Optimisation verification in embedded system design

By Dr Marcel Beemster, Chief Technical Officer, Solid Sands

Advanced compiler optimisations are not always robust and well-tested. Recent experiments with optimisation testing have uncovered errors in every compiler technology available, leading to the conclusion that advanced optimisation testing is currently an underdeveloped skill of compiler developers, requiring urgent action.

Compiler optimisations have huge economic value. Comparing unoptimised with optimised code demonstrates fifteen-fold faster execution speed of the post-optimisation generated programme. That is a large factor, but not uncommon to achieve for advanced loop optimisations, such as vectorisation. As for economics, fifteen times greater execution efficiency means a slower hence cheaper target processor, with lower power consumption and heat dissipation, and potentially a smaller size – important in almost all embedded applications.

For a typical compiler, over half its source code is related to optimisation; being a significant part, errors do occur.

"Non-existent" optimisations?

When writing any test, it is preferable to start from the language specification – a not-so-straightforward task for optimisations; from a C/C++ language definition point of view, optimisations hardly exist. The C programming standard C11 in section 5.1.2.3 states: "The semantic descriptions in this International Standard describe the behaviour of an abstract machine in which issues of optimisation are irrelevant."

The language definition specifies the behaviour of every particular language construct, but it does not specify how or when that behaviour is met. Optimisation is a so-called "non-functional" requirement, making it hard – if not impossible – to verify against a specification.

A more complicated control flow differentiates between special cases and makes them faster

To avoid these challenges, we developed new optimisation tests in our compiler test suite SuperTest. As an example, a text search for tail recursion in the SuperTest suites immediately reports around ten tests, excluding those that accidentally test for tail recursion or were not documented as doing so. Because of the nature of the compiler as a series of steps, every test is exposed to every component of the pipeline, including all optimisation stages. This means the chance of tests unintentionally hitting optimisations is high, which is 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 optimisation requirements.

Lessons learned

To get inspiration for code that triggers optimisation, we turned to benchmarks – some of them well known in the field of

performance testing. Not only are there (artificial) benchmarks written explicitly to trigger optimisations, but compiler developers have also worked hard to improve the performance of benchmark code by adding optimisations to their compilers. From this we learned a number of lessons that can be summarised as follows:

Lesson 1: Benchmarks are not the best tests of compiler. optimisation correctness, since they don't always verify their results.

The number-one metric for a benchmark is how fast its compiled code runs on the target processor.

But, high priority is not always given to verifying if what's computed is also correct. Which begs the question: how do we know if optimisations performed on the benchmark are correct?

For some graphics benchmarks, the generated image simply needs to be reviewed to assess correctness, which may not always work in a continuous integration environment.

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

While developing SuperTest, we considered 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, even incorrect, results. Going back to Lesson 1, if the result is not verified you have no evidence that optimisations take the data model into account.

Lesson 3: Benchmarks may not be free of undefined behaviour.

Undefined behaviour can easily happen when a benchmark is compiled for a smaller data-model than intended. The danger of this is that some compilers use the assumed absence of undefined behaviour as a property that can be optimised. For example, if a compiler analysis shows a branch in the code that leads to undefined behaviour, perhaps triggered by an overflowing signed integer computation, the compiler may 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.

