed by compiler optimizations
17         dummy((float*)X, (float*)Y);
18     }
19
20     end_t = clock();
21     clock_dif = end_t - start_t;
22     clock_dif_sec = (double) (clock_dif/1000000.0);
23     printf("S000\t %.2f \t\t", clock_dif_sec);
24     check(1);
25     return 0;
26 }
```

Listing 3.1: Example of a TSVC loop benchmark function [MGG+11].

```
1  // Header file that contains the TYPE macro definition
2  #include "testopt.h"
3
4  int s000(TYPE *X, TYPE *Y, int len) {
5      for (int i = 0; i < len; i++) {
6          X[i] = Y[i] + 1;
7      }
8
9      return 0;
10 }
```

Listing 3.2: TSVC loop benchmark function of Listing 3.1 adapted for correctness testing.

## 3.3 Checking programs for undefined behavior

Before including a test program in our test suite, we check if it does not contain any undefined behavior such that only correctly functioning programs suitable for correctness testing are included in our test suite. Undefined behavior regarding memory access is relatively easy to check using Valgrind [Val], which instruments an executable to detect invalid memory access at runtime. Integer overflow can be detected using the Undefined Behavior Sanitizer of the Clang compiler [Und], which instruments a program at compile time, after which possible integer overflow is again detected at runtime.

## 3.4 Machine code coverage analysis

Commonly used test coverage measurement tools, such as Gcov [Gco], measure test coverage based on the source code of a program. As we aim to create tests that cover optimized code at machine code level, test coverage of our test programs needs to be measured at machine code level as well. To do this, we use the GNATcoverage tool [Adaa]. 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 3.3. 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.

```
400cd2 +: cmpl   0x0,-0x8(rbp)
400cd8 >: jge    0x400cdf <f+0x1d>
400cdc -: movl   0x0,-0x4(rbp)
400cdf v: jne    0x400ce3 <f+0x33>
```

Listing 3.3: Simplified example output of GNATcoverage machine code coverage analysis.

## 3.5 Test input selection

As demonstrated in Chapter 2, from the source code of a program it is impossible to select test inputs that fully cover the corresponding optimized machine code, because of new conditional branches that are introduced by the compiler. To create test inputs that cover compiler optimizations at machine code level, we need to analyze the machine code such that we can select test inputs that trigger these conditional branches. To perform static analysis on machine code, Křoustek et al. transform the machine code back to a human-readable programming language using their *retargetable decompiler* and analyze the resulting code [Křo14, KP13]. Their 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 [TCD09, Der18].

Because of its X86-64 support, we use Snowman. We compile the test programs at different levels of optimization and decompile the resulting binary file with Snowman. While the decompiled code is often more complex than human-written code, it is still structured in if-then-else blocks and while-loops. This considerably facilitates analysis compared to assembly code with jumps to labels. This way also variables are traceable to their origin by following all assignment statements in the decompiled code, which is much simpler than analyzing variable state based on the assembly code. Figure 3.1 shows an example of Snowman's graphical user interface. By selecting an assembly instruction on the left, the corresponding decompiled code is highlighted on the right, such that an instruction can be traced to its corresponding decompiled code.

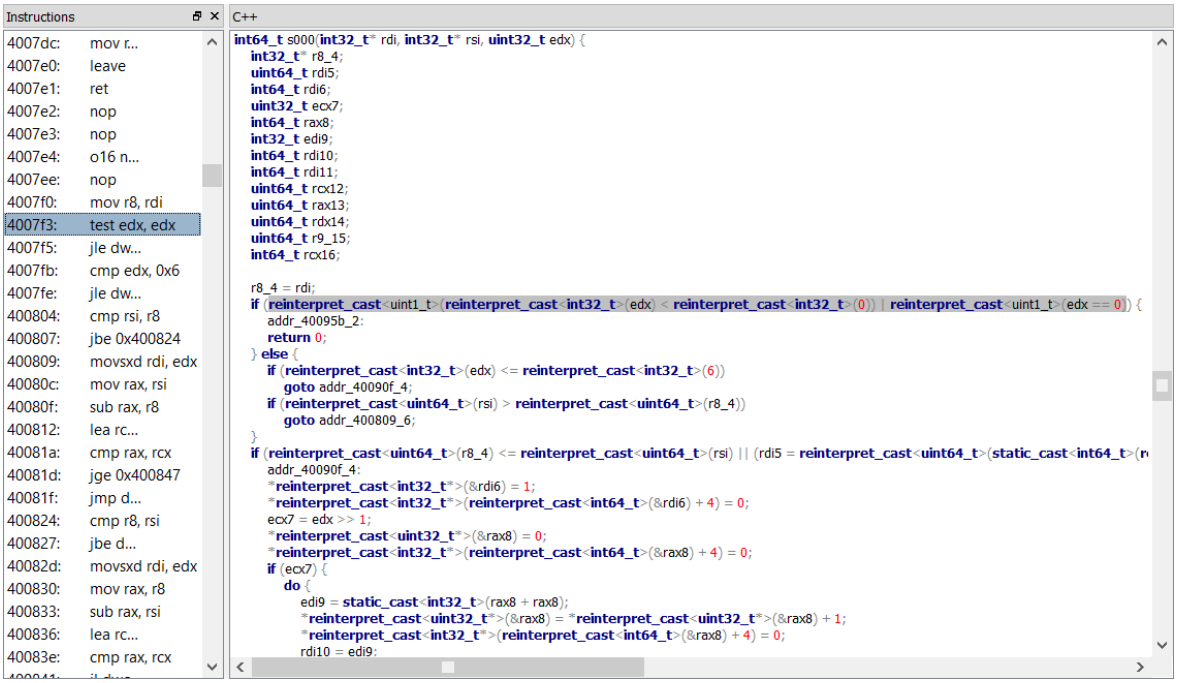


Figure 3.1: Illustration of Snowman's decompilation user interface [TCD09].

Our test input selection process for a compiled program works as follows. We start by selecting test inputs that cover the program on source code level and analyze the achieved machine code coverage with those inputs using GNATcoverage. Using the GNATcoverage instruction annotations, we investigate what branch instructions are not yet fully covered by our test inputs. By looking up the instruction in Snowman's graphical user interface, we can analyze the corresponding decompiled code to find what test input is needed to trigger the conditional branch. By repeating this process, we can systematically analyze each uncovered branch instruction and create the set of test inputs needed to fully cover the machine code of the test program.

However, this way full machine code coverage is only achieved for the machine code generated by one specific compiler. As each compiler can implement optimizations differently, full machine code coverage for a test program 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. 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 investigating optimizations performed by the GCC, Clang and ICC compilers and extracting recognizable patterns from their behavior to select test inputs that cover these patterns. Hence, instead of selecting specific test inputs that trigger specific conditional branches introduced by a compiler, we select a range of test inputs that covers the identified patterns. GCC is part of the GNU collection of open source compilers, Clang is an open source compiler based on the LLVM

compiler framework [Cla] and ICC is a commercial compiler developed by Intel [ICC].

In addition to analyzing the machine code generated by these compilers, we also use the optimization reports that they provide. These reports provide information on what optimizations are applied to a loop, as well as additional information such as the loop unroll factor that is used. This way we analyze whether the optimization that is targeted by a test program is actually applied by a compiler. Upon that, the additional information provided by the compiler can support the identification of optimization patterns that we use to select robust test inputs. Optimization reporting by GCC, Clang and ICC is enabled using the `-fopt-info`, `-Rpass` and `-qopt-report` flags respectively.

## 3.6 Comparison checking

To test loop optimizations applied by a compiler for correctness, we analyze the result of executing an optimized loop. Based on the methodology applied by Yang et al. [YCER11], after executing the loop we calculate a checksum of all variables that are used inside the loop body. For example, for the loop shown in Listing 3.2, this would be the sum of `X[0..len]` and `Y[0..len]`. Before executing the loop we initialize `X[0..len]` and `Y[0..len]` with unique values for each index, such that different checksums are obtained for every value of `len`. An important advantage of using a checksum is simplicity, as this way we only need to analyze a single value.

To validate the checksum, we use comparison checking, which is based on the work of Jaramillo et al. [JGS02]. This means a test program is compiled without optimizations enabled, as well as at different optimization levels, and the checksums obtained from running these executables are compared. If the checksum differs between two executables, the behavior of the executed code is changed by an optimization and thus the optimization is incorrect. An advantage of this methodology is again simplicity, as this way we do not need to determine predefined results to analyze whether the result of executing the code is correct, but we only need to compare the output of running multiple executables. As determining the result of executing a loop beforehand can be complex, this saves a lot of time when adding a new loop to the test suite.

Using comparison checking, the behavior of the code is not explicitly tested to be correct, but instead it is tested to remain consistent between unoptimized compilation and compilation at different optimization levels. This methodology thus assumes that the compiler produces correct code when compiling a program without applying optimizations. As our test suite is designed to be used in addition to a compiler test suite like SuperTest, such a test suite can be used to verify this assumption before executing our test suite.

# Chapter 4

# Test Suite Implementation

In this chapter, we discuss the implementation of our loop optimization test suite. We mainly focus on loop optimizations that introduce new conditional branches to the compiled code, as for these optimizations it is impossible to select test cases that fully cover the optimized machine code based on the source code of the optimized loop.

For each loop optimization, we first provide a short introduction on how the optimization works, followed by a description of the identified optimization patterns and the test inputs needed to cover those patterns. We extract these patterns by analyzing how the optimizations work in general, as well as by analyzing how the GCC, Clang and ICC compilers apply them. We perform all analysis on test programs compiled for our Intel Skylake X86-64 hardware setup.

## 4.1 Loop inversion

Loop inversion is a relatively simple loop optimization that transforms a `for` or `while` loop into a `do-while` loop that is wrapped in an `if` statement [Jub14]. This `if` statement provides a shortcut to skip the loop when its condition is not met, as the condition is the inverse of the loop condition. Loop inversion is applied to reduce the number of jump instructions needed to execute a loop, as the loop condition is moved from above the loop body to below it. Listing 4.1 shows an example of this optimization. In this example, `body(i)` is a placeholder for the actual loop body that depends on loop variable `i`.

