Verification of Optimization Correctness with SuperTest

If you think advanced compiler optimizations are robust and well-tested, you are wrong. Our recent experiments with optimization testing have uncovered errors in every compiler technology that we got our hands on. Our conclusion is that advanced optimization testing is currently an underdeveloped skill of compiler developers and that action is required. We introduce our new optimization tests in SuperTest, the most advanced compiler test suite available.

Compiler optimizations have huge economic value. Comparing unoptimized code with optimized code can demonstrate a fifteen-fold increase in the execution speed of the generated program after optimization. That is a large factor, but it is not uncommon for advanced loop optimizations, such as vectorization, to achieve it. As for the economic value, fifteen times greater execution efficiency can be translated into a slower and much cheaper target processor, less heat dissipation, a more compact build, and simultaneously, a reduction in (battery) power consumption by a factor of 15. When this happens in embedded or automotive applications, there are real benefits to using compiler optimizations.

For a typical compiler, more than half of its source code is optimization related. That is significant, and errors can occur in that part of the compiler as well as in other parts.

1 "Optimizations Don't Exist"

Over the past year, we have worked on improving the optimization testing capabilities of SuperTest. When writing any test, including tests for SuperTest, it is preferable to start from the language specification. However, this is not straightforward for optimizations. From the C/C++ language definition point of view, optimizations hardly exist. In C11:5.1.2.3, it says:

> *The semantic descriptions in this International Standard describe the behavior of an abstract machine in which issues of optimization are irrelevant.*

The language definition specifies the behavior of every particular language construct, but it does not specify how or when that behavior is achieved. Optimization is a so-called 'non-functional' requirement. This makes it hard, if not impossible, to verify optimizations against a specification.

Our belief, backed up by real-world experience, is that SuperTest has always done a pretty good job of optimization testing. Many of its tests are specifically designed to test compiler optimizations. For example, a text search for tail recursion in the SuperTest suites immediately reports about ten tests. That does not include tests that accidentally test for tail recursion or were not documented as doing so. Because of the nature of compilers as a pipeline of steps, every test is exposed to all components of the pipeline, including all optimization stages. This means that the chance of tests unintentionally hitting optimizations is high, which is in fact what we see.

The weak link is that this is not good enough to meet the formal requirements of functional safety standards, and rightly so. These standards demand a less 'accidental' approach, which requires a rigid framework to link tests to the requirements of optimization. Providing this framework is what we set out to do. Our goal is not simply to meet the requirements of functional safety standards, but also to actually create a high-quality optimization test suite that has wide applicability.

2 Inspiration from Benchmarks, Lessons Learned

To get inspiration for code that triggers optimization, we turned to benchmarks – some of them well known in the field of performance testing. Not only are there (artificial) benchmarks that are written explicitly to trigger optimizations, but compiler developers have also worked hard to improve the performance of benchmark code by adding optimizations to their compilers. From this we learned a number of lessons that can be summarized as follows:

Benchmarks are not the best tests of correctness of compiler optimizations.

Lesson 1: Benchmarks do not always verify their results

The number one metric for a benchmark is how fast its compiled code runs on the target processor. There is no number two metric. High priority is not always given to verifying if what is computed is also correct. So how do we know that optimizations performed on the benchmark are performed correctly? For some graphics benchmarks, you have to review the generated image to assess correctness. Clearly that approach does not work in a continuous integration environment.

Lesson 2: Benchmarks are not written to deal with different data models

In the development of SuperTest, we are constantly aware of the many different data models used in embedded computing platforms. Computing with a different data model (for example, using 24-bit instead of 32-bit integers) may result in different, potentially incorrect, results. This must be taken into account when optimizing. But referring back to Lesson 1 above, if the result is not verified, you have no evidence that optimizations take the data model into account.

Lesson 3: Benchmarks may not be free of undefined behavior

Undefined behavior can easily happen when a benchmark is compiled for a smaller data-model than it was intended for. The danger of this is that some compilers use the assumed absence of undefined behavior as a property that can be optimized. For example, if a compiler analysis shows that a branch in the code leads to undefined behavior, perhaps triggered by an overflowing signed integer computation, the compiler may rightly assume that the branch is never executed and remove it. This may not have been the intention of the benchmark, but it does make it run faster.

Lesson 4: Benchmarks do not execute all the generated code

Even when a benchmark verifies that its computed result is correct, if it does not execute all the generated code there is no evidence that the transformation that created the non-executed code is correct. This is not simply a case of 'dead-code' in the benchmark being optimized away. On the contrary, when compiled without optimization, all generated code may be executed. The reason for not-executed code existing after optimization is that optimizing transformations often duplicate and specialize code. As part of this process, a much more complicated control flow is constructed to differentiate between special cases and make them faster. If every special case is not present in the benchmark, some parts of the generated code will not be executed.

It should now be clear that even if your compiler manages to optimize a benchmark suite really well, there is no guarantee that its optimizations are correct.

These ‘lessons learned’ are applied in our construction of the SuperTest optimization test suite. We have done as much as we can to make sure that computed results are verified, that tests are applicable to the data model the suite supports, that they contain no undefined behavior, and that they execute all the generated code.

