Compile-Time Function Call Interception for Testing in C/C++

Gábor Márton Zoltán Porkoláb

Eötvös Loránd University, Faculty of Informatics
Dept. of Programming Languages and Compilers
H-1117 Pázmány Péter sétány 1/C Budapest, Hungary
martongabesz@gmail.com gsd@elte.hu

1. Performance Evaluation

We did not notice any degradation in the run time and in the memory usage of the compilation process itself when our instrumentation was turned on. We measured the runtime performance of the compiled instrumented code with the help of the Adobe C++ Performance Benchmark [1]. Adobe's primary goals with their benchmark are

- to help compiler vendors identify places where they may be able to improve the performance of the code they generate
- to help developers understand the performance impact of using different data types, operations, and C++ language features with their target compilers and OSes.

We customized their benchmark to our needs. We removed test suites about loop unrolling, constant folding and loop invariants. We use exactly 3 different test suites, they measure overhead about different abstractions:

- The function objects test suite compares the performance of function pointers, functors, inline functors, standard functors, and native comparison operators. (This test suite contains only one test case.)
- The Stepanov abstraction test suite examines any change in performance when adding abstraction to simple data types. For instance, a value wrapped in a class may perform worse than a raw value. Also, a value recursively wrapped in a struct or class may perform worse than a raw value. There are several different test cases in this test suite. For instance, there is a test case for measuring the performance of an insertion sort when several layers of abstractions are applied.
- The Stepanov vector test suite examines any change in performance when moving from pointers to vector iterators. Vector iterators may perform worse than raw pointers. There is only one test case in this test suite, which measure the performance in case of applying quicksort on a `std::vector`.

Note that a good optimizing compiler with `-O2` or `-O3` optimization level should not produce any performance penalty on behalf of the abstractions. For instance, the MSVC 2008 compiler has an abstraction penalty ratio not greater than 1.82 [2].

We compiled the test suites with different compiler flags and we compared the absolute total time of run time of each test cases. On Figure 1 we show the total time for the function objects test case with the different compiler setups. Our instrumentation is turned on with the `-fsanitize=mock` option. By providing the `-DSUB` switch we define a macro. If that macro is present then we do substitute functions in the busy loop of each test cases. With the `-finstrument-functions` setup we define the `__cyg_profile` functions in a standalone, separate translation unit. Figure 2 presents the total absolute time for the vector use case from the Stepanov vector test suite. Similarly, Figure 3 represents the total time in one test case of the Stepanov abstraction suite.

On Figure 4 we can see the normalized total times. We display for each compiler setup the execution times of all the test cases (normalized by the longest execution time per test case).

Both our instrumentation and `-finstrument-functions` causes performance degradation. In most cases our technique performs similarly to `-finstrument-functions`. Our method exchanges two extra function calls (`__cyg_profile_func_enter` and `__cyg_profile_func_exit`) with one extra function call to `__fake_hook` followed by an efficient lookup.

When we enable our instrumentation and we do replace some functions, then we may experience some performance degradation compared to when we just simply enable the new instrumentation (up to 15%). The reason behind this is that when we replace a function we always take that branch which calls a function via a pointer and in some cases we may loose important hardware optimization opportunities (e.g. prefetching of instructions). Note that there is an important optimization possibility in case of the non-replacing branch. If we could include the definition of the `__fake_hook` function when we emit the LLVM IR for a call expression than the upcoming optimizer passes of the compiler might be able to inline it. Also, by this inlining, on the non-replacing branch, further inlining optimization opportunities might be exploited. We plan to investigate the feasibility of this inlining in the future.

We experienced that with our instrumentation, the size of the binary may grow bigger. In the case of a simple C program we measured around 15% (bzip2). In the case of a template heavy program (Stepanov abstractions test suite) we measured that the executable could be up to 3 times bigger compared to an `-O2` optimized binary. We measured very similar size growth in case of the `-finstrument-functions` feature.

We performed the measurements on a Linux machine with an Intel(R) Core(TM) i7-4610M CPU @ 3.00GHz processor and with 16GB RAM. The given CPU is a laptop-class hardware that scales the frequency dynamically from 0.8Ghz to 3.7Ghz, therefore we turned off turbo boost and frequency scaling by using the appropriate ACPI kernel driver.