```
1  // Original loop
2  void loop_inversion(int n) {
3      for(int i = 0; i < n; i++) {
4          body(i);
5      }
6  }
7
8  // After loop inversion
9  void loop_inversion(int n) {
10     if(n > 0) {
11         do {
12             body(i);
13             i++;
14         } while(i < n);
15     }
16 }
```

Listing 4.1: Loop inversion example [Jub14].

While for `n = 1` full coverage on the original loop is achieved, for the inverted loop two inputs are needed. First of all, `n <= 0` is needed to trigger the shortcut branch. If this test case is missing, the shortcut branch is only covered on the true condition, which means full branch coverage is not

achieved. Secondly, as the loop condition is evaluated below the loop body, the loop body needs to be executed at least twice to fully cover the loop condition branch. Therefore, in this case `n > 1` is needed to fully cover the loop condition branch.

These test inputs show a clear pattern based on the original loop. An `n` that does not meet the loop condition is needed to skip the execution of the loop, while an `n` for which the loop body is executed at least twice is needed to cover the loop branch condition both ways.

Loop inversion is a very common optimization that is applied to almost all loops that we analyze during this project. Therefore, all loops in our test suite need to be tested with inputs that take loop inversion into account, such that this optimization is fully covered at machine code level.

## 4.2 Loop unrolling

As explained in the introduction, a loop can be optimized by unrolling it to reduce the number of jumps needed to execute the loop. When unrolling, the loop body is replicated a certain amount of times and the loop increments by the same number [BGS94]. This number is called the unroll factor (UF).

When the number of loop iterations depends on a variable and thus is not known at compile time, the compiler needs to take into account that it is possible that the number of iterations is not a multiple of the loop unroll factor. In that case, the compiler adds a remainder loop below the unrolled loop, which is used to handle the remaining loop iterations. Listing 4.2 shows an example of loop unrolling by factor 4 and the corresponding remainder handling branch. Again, `body(i)` is a placeholder for the actual loop body that depends on loop variable `i`.

```
1  // Original loop
2  void loop_unrolling(int n) {
3      for(int i = 0; i < n; i++) {
4          body(i);
5      }
6  }
7
8  // After unrolling
9  void loop_unrolling(int n) {
10     // Unrolled loop
11     for(int i = 0; i < n - 3; i += 4) {
12         body(i);
13         body(i + 1);
14         body(i + 2);
15         body(i + 3);
16     }
17
18     // Remainder
19     for(int j = n - (n & 3); j < n; j++) {
20         body(j);
21     }
22 }
```

Listing 4.2: Example of loop unrolling by factor 4 [Sar15, BGS94].

The test inputs we need to fully cover unrolled loops at machine code level depend on the unroll factor used by the compiler. Compilers may select this factor dynamically based on what the compiler's analysis determines is the most efficient. For the example of loop unrolling by factor 4, it is sufficient to test with `n = 5` to cover the loop branch both ways, as well as to cover the remainder loop branch both ways. However, this test input would not fully cover a loop that is unrolled by factor 5, as in that case the remainder branch would not be executed.

Furthermore, GCC implements remainder handling a bit differently. First of all, it performs remainder handling before executing the unrolled loop instead of afterwards. Upon that, before executing the unrolled loop it checks if the remainder is equal to `n`, which means that the entire unrolled loop

can be skipped. Doing remainder handling before or after executing the unrolled loop is just a design decision by the compiler developers, as it does not make a significant difference in terms of performance. Most importantly, instead of adding a loop that handles all remaining iterations, GCC adds nested conditional branches that explicitly check for all possible remainder values. To cover all these branches, test inputs that trigger all possible remainder values are needed while a remainder handling loop only needs one test input to be fully covered. Listing 4.3 illustrates the way GCC performs remainder handling.

To robustly cover the machine code of an unrolled loop, we use test inputs `n = [0..2UF]`, where `UF` is the maximum loop unroll factor that the tested compiler will apply. The maximum loop unroll factor can usually be retrieved from the compiler's documentation, so this value is configured before executing our test suite. For `n = 2UF` full loop branch coverage is assured, as for this value the loop body is executed at least twice for every unroll factor. Together with `n = 0`, this way we take into account that loop inversion could be applied to the unrolled loop. Using all values in between, we ensure that every possible remainder value is triggered by our test cases.

```
1  void loop_unrolling(int n) {
2      // Remainder
3      int remainder = n & 3;
4      if(remainder != 0) {
5          body(0);
6          if (remainder != 1) {
7              body(1);
8              if (remainder != 2) {
9                  body(2);
10             }
11         }
12
13         if(remainder == n) {
14             return;
15         }
16     }
17
18     // Unrolled loop
19     ...
20 }
```

Listing 4.3: C representation of GCC's remainder handling branches when unrolling the loop of Listing 4.2 by factor 4.

## 4.3 Vectorization

Modern CPU architectures support vector instructions that make it possible to apply the same operation to multiple values at once, instead of doing this one value at a time [Eme16]. This optimization leads to significant speedup, as the same operation can be applied in less loop iterations. To do so, special vector registers are used. Intel Xeon Phi processors have vector registers that are up to 512 bits long [Inta], but more common processors like the Intel Ivy Bridge processors have vector registers that are 128 and 256 bits long [Intb]. For the latter register size, this means that for example on 32-bit integer pointers, $256/32 = 8$ indices can be processed using a single vectorization instruction. The number of indices that can be processed using a single vectorization instruction is called the vectorization factor (VF) [HRCK15].

Listing 4.4 illustrates an example of loop vectorization. As the loop iteration count is not known at compile time, a conditional branch is added around the vectorized loop, as the loop can only be vectorized if the iteration count is at least the same as the vectorization factor. If it is not, the loop is executed sequentially. Furthermore, if the iteration count is not a multiple of the vectorization factor, again a remainder handling loop is added below the vectorized loop.