3 Executing All the Generated Code (Lesson 4)

Not all benchmarks suffer from these deficiencies. EEMBC’s CoreMark®, for example, goes through great lengths to verify its computed result. But the incomplete execution of all the generated code is a real issue, and it is not addressed by any of the benchmarks that we know of.

Verifying test results, working with the right data model and having no undefined behavior are all properties of the test source code. However, executing all the generated code after optimization is dependent on the transformations that are applied by the compiler. We cannot know these beforehand, and although we cannot guarantee that all generated code is executed, we have worked hard to make sure that this property is true.

To analyze generated code execution, we have used a number of different compilers and performed a run-time coverage analysis at the assembly level. In effect, we have performed assembly-level MC/DC analysis, looking both at structural coverage and branch coverage. Then, for all possible optimizations, we looked at the code specialization that the compiler applied and analyzed the specific inputs that are needed to hit all the generated specialized code.

For example, take the following relatively simple loop:

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

This loop is sufficiently complex that the compiler does not apply the algebraic analysis that would remove the loop completely. Instead, the loop can be vectorized. At the source code level, a single call of the loop with a value of **n** different from zero achieves full structural code coverage. Also, the single branch due to the loop condition is called in both ways at run-time, achieving full branch coverage.

When we look at the generated assembly code at a high optimization level, we see that the code has expanded significantly and contains no less than fifteen conditional branches. Calling the loop with **n=999** (a non-trivial number that is not a multiple of the loop-unroll factor or the vector length) results in about 80% code coverage (reasonably good) but coverage of less than half of the control flow edges in the generated code (not good).

To achieve maximal code and branch coverage, the optimized code has to be called with at least *five* different values of the loop value **n**.

One interesting, perhaps surprising, discovery was that compilers generate redundant conditional branches. This was observed in multiple, different, compiler technologies, so it is not merely an artifact of one compiler. It happens in this loop too. Redundant branches recheck a condition that was established earlier in its flow of control. As a result, the condition always has the same value and the redundant conditional branch is always taken in the same direction. Full branch coverage for these redundant branches cannot be achieved!

The unwelcome result of redundant branches is that *maximal* code and branch coverage is less than *full* code and branch coverage. Also, the redundant branches are not easy to analyze, so understanding when maximal coverage is not equal to full coverage is a non-trivial and, above all, time-consuming exercise.

Taking all of this into account, we were able to create a generalized method of constructing, for every test, a range of test-cases that achieved maximal coverage for all

the compilers we looked at. Since our methods are robust and work independently of a particular compiler technology, we believe that our tests and test-cases are highly likely to achieve maximal coverage for other compiler technologies as well.

4 Results

Using our old and new optimization tests, we have found optimization errors in all the compiler technologies we have analyzed so far. This does not mean that optimizations cannot be trusted in general, but it does suggest that it pays to tread carefully.

Here is the result of running a SuperTest test on a compiler that is commonly used in embedded systems:

```
s[0] = 42;
*(sp[0]) = -1;    /* *(sp[0]) is an alias of s[0] */
if( s[0] == 42 ){ /* Incorrectly yields true */
```

This error is the result of an incorrect value propagation, from the first statement to the third, through variable **s[0]**, while that variable was modified by the second statement through an alias. The dangerous aspect of this compiler error is that it is due to an optimization that is applied without any of the optimization options being given to the compiler. So even if you think that no optimization takes place, it still happens. This is OK as far as the language definition is concerned, but many developers would not expect it to happen.

Here is another result from the optimization suite, compiled with Intel's ICC compiler. This compiler is well regarded and known for its high-performance loop optimizations, but, as shown here, it is not perfect:

```
void s482(double *a, double *b, double *c, int len) {
    int i;
    for (i = INT_MIN; i < INT_MIN+len; i++) {
        a[i-INT_MIN] = b[i-INT_MIN] + c[i-INT_MIN];
        if (c[i-INT_MIN] > b[i-INT_MIN])
            break;
    }
}
```

The special property of this test is that its iterator **i** operates close to the lower bound of the integer domain, something that the optimizer is not prepared for. As a result, the program crashes at run-time with a segmentation fault. We also found optimization errors in other trusted technologies such as CLANG/LLVM compilers, GCC compilers and Microsoft's compiler.

5 Conclusion

The most important conclusion is that if you want to use compiler optimizations, you must know the weaknesses of your compiler. We have found no compiler technology immune to optimization errors. Sometimes the errors are obscure and easy to avoid. In other cases, they are so severe that you should consider not using a specific optimization. So make sure that you have a test-suite that aims to verify the correctness of optimizations performed by the compiler – and not only a benchmark suite aimed at performance analysis.

If you need to qualify the use of optimizations in a compiler for functional safety or other mission-critical requirements, our new optimization test suite provides a solid basis. Optimization is not a functional requirement of the compiler, and every compiler technology has its own specific implementation strategy for optimization. Our optimization test-suite, in conjunction with the many optimization-triggering tests in the core suites of SuperTest, is aimed at triggering many, if not all, of the optimizations in any specific compiler.