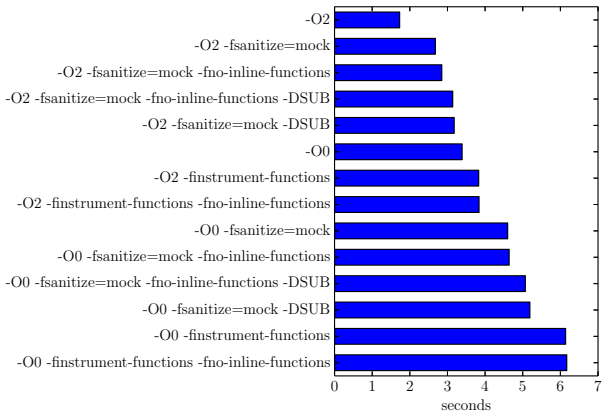

Figure 1. Total absolute time for function objects

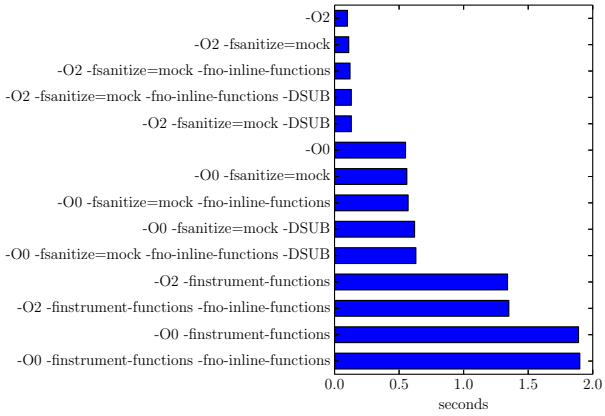

Figure 2. Total absolute time for vector quicksort

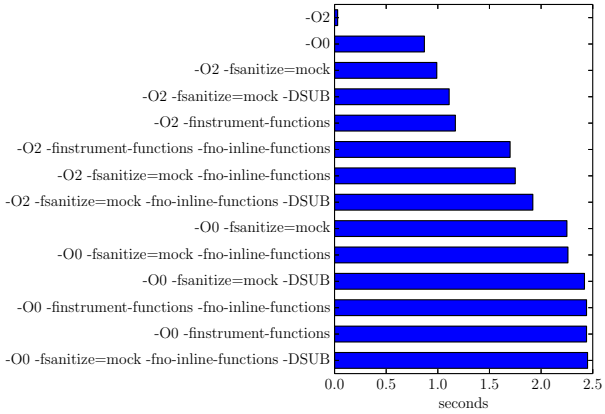

Figure 3. Total absolute time for abstraction insertion sort

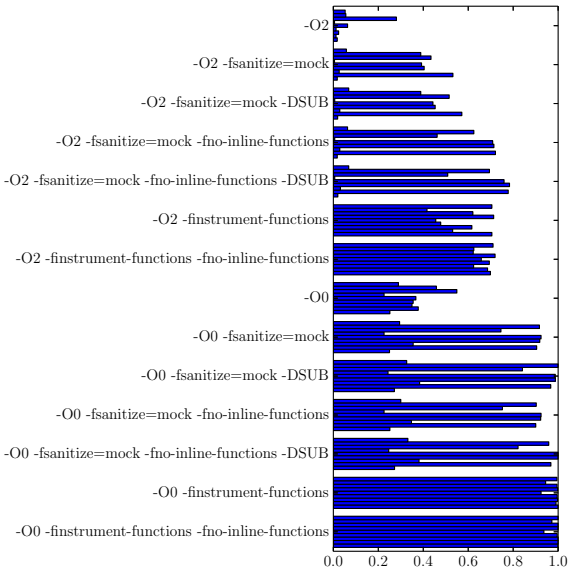

Figure 4. Normalized total absolute times

References

[1] Adobe. C++ performance benchmarks, 2017. URL `https://stlab.adobe.com/performance`.

[2] Adobe. Abstraction penalty with msvc 2008, 2017. URL `https://stlab.adobe.com/wiki/index.php/Performance/Analysis/Example3`.