```
// Original loop
void vectorization(int *X, int *Y, int n) {
    for(int i = 0; i < n; i++) {
        X[i] = Y[i] + 1;
    }
}

// After vectorization (pseudo code)
void vectorization(int *X, int *Y, int n) {
    if(n >= VF) {
        // Vectorized loop
        for(int i = 0; i < n; i += VF) {
            X[i..i+VF] = Y[i..i+VF] + 1;
        }

        // Remainder
        for(int j = n - (n & VF); j < n; j++) {
            X[j] = Y[j] + 1;
        }
    } else {
        // Original sequential loop
        ...
    }
}
```

Listing 4.4: Example of loop vectorization and remainder handling. If the number of loop iterations if below the vectorization factor (VF), the original sequential loop is executed instead.

This simplified example already shows that multiple values for `n` are needed to fully cover the generated machine code. An `n` below the vectorization factor is needed to cover the sequential loop, while an `n` above the vectorization factor is needed to fully cover the vectorized loop. Like when applying loop unrolling, it is possible that a compiler again adds nested conditional branches for remainder handling. Therefore, also multiple values for `n` are needed to take this into account.

Instead of using the vectorization factor as a threshold for vectorization, most compilers use a *profitability threshold*. This is the minimum number of loop iterations for which it is assumed to be profitable to execute the vectorized loop instead of the sequential one. While Clang and ICC always use a threshold of 1 or 2 times the vectorization factor, GCC dynamically calculates a profitability threshold by analyzing the loop body. This is important to take into account when selecting robust test cases to cover vectorized loops.

Both the vectorization factor and the profitability threshold depend on the size of the data type of the pointer that is vectorized, as well as the size of the vector registers of the processor that the program is being compiled for. The 8-bit `char` data type is the smallest possible data type in C/C++ and thus has the largest possible vectorization factor of all data types. The largest possible vector registers on our hardware setup are 256 bits long, so $256/8 = 32$ is the largest possible vectorization factor. As Clang and ICC always use a threshold of 1 or 2 times the vectorization factor, we need an `n = 64` to robustly cover the vectorized loop body. For this value, the loop body is executed at least twice, which means possible loop inversion is also taken into account. This value is also robust for dynamically calculated thresholds by GCC, as we did not encounter dynamically calculated thresholds that were larger than 20.

Next, we need to robustly cover the remainder handling code, taking into account that a compiler might again introduce nested conditional branches that test for every possible remainder value instead of a single remainder handling loop. Using all values between two multiples of 32, every possible remainder value is triggered, so robust test coverage is achieved. Hence, we use `n = [65..95]` as test inputs for the remainder handling code.

In total, we use `n = [0..95]` as our robust set of test inputs. In addition to the values discussed before, all values below 64 ensure that the original sequential loop that is executed when the vectorization threshold is not passed is also robustly covered. When testing a compiler on hardware with larger

vector registers, we adjust this range of values accordingly. Our range of test inputs then becomes `n = [0..3VF-1]`, where `VF` is the largest possible vectorization factor of the target hardware.

### 4.3.1 Pointer aliasing

In addition to test cases based on the loop iteration count, for vectorized loops we also need test cases based on the pointer parameters that are accessed in the loop body. First of all, conditional branches that check for pointer aliasing are added to the generated machine code. When two pointers are aliases, they point to overlapping regions in memory, as illustrated in Figure 4.1. Passing overlapping pointers to the loop shown in Listing 4.4 would lead to a dependency between consecutive loop iterations. The first loop iteration writes to `X[i]`, which means the value of `Y[i+1]` changes as well. In the next loop iteration, `Y[i+1]` is accessed, which was thus updated by the previous loop iteration. If the loop would be vectorized, multiple loop iterations would be executed at the same time so the old value of `Y[i+1]` would be used, leading to incorrect results. To prevent this, the conditional branch that checks for pointer aliasing ensures that in this case the sequential version of the loop is executed instead.

Pointer aliasing only prevents vectorization when the overlapping distance is smaller than the vectorization factor, as otherwise there is no dependency between any of the loop iterations that are executed at the same time. To cover conditional branches that check for pointer aliasing, we use the three test cases shown in Listing 4.5. In the first test case `X` overlaps `Y`, in the second test case `Y` overlaps `X` and in the third test case `X` and `Y` point to the exact same memory. This way we robustly cover all possible aliasing variations. Even though not all of these aliasing variations would lead to actual dependencies between loop iterations, the compilers we investigated execute the sequential version of the loop for all of them. We use pointers that overlap by a distance of 1 index, such that the overlapping distance is always smaller than the vectorization factor. The value for `n` used with these test cases should be above the vectorization threshold, as aliasing checks are only performed when a loop is potentially being vectorized.

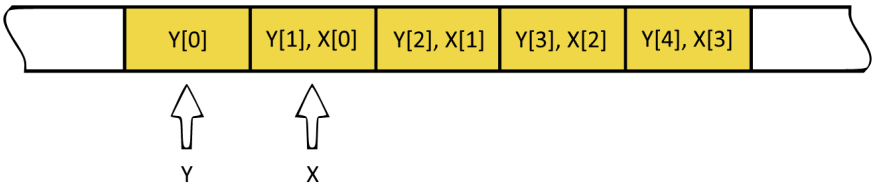


Figure 4.1: Illustration of pointers `X` and `Y` pointing to overlapping memory regions [Ins].

```
1 void vectorization(int *X, int *Y, int n);
2 vectorization(X, &X[1], n); // first parameter overlaps second parameter
3 vectorization(&X[1], X, n); // second parameter overlaps first parameter
4 vectorization(X, X, n);     // direct aliasing
```

Listing 4.5: Test cases to robustly cover conditional branches that check for pointer aliasing.

### 4.3.2 Memory alignment

Other conditional branches based on the pointer parameters are related to the memory alignment of the pointers. Vectorization is the most efficient when the starting index of the pointer that is accessed in the vectorized loop points to memory that is aligned by the same number of bytes as the size of the vector registers. That way, aligned load and store instructions can be used, which are more efficient than their unaligned counterparts [Kri15, Cen16]. Hence, on processors with 256-bit vector registers 32-byte alignment is the most efficient while on processors with 512-bit vector registers 64-byte alignment is the most efficient.

To benefit from the more efficient aligned load and store instructions, compilers might enforce that a vectorized loop starts at a pointer index that is aligned efficiently. This is done by loop peeling,

which means that the first iterations of the loop are executed sequentially until a pointer index is reached that has the desired memory alignment [Eme16]. Listing 4.6 illustrates an example of loop peeling until a 32-byte aligned address is reached.

```
1  // Loop sequentially until a 32-byte aligned address is reached
2  int peeled = 0;
3  while(alignment_of(&X[peeled]) != 32) {
4      X[peeled] = Y[peeled] + 1;
5      peeled++;
6  }
7
8  // Vectorized loop with efficient memory alignment
9  for(int i = peeled; i < n; i += VF) {
10     X[i..i+VF] = Y[i..i+VF] + 1;
11 }
```

Listing 4.6: Pseudocode example of loop peeling applied to the vectorized loop of Listing 4.4 to improve memory alignment.

While Clang and GCC use a loop peeling branch that only needs one test case to be fully covered, GCC again uses several nested conditional branches to explicitly check for every possible memory alignment. For example, when assuring an integer pointer to be aligned by 16 bytes, GCC adds branches to check whether the pointer is aligned by 16, 12, 8 or 4 bytes. This means that for robustness, on hardware with vector registers that are up to 256 bits long, we need to test a vectorized loop with pointers with all possible alignments up to 32 bytes. Likewise, on hardware with vector registers that are up to 512 bits long, we need to test this with pointers with all possible alignments up to 64 bytes.

To do this on hardware with 256-bit vector registers, we allocate a 32-byte aligned pointer and pass different pointer indices as parameters to the tested loop vectorization function. For example, as integers have a size of 4 bytes, the address of the second index of a 32-byte aligned integer pointer is only aligned by 4 bytes and will thus trigger the loop peeling branch. To cover all possible cases up to 32-byte alignment, for integer pointer `X` we need to pass `&X[0]` until `&X[7]` to the tested function, as `&X[8]` will again be 32-byte aligned. Similarly, as the size of the `char` data type is 1 byte, for `char` pointer `X` we need to pass `&X[0]` until `&X[31]`. Again, as `char` is the smallest possible data type, this is a robust selection of test cases that covers all peeling branches for all possible data types. Similarly, to robustly cover all peeling branches on hardware with 512-bit vector registers, we allocate a 64-byte aligned pointer and pass `&X[0]` until `&X[63]` to the tested function.

Listing 4.7 demonstrates this approach. The standard C `malloc` function to allocate memory does not guarantee the memory to be aligned by 32 bytes, so we need an alternative function that does guarantee alignment. On UNIX systems the `memalign` function can be used to do this, but this does for example not work on Windows systems. Alternatively, since the C11 standard the `aligned_alloc` function is available in the standard library, but we also want to be able to test compilers that implement older standards. Therefore, we use a custom implementation of this function that works on any system and with every C/C++ standard [mal13].