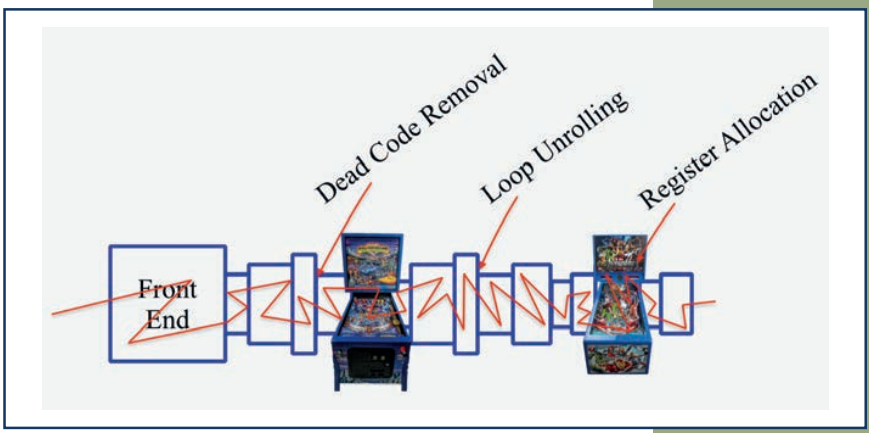
Even when a benchmark verifies its computed result as correct, if it doesn't execute all the generated code there's no evidence that the transformation that created the non-executed code is correct. This is not simply a case of 'dead-code' in the benchmark being optimised away. On the contrary, when compiled without optimisation, all generated code may be executed. The reason for not-executed code existing after optimisation is that optimising transformations often duplicate and specialise code. As part of this process, a much more complicated control flow is constructed to differentiate between special cases and make them faster. Unless every special case is 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 optimise a benchmark suite really well, there's no guarantee that its optimisations are correct.

Executing the entire generated code

Not all benchmarks suffer from these deficiencies, however. EEMBC's CoreMark, for example, goes to great lengths to verify its computed results.

Verifying test results, working with the right data model and having no undefined behaviour, should be all properties of the test source code, even though executing all the generated code after optimisation is dependent on the transformations applied by the compiler. These can't be known beforehand, and although execution of all generated code is not guaranteed, in SuperTest we have ensured this property is true.



Optimisations by stages

This graphic shows an abstract view of the chaotic journey a program takes through the compiler, like the ball in a pinball machine.

A compiler is a pipeline with stages, where each stage performs a specific analysis and/or transformation of the code. Many of these stages are optimisations that improve some aspects of code quality for the target processor. Whilst the code's path is not truly chaotic, and is usually deterministic, it is true that a small change in the source code or a minor update to an early compiler transformation can have a huge effect on the program's shape as it moves from one stage to the next. This makes it difficult for a test to aim at a specific optimisation inside the compiler.

To analyse 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 optimisations, we looked at the code specialisation the compiler applied, and analysed the specific inputs needed to hit all the generated specialised 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;
}
}
```

It is a sufficiently complex loop that the compiler doesn't apply algebraic analysis that would remove the loop completely. Instead, the loop can be vectorised: At 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.

Looking at the generated assembly code at a high optimisation 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's 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 so good).

To achieve maximal code and branch coverage, the optimised 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's not merely an artifact of one compiler. It happens in this loop, too. Redundant branches re-check 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 analyse, so understanding when maximal coverage is not equal to full coverage is a non-trivial and, above all, time-consuming exercise.

Taking all this into account, we created a generalised method of constructing for every test a range of test cases that achieve maximal coverage for all the compilers we analysed. 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.

Old and new

Through our analyses, we found optimisation errors in all the compiler technologies. This doesn't mean that optimisations can't be trusted in general, but it does suggest that you should tread carefully.

Here is the result of running a SuperTest test on a compiler 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 a result of an incorrect value propagation, from the first statement to the third, through variable s[0], whilst 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 optimisation applied without any optimisation options being given to the compiler. So, even if you think no optimisation takes place, it still happens. This is alright as far as the language definition is concerned, but many developers would not expect it to happen.

Here is another result from the optimisation suite, compiled with Intel's ICC compiler. This compiler is well regarded and known for its high-performance loop optimisations, 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 the optimiser is not prepared for. As a result, the program crashes at run-time with a segmentation fault. We also found optimisation errors in other trusted technologies such as CLANG/LLVM compilers, GCC compilers and Microsoft's compiler.

Know your compiler

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

Optimisation is not a functional requirement of the compiler, and every compiler technology has its own specific implementation strategy for optimisation. EW