```
1  // Allocate a 32-byte aligned pointer
2  TYPE *X = aligned_malloc(32, (n + 32) * sizeof(TYPE));
3  TYPE *Y = aligned_malloc(32, (n + 32) * sizeof(TYPE));
4
5  for(int i = 0; i < 32; i++) {
6      vectorization(&X[i], Y, n); // variate the alignment of pointer X
7      vectorization(X, &Y[i], n); // variate the alignment of pointer Y
8  }
```

Listing 4.7: Example of test inputs to trigger memory alignment optimization branches for the vectorized loop of Listing 4.4.

While for the loop shown in Listing 4.4 GCC and Clang apply loop peeling to enhance the alignment of pointer `X`, ICC applies loop peeling to enhance the alignment of pointer `Y`. Hence, for robustness we

include memory alignment test cases for both pointers that are accessed inside the loop body. Again, the value for `n` used with these test cases should be above the vectorization threshold to trigger the loop peeling branches.

### 4.3.3 Multi-dimensional pointer access vectorization

When a loop accesses a multi-dimensional pointer instead of a one-dimensional pointer, similar test cases are needed to cover all vectorization-related branches. Again, test cases for aliasing and overlapping pointers are needed, as well as test cases with pointers with different memory alignment. These cases need to be applied to the deepest pointer dimension, as those pointer indices are next to each other in memory and operations applied to these indices can thus be vectorized. For two-dimensional pointer access, we change the alignment of the deepest pointer dimension as demonstrated in Listing 4.8. After testing, we move the alignment back to as it was originally to be able to properly free the pointer from memory. Similarly, pointer aliasing is triggered as demonstrated in Listing 4.9.

```
1 X[0] = &X[0][1];     // Change alignment of deepest dimension
2
3 ...                  // Test vectorized loop
4
5 X[0] = &X[0][-1];    // Move pointer back to as it was originally
```

Listing 4.8: Example of changing the alignment of the deepest dimension of a two-dimensional pointer.

```
1 TYPE *temp = Y[0];   // Save original pointer address
2
3 Y[0] = &X[0][1];     // Let Y[0] overlap X[0]
4
5 ...                  // Test vectorized loop
6
7 Y[0] = temp;         // Restore original pointer
```

Listing 4.9: Example of overlapping two-dimensional pointers.

## 4.4 Strip mining

Strip mining is a loop optimization that divides a single loop into two nested loops [PW86, Vla15]. The outer loop iterates in strips over the original loop iteration space while the inner loop operates on the iterations inside the strip. This optimization is often applied in combination with vectorization or loop unrolling, where the strip size is a multiple of the vectorization factor and the inner loop is vectorized or unrolled. An example of strip mining is shown in Listing 4.10. If the iteration count `n` is not a multiple of the strip size, again a remainder handling loop is used to execute the remaining loop iterations. Furthermore, if the loop iteration count is below the strip size, the original loop is executed instead of the strip mined one.

We only found examples of strip mining applied by the ICC compiler on loops that iterate over integer pointers. Therefore, we were not able to identify possible patterns based on different compilers or based on the data type of the pointers accessed in the loop body. In addition to strip mining, ICC indeed applies vectorization to the inner loop. From this, the only clear pattern we found is that the strip size is a multiple of the vectorization factor. For all strip mined loops in our test suite a strip size of 64 is used, so a loop iteration count of at least 64 is needed to achieve full loop branch coverage. To also cover all remainder branches and to take into account possible loop inversion, we test strip mined loops with `n = [0..128]`. Additionally, the test cases for pointer aliasing and memory alignment discussed in Section 4.3 are needed to cover all branches introduced by the vectorization of the inner loop.

```
// Original loop
void strip_mining(int *X, int *Y, int n) {
    for(int i = 0; i < n; i++) {
        X[i] = Y[i] + 1;
    }
}

// After strip mining
void strip_mining(int *X, int *Y, int n) {
    if(n >= STRIP_SIZE) {
        // Strip mined loop
        for(int j = 0; j < n; j += STRIP_SIZE) {
            for(int i = j; i < j + STRIP_SIZE; i++) {
                X[i] = Y[i] + 1;
            }
        }

        // Remainder
        ...
    } else {
        // Original sequential loop
        ...
    }
}
```

Listing 4.10: Example of a loop optimized by strip mining [Vla15].

## 4.5 Further optimizations

Loop unrolling, vectorization and strip mining are the most significant loop optimizations in terms of newly introduced conditional branches to the optimized machine code. Many other loop optimizations that are covered by our test suite, are optimizations that are designed to make vectorization possible. This means that a loop can not directly be vectorized, but by applying another optimization first, vectorization can be applied afterwards. Loops that trigger these optimizations therefore do not need new optimization-specific test cases, but instead need the selection of vectorization test cases discussed in Section 4.3. In this section, we discuss such loop optimizations that are targeted by our test suite.

```
// Before loop unswitching
for (int i = 0; i < n; i++) {
    if(n > 100) {
        X[i] = Y[i] + 1;
    } else {
        Y[i] = X[i] + 1;
    }
}

// After loop unswitching
if(n > 100) {
    for (int i = 0; i < n; i++) {
        X[i] = Y[i] + 1;
    }
} else {
    for (int i = 0; i < n; i++) {
        Y[i] = X[i] + 1;
    }
}
```

Listing 4.11: Loop unswitching example [BGS94].

When inside the body of a loop an if-else condition is present, the loop cannot be vectorized as it is possible that one iteration the true branch should be executed while in another iteration the false branch should be executed. However, if the condition does not depend on a loop variable, it is loop-invariant and can be optimized by loop unswitching. The if-else condition is moved outside of the loop and the loop is replicated inside each branch [BGS94]. After applying this optimization, the individual loops do not have an if-else condition in the loop body and can thus be vectorized. An example of loop unswitching is shown in Listing 4.11.

Another example is loop distribution. Loop distribution breaks a single loop into multiple loops, after which some of the distributed loops can be vectorized [BGS94]. For example, the loop in Listing 4.12 cannot be vectorized, as the first statement in the loop body depends on an earlier loop iteration. The second statement, however, does not have such a dependency and is therefore suitable for vectorization. By splitting the loop into two separate loops, the second loop can be vectorized while the first loop is still executed sequentially.

```
1  // Before loop distribution
2  for (int i = 1; i < n; i++) {
3      X[i] = X[i-1] + 1;
4      Y[i] = Z[i];
5  }
6
7  // After loop distribution
8  for (int i = 1; i < n; i++) {
9      X[i] = X[i-1] + 1;
10 }
11
12 for (int i = 1; i < n; i++) {
13     Y[i] = Z[i];
14 }
```

Listing 4.12: Loop distribution example [Eme16].

Combined with loop distribution, statement reordering is an optimization that can allow for vectorization by changing the order of statements in the loop body [Asl]. This principle is illustrated by the loop in Listing 4.13. In the original loop body, the first statement accesses `Y[i]`, which depends on the second statement that writes to `Y[i+1]`. Vectorization of this loop would lead to incorrect results, as this dependency would not be taken into account. However, by changing the order of the two statements and distributing the resulting loop, this dependency is taken into account and vectorization of the distributed loops is possible.

```
1  // Before statement reordering
2  for (int i = 0; i < n-1; i++) {
3      X[i] = Y[i] + 1;
4      Y[i+1] = Z[i];
5  }
6
7  // After statement reordering
8  for (int i = 0; i < n-1; i++) {
9      Y[i+1] = Z[i];
10     X[i] = Y[i] + 1;
11 }
```

Listing 4.13: Statement reordering example [Asl].

The opposite of loop distribution is loop fusion [BGS94]. Here, two separate loops that have the same iteration space are fused into a single loop. While this optimization does not allow for vectorization of a loop that could not be vectorized before, this optimization reduces both the number of jump instructions needed to execute the program and the program size. Listing 4.14 shows an example of loop fusion.

```
// Before loop fusion
for (int i = 0; i < n; i++) {
    X[i] = Y[i] + 1;
}
for (int i = 0; i < n; i++) {
    Z[i] = Y[1] + 2;
}

// After loop fusion
for (int i = 0; i < n; i++) {
    X[i] = Y[i] + 1;
    Z[i] = Y[i] + 2;
}
```

Listing 4.14: Loop fusion example.

Furthermore, loop interchange inverts the order of nested loops [Eme16]. This optimization leads to a more efficient memory access pattern, improving locality of reference. Upon that, this improved order of memory access could again allow for vectorization. Listing 4.15 shows an example of this optimization. Before loop interchange is applied, the inner loop iterates over the outer pointer index. Here, consecutive loop iterations do not access consecutive memory addresses, which means that the inner loop cannot be vectorized. However, by interchanging the loops, consecutive loop iterations do access consecutive memory addresses, so the inner loop can be vectorized.

```
// Before loop interchange
for (int j = 0; j < m; j++) {
    for (int i = 0; i < n; i++) {
        X[i][j] = Y[i][j] + 1;
    }
}

// After loop interchange
for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
        X[i][j] = Y[i][j] + 1;
    }
}
```

Listing 4.15: Loop interchange example [Eme16].

Finally, loop rerolling is the opposite of loop unrolling that could allow for vectorization by simplifying a loop [MGG+11]. Listing 4.16 shows an example of a manually unrolled loop that can be rerolled. The original loop cannot directly be vectorized, as the loop body accesses the same pointer multiple times and the loop does not increment by 1. However, when applying loop rerolling the loop body is shrunk to a single statement and the loop increments by 1 instead of 3. The resulting simplified loop is again suitable for vectorization.

```
// Manually unrolled loop
for (int i = 0; i < n; i += 3) {
    X[i] = Y[i] + 1;
    X[i + 1] = Y[i + 1] + 1;
    X[i + 2] = Y[i + 2] + 1;
}

// After loop rerolling
for (int i = 0; i < n; i++) {
    X[i] = Y[i] + 1;
}
```

Listing 4.16: Loop rerolling example [MGG+11].

## 4.6 Compile time versus runtime analysis

As discussed in Section 3.2, we parameterize all loop variables such that the compiler knows as little as possible about the variables that are used by a loop. This way, the compiler for example needs to take into account that the loop iteration count is not a multiple of the loop unroll factor, or that the pointers passed as parameters could be aliases. If instead the loop iteration count is a constant, so the iteration count is already known at compile time, the loop can be optimized without taking into account other possibilities. Similarly, a compiler could be informed that pointer aliasing will not occur by the `restrict` keyword added in the C99 standard. By adding this keyword to the pointer parameters of a function, the programmer promises the compiler to never pass aliasing pointers to the function and the compiler thus does not need to add runtime aliasing checks to the code.

Some loop optimizations are only applied when the compiler has such knowledge on the properties of a loop at compile time. An explanation for this is that compilers could consider the profitability of an optimization too low when a remainder loop is needed after loop unrolling, or when runtime aliasing checks are needed before vectorizing a loop. For example, ICC sometimes reports "vector dependence prevents vectorization" and thus does not vectorize a loop. However, after adding the `restrict` keyword to the pointer parameters, the same loop is vectorized by the compiler.

Because of this, the loops in our test suite could be varied by including both a version with and without the `restrict` keyword in the parameters. Similarly, loops can be varied by using a fixed iteration count instead of a parameterized iteration count. This way, our test suite also covers compilers that require such information at compile time in order to apply certain optimizations. As the `restrict` keyword was added in the C99 standard, tests utilizing this keyword are only suitable for compiler that implement C99 or a newer standard.

## 4.7 Unsatisfiable branches

While our goal is to achieve full test coverage at machine code level on the test programs in our test suite, we found that this is sometimes impossible to achieve. Compiler optimizations might introduce unsatisfiable branches: nested branch conditions that can never be covered on both the true and the false condition. In that case, the machine code thus still shows room for optimization, as the conditional jump could be replaced with an unconditional jump.

An example of an unsatisfiable branch is the following. For the example loop in Chapter 2, at optimization level -O2 the Clang 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`. Hence, this branch condition can never be covered on the true condition.

Unsatisfiable branches like these mean that the test programs that are designed for this project not always achieve full machine code coverage. If it can be proven that the uncovered branches are impossible to satisfy, this is not a problem, as the untested code can never be executed. When testing a compiler this is a slowing down factor as it requires an extra analysis step to prove that remaining uncovered branches of a test program are in fact unsatisfiable. However, also unsatisfiable branches show recognizable patterns which could simplify this process. For example, Clang always introduces the same unsatisfiable branches when a loop is vectorized, so these unsatisfiable branches can be recognized based on the optimization that is being performed.

A possible explanation for occurrence of unsatisfiable branches is that the analysis of branch constrains could be expensive. To analyze whether a nested conditional branch is unsatisfiable, constraint solving should be performed on the branch condition. While this could be done using SMT solving [DMB08], such algorithms could significantly slow down the compilation process of a program. As the elimination of an unsatisfiable conditional jump would not lead to significant performance improvements of the compiled program, the slowdown in the compilation process outweighs the potential performance improvements of the compiled program.

For the compilation of all test programs in our test suite by GCC, Clang and ICC, we are able to either achieve full machine code coverage, or to demonstrate that any missed coverage is caused by unsatisfiable branches.

# Chapter 5

# Test Suite Evaluation

In this chapter, we evaluate the results of implementing our test suite. First, we provide an overview of the optimizations that are targeted by our test suite and the corresponding number of test programs that are designed to do so. Secondly, we provide an overview of the different types of optimization patterns we use for selecting robust test inputs. Finally, we discuss the performance of our test suite.

## 5.1 Overview of test programs

Our current test suite consists of 40 loops targeting the 10 different loop optimizations discussed in the previous chapter. Our test suite thus contains multiple programs targeting the same optimization. Table 5.1 shows an overview of the optimizations covered by our test suite and the number of test programs that target these optimizations.

It is important to note that while a test program is not designed to trigger a specific optimization, compilers might still apply that optimization to the test program. For example, loop unrolling is often applied to the remainder of a vectorized loop and loop inversion is applied to every loop in our test suite. We take this into account by using test inputs that take this into account for every program in our test suite. However, in Table 5.1 we only count the test programs that are specifically designed to trigger an optimization.

| **Loop optimization** | **Number of test programs** |
|---|---:|
| Loop unrolling | 3 |
| Vectorization | 20 |
| Strip mining | 4 |
| Loop unswitching | 2 |
| Loop distribution | 3 |
| Statement reordering | 2 |
| Loop fusion | 2 |
| Loop interchange | 2 |
| Loop rerolling | 2 |

Table 5.1: Overview of the number of test programs that are designed to trigger a specific loop optimization in our test suite.

The table clearly shows that most of our test programs focus on vectorization, while only a small number of programs focus on other loop optimizations. This is because vectorization can be applied in varying scenarios. For example, vectorization can be applied to one dimensional pointers, but also to multi dimensional pointers. Similarly, vectorization can be applied to loops accessing only a single pointer, but also to loops accessing a large number of pointers. These factors influence the way the optimization is applied, for example because pointer aliasing checks need to be performed on a larger number of pointers, adding more conditional branches to the optimized machine code.

Most other optimizations cannot be varied like this. For example, loop unswitching can only be triggered by a loop invariant `if` statement in the loop body, while pointer dimensions or the number of variables accessed in the loop body do not influence the way the optimization is applied. Therefore, our test suite contains a wide range of loops targeting different vectorization scenarios, while other optimizations are only targeted by a small number of loops. This reflects the ratio of the benchmarks in the TSVC benchmark suite, which most of our test programs are based on.

## 5.2 Types of optimization patterns

As discussed before, each compiler can implement optimizations differently. Hence, full machine code coverage for a test program compiled by one compiler does not guarantee full machine code coverage when using another compiler. Therefore, our goal is to select robust test inputs that cover the optimizations of compilers in general as widely as possible. From the implementation of our test suite discussed in Chapter 4, we are able to extract three different types of optimization patterns that can be used to select a robust range of test inputs. In this section, we provide an overview of these patterns.

The first optimization pattern is purely based on the optimization itself and independent of the compiler that is being used or the target hardware. An example is loop inversion, where a `for` or `while` loop is transformed into a `do-while` loop that is wrapped in an `if` statement. Here the `if` condition is always the inverse of the loop condition and the loop condition is moved from above the loop body to below it. To cover this, we thus need a test input that does not meet the loop condition, as well as a test input that ensures that the loop body is executed at least twice, such that the loop condition is fully covered. These test inputs are purely based on the way the optimization works and not on any compiler settings or target hardware details.

The second optimization pattern is based on the configuration of the compiler that is being used. We use this to robustly cover loop unrolling. Here we select the range of test inputs to robustly cover the remainder handling branches based on the maximum loop unroll factor that a compiler will apply. For example, if 8 is the maximum unroll factor that a compiler will apply, we do not need take a remainder value of 9 into account. However, if the maximum unroll factor is 10 instead, we do need to take a possible remainder value of 9 into account. Knowing the maximum unroll factor that the compiler will apply, we can set an upper bound for our range of test inputs for which we are sure that every possible remainder value is covered. The maximum unroll factor can be retrieved from the configuration of the compiler that is being tested.

The final optimization pattern is based on the target hardware that a test program is compiled for. We use this to robustly cover vectorization. As discussed in Section 4.3, the vectorization factor that a compiler can apply depends on the size of the vector registers of the target hardware. For example, if the largest vector registers on the target hardware are 256 bits long, 32 is the largest possible vectorization factor. Hence, on this hardware we do not need to take a vectorization factor of 64 into account. On hardware with vector registers that are 512 bits long, however, we do need to take this vectorization factor into account to robustly cover the optimized machine code. Based on the target hardware, we can again configure the largest vector register size that needs to be taken into account to ensure robust coverage.

## 5.3 Performance analysis

C/C++ compiler test suites are designed to be used with any C/C++ compiler targeting any hardware setup, ranging from high-end to relatively slow processors. Therefore having a test suite that is not very computationally expensive could be beneficial, as that means it can be executed in a relatively short amount of time on slower hardware as well. On modern hardware, all tests programs terminate within a few milliseconds, but timing results for one specific hardware configuration are hard to map onto other hardware configurations.

Instead, to assess the computational costs of our test suite, we analyze the number of CPU instructions executed for each test program. This way, results that can more easily be mapped onto different hardware configurations are obtained. As different instructions could take a different number of CPU

cycles and instructions could for example speed up because of caching, the number of instructions executed by a program are not an absolute performance measurement. However, the results can be used as a rough performance indication, as there is a significant difference between executing thousands, millions or billions of instructions per test program.

We analyze the performance of our test suite using Callgrind [Cal]. A program is executed through Callgrind's execution tracer, after which it outputs the number of instructions executed for each function of the program. In Table 5.2 we provide results for a selection of test loops. The results are split into three phases of our testing routine: initialization, loop execution and finalization. During initialization, pointers that will be passed to the tested loop are allocated and initialized with values. During finalization, the checksum used for comparison checking is calculated and all pointers are freed from memory again. The loop execution results for both optimized and unoptimized versions of the code are included. The number of instructions for unoptimized loops could be considered a worst-case result, as loop optimizations generally reduce the number of instructions needed to execute a loop. The analysis is done for programs compiled by the ICC compiler, as that is the only compiler that applies the strip mining optimization.

| **Test description** | **Test initialization** | **Loop body -O0** | **Loop body -O3** | **Test finalization** |
|---|---|---|---|---|
| Loop unrolling by factor 8 | 3,664 | 2,805 | 920 | 2,414 |
| Integer pointer vectorization | 256,320 | 102,166 | 17,486 | 172,940 |
| 2D integer pointer vectorization | 10,086,988 | 6,932,091 | 558,924 | 7,983,616 |
| Strip mining and vectorization | 267,256 | 134,411 | 48,913 | 180,356 |

Table 5.2: Number of instructions executed for a selection of loops from our test suite compiled with ICC 18.0.2.

As discussed before, loop unrolling, vectorization and strip mining are the most significant loop optimizations in terms of newly introduced branches to the machine code. Other optimizations that are triggered by our test suite are mostly designed to make vectorization of a loop possible. Hence, for such optimizations the vectorization results can be used as a performance indication.

Results show that for our loop unrolling test program, a very small number of instructions is executed compared to the other optimizations. This can be explained by the fact that for loop unrolling, we only need a relatively small number of test cases to achieve full machine code coverage. For two dimensional pointer vectorization, the number of instructions significantly increases compared to one dimensional pointer vectorization, which is caused by the increase in pointer size from `n` to $\texttt{n}^2$.

Furthermore, the results show that test initialization and finalization take in the largest part of the test program execution. This is caused by the fact that for every test case, we initialize all pointer values before executing the loop and calculate a checksum of all pointer values after executing the loop. As discussed in Section 3.2, these parts of the test program are not optimized by the compiler such that no optimization errors can be introduced to this part of the code. Consequently, the initialization and finalization loops are executed relatively inefficiently.

# Chapter 6

# Discussion

In this chapter, we address points of discussion that came up during the day-to-day work on our project, as well as points of discussion brought up during the presentation of our work at the 2018 *Seminar Series on Advanced Techniques & Tools for Software Evolution* (*SATToSE*) in Athens [vVBG18]. Furthermore, we point out the limitations of our test suite.

## 6.1 Machine code MC/DC coverage

As mentioned in the introduction, the ISO 26262 [ISO11] functional safety standard for automotive software requires safety-critical software to be tested with MC/DC coverage. Important criteria of MC/DC coverage are that every conditional branch is covered on both the true and false condition and that each variable that is part of the condition is shown to independently affect the outcome of the decision [HVCR01]. The example in Listing 6.1 illustrates this. Here, testing the function with `a = 1` and `b = 0` results in coverage of the conditional branch on the true condition while testing with `a = 0` and `b = 0` results in coverage on the false condition. However, the branch would also be covered on the true condition for `a = 0` and `b = 1`, which means that `b` can independently affect the outcome of the decision. Therefore, this test case is required to achieve MC/DC coverage on this conditional branch.

For creating confidence in the correctness of a compiler, MC/DC coverage is not required by the ISO 26262 functional safety standard. However, achieving MC/DC coverage would mean that the compiler optimizations are tested even more thoroughly. While for our tests we aim to fully cover all conditional branches at machine code level, this does not guarantee MC/DC coverage at machine code level. With the GNATcoverage tool branch coverage on both the true and false condition can be analyzed, but it is not possible to analyze the additional MC/DC requirement of demonstrating that all variables independently affect the outcome of the decision. To the best of our knowledge, no tools that are able to do this are publicly available at the time of writing this thesis.

```
1 void f(int a, int b) {
2     if(a || b) {
3         // statements
4     }
5 }
```

Listing 6.1: Simple code snippet to demonstrate MC/DC coverage.

## 6.2 Automatic test input generation

While we currently manually analyze machine code to find the test inputs needed to achieve full test coverage, automatic test input generation would take less manual labor. A popular technique for automatic test input generation is symbolic execution. Using this technique, program inputs are

replaced by symbolic values and when program execution branches based on a symbolic value, the system maintains a set of constraints which must hold on the execution of that path [CDE+08]. This way, to cover a conditional branch, a test case can be generated by solving the current path condition with concrete values.

We investigated if it is possible to use the symbolic execution tool KLEE [CDE+08] to automatically generate test inputs that cover a program on machine code level. A problem here is that symbolic execution is performed on source code level instead of machine code level. Therefore we investigated if it is possible to perform symbolic execution on the decompiled code obtained from Snowman, as that way also the branches that are introduced by compiler optimizations are covered. We found that Snowman's decompilation results cannot always be directly compiled again, as the decompiled code sometimes contains invalid typecasts. While this issue is known to the developer of Snowman, he argues that the tool is meant to produce understandable code for analysis rather than actually compilable code [Der17].

Even when the invalid typecasts are manually fixed and the decompiled code by Snowman can thus be compiled again, KLEE is not able to find test inputs that fully cover the program. Whereas test inputs for integer value constrains like loop iteration counts can be generated, test inputs for pointer overlap or memory alignment of a pointer are not found by the tool. From this, we conclude that manually analyzing decompiled code and extracting recognizable optimization patterns is the most powerful approach to find test inputs that achieve full machine code coverage on a program.

## 6.3 Limitations

Before a compiler applies an optimization to a piece of code, the compiler analyses the code to determine whether it is suitable to be optimized. As discussed before, a loop could for example not be suitable for vectorization due to data dependencies. While we test the applied optimization for correctness by fully covering the generated machine code, we do not fully test the analysis phase this way. Instead, a broad range of example programs that are suitable to be optimized would be needed to test the compiler's analysis with different scenarios. Also negative examples of test programs that are not suitable to be optimized would be needed, to ensure that such programs are not incorrectly being optimized. The TSVC benchmark suite does contain some examples of programs that are not suitable to be vectorized, but for other optimizations such examples are harder to find. Another difficulty here is that it is hard to determine when a collection of code examples is sufficient to fully test the compiler's analysis phase, as there could be edge cases that are still being missed. Therefore it is almost impossible to guarantee that after testing a compiler using a collection of code examples no incorrect analysis can be performed by the compiler. However, testing compiler optimizations with a large collection of test programs shows that the compiler correctly optimizes code in a broad range of scenarios and thus helps enhancing confidence in the correctness of the compiler.

# Chapter 7

# Future Work

The goal of our project is not to create a test suite that fully covers all compiler optimizations, but instead to develop a methodology to create tests that can be incorporated in such a test suite. In this chapter, we discuss possible directions for the further development of our test suite.

## 7.1 Expanding the test suite

As discussed in the Section 6.3, a broad range of example programs with varying properties are needed to test the compiler's optimization analysis functionality with different scenarios. In this section, we discuss sources that can be used to expand our test suite.

While the TSVC benchmark suite consists of 151 loops, in the scope of our project it is not achievable to adapt all of them for correctness testing. Hence, our test suite can be expanded by adapting the full set of TSVC benchmarks for correctness testing. Furthermore, a useful source for additional test programs could be the Loop Repository for the Evaluation of Compiler (LORE) [CGS+17]. LORE is an online database that consists of loop benchmarks and code snippets from real software projects that can be used for benchmarking compilers. Another useful source could be the GCC test suite [GTS], which consists of a large collection of tests categorized by the optimization for which they are designed to trigger. Those tests are again not designed to be covered at machine code level, which could thus be an improvement when incorporating them into our test suite. Also interesting is the fact that the GCC test suite contains test programs that are based on past bug reports [GBR], which means those programs triggered optimization bugs in the past. Such bugs could possibly be caused by edge cases that were not taken into account, which makes these programs interesting for testing other compilers as well. To cover newly added test programs to our test suite at machine code level, the test inputs discussed in Chapter 4 can be used.

Besides adding additional test programs, in Section 4.6 we discussed that existing test programs can be varied by, for example, using the `restrict` keyword to tell a compiler that no pointer aliasing will occur, or by using fixed instead of variable loop iterations counts. This way, optimizations are tested in different scenarios and optimizations that are only applied when the `restrict` keyword is present are covered as well. Newly added test programs to our test suite can be varied in this way as well.

## 7.2 Testing on different hardware architectures

For this project, we only focus on compiler optimizations and their patterns when compiling code for the X86-64 architecture. However, it is possible that on other hardware architectures different optimizations are applied, or the same optimizations are applied differently. For example, on the X86-64 architecture vectorization is the most efficient when it is applied to 32-byte or 64-byte aligned memory while this number might be different on other hardware architectures. Hence, it could be interesting to also investigate compiler optimizations that target other hardware architectures. While this could be done by simply using different hardware configurations, the GNATcoverage tool we use also sup-

ports hardware emulation. That way, code could be compiled for a certain hardware architecture and executed through the GNATcoverage emulator, after which the corresponding machine code coverage results can directly be analyzed. However, to do this GNATcoverage relies on the GNATemulator tool [Adab], which is only commercially available. Therefore, we were not able to experiment with this tool during our project.

## 7.3 Incorporating our test suite into SuperTest

Solid Sands has concrete plans to further expand our test suite and incorporate it into SuperTest. This way, a single test suite can be used to create confidence in the conformance of a compiler with the C and C++ language specification, as well as to enhance confidence in the correctness of the loop optimizations that it applies.

For compiler validation, SuperTest uses test programs that compare predefined expected results or properties of the program to the actual results of executing the program. Therefore, to incorporate our loop optimization tests into SuperTest, these expected results need to be determined and code that tests these results needs to be added to our test programs, instead of applying comparison checking between program executions at different optimization levels like we currently do. To do this, properties need to be extracted from the test programs to determine the correct results of executing them with certain input values. Alternatively, this could be done by compiling the test programs with a validated compiler without enabling optimizations, and collecting the result of executing the program. As the results are obtained from a validated compiler, that way correct results are obtained as well.

# Chapter 8

# Related Work

CompCert is a C compiler that is claimed to be formally verified using machine-assisted mathematical proofs [Ler09]. It promises to produce machine code that behaves exactly according to the semantics of the source code. Therefore, it is claimed to be suitable for the compilation of safety-critical software. However, 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 [Bee17]. 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 [YCER11]. They compile randomly produced programs with different optimizing compilers, run the executables, and compare all outputs. If the result of running one executable is different than the others, they know that at least one of the compilers incorrectly optimized the program. 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 like SuperTest [San].

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]. Similarly, Barrett et al. use the intermediate representations from Intel's Open Research Compiler to proof the correctness of optimizations applied by the compiler [BFG+05]. These methodologies rely on the intermediate representation structures of specific compilers and are therefore not generic methodologies that can be used with any C and C++ compiler.

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]. As analysis is done at runtime, this methodology could be applied to any compiler. The authors introduce an accurate way of testing compiler optimizations, but they do not mention what input programs are used for testing or what test coverage is achieved on them. Therefore, while their methodology is suitable to systematically test compiler optimizations for correctness, still a collection of test programs and corresponding inputs that cover these optimizations is needed.

# Chapter 9

# Conclusion

In this thesis, we investigate how a test suite can be created that can be used to create confidence in the correctness of compiler optimizations. We demonstrate that when compiling a simple function, test coverage can drop from 100% at source code level to only 18% on machine code level because of loop optimizations. Hence, if compiler optimizations are not tested for correctness, a significant fraction of the executed code remains untested.

For our test suite we use loop optimization benchmarks as a basis and adapt them to use for correctness testing. Using a decompilation tool, we transform optimized machine code back to a human-readable form, such that we can analyze what test inputs are needed to achieve full machine code coverage. 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. Using the GNATcoverage tool that analyzes coverage at machine code level instead of at source code level, we are able to analyze whether our selection of test inputs fully covers an optimized loop at machine code level.

Our current test suite consists of 40 test programs targeting 10 different loop optimizations. For these optimizations, we are able to extract clear patterns from which test inputs can be selected that robustly cover the optimized machine code. Patterns can be purely based on the behavior of an optimization, but also on the behavior of a compiler, such as the maximum loop unroll factor that it will apply, or on the target hardware, such as the size of the vector registers.

We found that it is not always possible to achieve full machine code coverage on optimized loops, as unsatisfiable branches might be introduced by the compiler. This means that nested contradicting conditional branches are inserted to the code, causing the condition to be impossible to satisfy. For the compilation of all test programs in our test suite by GCC, Clang and ICC, we are able to either achieve full machine code coverage, or to demonstrate that any missed coverage is caused by unsatisfiable branches.

To also thoroughly test the analysis that a compiled performs before applying an optimization, our test suite needs to be expanded with loops targeting the same optimization with different scenarios, as well as with negative examples of loops that are not suitable to be optimized. Additional test programs can still be retrieved from the TSVC benchmark suite [CDL88, MGG+11], as we only partially adapted the benchmark suite for correctness testing. Furthermore, sources like the Loop Repository for the Evaluation of Compiler (LORE) [CGS+17] can be used. To achieve full machine code coverage on these additional test programs, the optimization patterns and corresponding test inputs that we discuss in this thesis can again be used.

Solid Sands has concrete plans to incorporate our test suite into the SuperTest compiler test suite and to expand it with additional test programs based on our findings discussed in this thesis. By incorporating our test suite into SuperTest, a single test suite can be used to create confidence in the conformance of a compiler with the C and C++ language specification, as well as to create confidence in the correctness of the loop optimizations that it applies.

To conclude our project, we answer our research questions below.

**How can we design test programs that target specific compiler optimizations?** To create test programs that specifically trigger compiler optimizations, we use the Test Suite for Vectorizing Compilers (TSVC) [CDL88, MGG+11] benchmark suite as a basis. This benchmark suite consists of loops that target specific compiler optimizations, with the goal of benchmarking the performance of optimizing compilers. We adapt these benchmarks to use for correctness testing instead, by selecting test inputs that cover the optimized benchmarks at machine code level and validating the results of executing them. Besides using the TSVC benchmark suite as a source for our test loops, we also use loops from papers and online resources on compiler optimizations [BJL12, CGS+17, Sar15]. These sources are used to cover optimizations that are not triggered by any of the TSVC benchmarks, such as loop unswitching and strip mining.

**How can we identify conditional branches introduced by compiler optimizations, such that test inputs that fully cover the machine code can be selected?** To identify conditional branches introduced to a program by compiler optimizations, we transform the optimized machine code back to a human-readable programming language using the Snowman decompiler [Der18] and analyze the resulting code. This considerably facilitates analysis compared to assembly code with jumps to labels. This way also variables are traceable to their origin by following all assignment statements in the decompiled code, which is much simpler than analyzing variable state based on the assembly code.

**How can we measure test coverage of a program at machine code level?** As we aim to create tests that cover optimized code at machine code level, test coverage of our test programs also needs to be measured at machine code level as well. To do this, we use the GNATcoverage tool [Adaa]. 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. We parse the output and calculate coverage metrics using the coverage indicators.

**How large is the gap between test coverage at source code level and test coverage at machine code level?** In this thesis we demonstrate that for the compilation of a very simple 4-line function using Clang, test coverage drops from 100% at source code level to 18% at optimized machine code level. This drop is caused by conditional branches introduced by loop optimizations. As optimizations are applied to commonly used loop 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 untested, despite high or even perfect test coverage at source code level.

# Bibliography


[Adaa] AdaCore. GNATcoverage. https://github.com/AdaCore/gnatcoverage.

[Adab] AdaCore. GNATemulator. https://www.adacore.com/gnatpro/toolsuite/gnatemulator.

[Asl] Toheed Aslam. Program Analysis & Transformations. http://slideplayer.com/slide/8989843/.

[Bee17] Marcel Beemster. C99 for-loop defined variable gets incorrect scope. https://github.com/AbsInt/CompCert/issues/211, 2017.

[BFG+05] Clark Barrett, Yi Fang, Benjamin Goldberg, Ying Hu, Amir Pnueli, and Lenore Zuck. TVOC: A Translation Validator for Optimizing Compilers. In *International Conference on Computer Aided Verification*, pages 291–295. Springer, 2005.

[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.

[BJL12] A Berna, M Jimenez, and JM Llaberia. Source code transformations for efficient SIMD code generation, 2012.

[Cal] Callgrind. http://valgrind.org/docs/manual/cl-manual.html.

[CDE+08] 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.

[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.

[Cen16] National Energy Research Scientific Computing Center. Vectorization. http://www.nersc.gov/users/computational-systems/edison/programming/vectorization/, 2016.

[CGS+17] Zhi Chen, Zhangxiaowen Gong, Justin Josef Szaday, David C Wong, David Padua, Alexandru Nicolau, Alexander V Veidenbaum, Neftali Watkinson, Zehra Sura, Saeed Maleki, et al. LORE: A Loop Repository for the Evaluation of Compilers. In *2017 IEEE International Symposium on Workload Characterization (IISWC)*, pages 219–228. IEEE, 2017.

[Cla] Clang: a C Language Family Frontend for LLVM. https://clang.llvm.org/.

[Der17] Yegor Derevenets. Snowman issue: Array declaration sizes are in wrong place. https://github.com/yegord/snowman/issues/137, 2017.

[Der18] Yegor Derevenets. Snowman decompiler. https://github.com/yegord/snowman, 2018.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[Eme16] Amara Emerson. Compiler auto-vectorization: techniques and challenges, 2016.

[GBR] GCC Bug Report: Wrong vectorized code generated for x86_64. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=82108.

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

[GTS] GCC Test Suite. https://github.com/gcc-mirror/gcc/tree/master/gcc/testsuite.

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

[HRCK15] Nabil Hallou, Erven Rohou, Philippe Clauss, and Alain Ketterlin. Dynamic Re-Vectorization of Binary Code. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), 2015 International Conference on*, pages 228–237. IEEE, 2015.

[HVCR01] 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.

[ICC] Intel C++ Compilers. https://software.intel.com/en-us/c-compilers.

[Ins] Arrays, Part III: Vectorization. http://www.insideloop.io/blog/2014/10/14/arrays-part-iii-vectorization/.

[Inta] Intel. Intel Advanced Vector Extensions 512 (Intel AVX-512) Overview. https://www.intel.com/content/www/us/en/architecture-and-technology/avx-512-overview.html.

[Intb] Intel. Intrinsics for Intel Advanced Vector Extensions 2 (Intel AVX2) Instructions. https://software.intel.com/en-us/node/523876.

[ISO11] 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.

[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.

[Jub14] Chae Jubb. Loop Optimizations in Modern C Compilers, 2014.

[KP13] Jakub Křoustek and Fridolín Pokorný. Reconstruction of Instruction Idioms in a Retargetable Decompiler. In *Computer Science and Information Systems (FedCSIS), 2013 Federated Conference on*, pages 1519–1526. IEEE, 2013.

[Kri15] Rakesh Krishnaiyer. Data Alignment to Assist Vectorization. https://software.intel.com/en-us/articles/data-alignment-to-assist-vectorization, 2015.

[Křo14] Jakub Křoustek. *Retargetable Analysis of Machine Code*. PhD thesis, Brno University of Technology, Czech Republic, 2014.

[Ler09] Xavier Leroy. A Formally Verified Compiler Back-end. *Journal of Automated Reasoning*, 43(4):363, 2009.

[mal13] Custom aligned_malloc implementation. https://github.com/dhara04/cracking-the-coding-interview-c--/blob/master/aligned-malloc.cpp, 2013.

[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.

[PW86] David A Padua and Michael J Wolfe. Advanced Compiler Optimizations for Supercomputers. *Communications of the ACM*, 29(12):1184–1201, 1986.

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

[Sar15] Mark Saroufim. Description of commonly done compiler optimizations in C. https://github.com/msaroufim/C-compiler-optimizations, 2015.

[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.

[Und] Undefined Behavior Sanitizer. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html.

[Val] Valgrind. http://valgrind.org/.

[Vla15] Andrey Vladimirov. Optimization Techniques for The Intel Mic Architecture. Part 2 of 3: Strip-mining for Vectorization, 2015.

[vVBG18] Remi van Veen, Marcel Beemster, and Clemens Grelck. Correctness Testing of Loop Optimizations in C and C++ Compilers. In *Eleventh Seminar on Advanced Techniques & Tools for Software Evolution (SATToSE)*. http://sattose.wdfiles.com/local--files/2018:schedule/SATToSE_2018_paper_13.pdf, 2018.

[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.