Easy::Jit: Compiler Assisted Library to Enable Just-in-Time Compilation in C++ Codes

Juan Manuel Martinez Caamaño
jmmartinez@quarkslab.com
Quarkslab
France

Serge Guelton
sguelton@quarkslab.com
Quarkslab
France

ABSTRACT

Compiled languages like C++ generally don't have access to Just-in-Time facilities, which limits the range of possible optimizations. This paper introduces an hybrid approach that combines classical ahead of time compilation with user-specified dynamic recompilation of some functions, using runtime information to improve compiled code.

CCS CONCEPTS

• **Software and its engineering → Just-in-time compilers**; *Runtime environments*; Frameworks;

KEYWORDS

LLVM, Just-In-Time Compilation, C++

ACM Reference Format:
Juan Manuel Martinez Caamaño and Serge Guelton. 2018. Easy::Jit: Compiler Assisted Library to Enable Just-in-Time Compilation in C++ Codes. In *Proceedings of 2nd International Conference on the Art, Science, and Engineering of Programming (<Programming'18> Companion).* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3191697.3191725

1 INTRODUCTION

C++ programs are typically compiled once, Ahead-of-Time, before deploying or distributing the application. The category is best represented by traditional compilers such as Clang or GCC.

However, there is still room for improvement. Performance critical optimizations may depend on concrete information only available at runtime: the system sub-architecture, the value taken by a variable, a function available in a plugin, the trace of the execution, etc.

Just-In-Time(JIT) compilation is often used in the context of dynamic behavior [1] and its static counterpart, code versioning, has been used to statically specialize a function for a known, discrete set of parameters, in domain specific context as in ATLAS [3]. Modern languages like Julia even propose an automatic hybrid approach where some kernels are recompiled on the fly using extra type informations [2].

We propose a framework in the form of a library and a compiler plugin to enable on-demand JIT compilation for C++ codes. The programmer interacts directly with the abstractions from the library, while the compiler plugin parses the library calls and generates some support code around it as well as embedding a bitcode implementation of a subset of the program. It removes the need for tracing often found in dynamically compiled language: the developer feeds the JIT compiler with the information concerning hot functions and argument that needs to be specialized on.

The code of the framework, the examples, and the benchmarks presented in this article are available at https://github.com/jmmartinez/easy-just-in-time.

Section 2 introduces the approach by the means of an example. In section 3 some performance results are presented. Finally, in section 4 the future of this approach and conclusions are discussed.

<Programming'18> Companion, April 9–12, 2018, Nice, France

ACM ISBN 978-1-4503-5513-1/18/04…$15.00
https://doi.org/10.1145/3191697.3191725

2 EXAMPLE

The code in Listing 1 shows a call to a text-book convolution kernel being applied on a video stream obtained from a webcam. The values taken by mask, mask_size and mask_area change periodically depending on the user input. The frame dimensions (parameters img.rows and img.cols) and number of channels (parameter img.channels()) tend to remain constant during the entire execution; however, its impossible to know their values at compile-time.

Listing 1: Original kernel invocation

```
kernel(mask, m_size, m_area, img.ptr(0,0),
  out.ptr(0,0), img.rows, img.cols, img.channels());
```

With this information, we adapted this code to use an optimized version generated at runtime as shown in Listing 2. The main abstraction of the library is the easy::jit function. This function mimics the semantics of std::bind. In this example, a specialized version of kernel is generated, with two parameters, corresponding to in and out respectively and returning void. The values for mask, mask_size, mask_area, rows cols and channels are fixed to the values passed to the easy::jit function.

Listing 2: Runtime code generation

```
auto opt = easy::jit(kernel, mask, m_size, m_area,
  _1, _2, img.rows, img.cols, img.channels());
kernel_opt(img.ptr(0,0), out.ptr(0,0));
```

Still, triggering the compilation for each frame is not efficient. Using a code cache to store previously generated versions is a common solution, illustrated in Listing 3.

Listing 3: Runtime code generation using a cache

```
static easy::Cache cache;
auto opt = cache.jit(kernel, mask, m_size, m_area,
  _1, _2, img.rows, img.cols, img.channels());
```

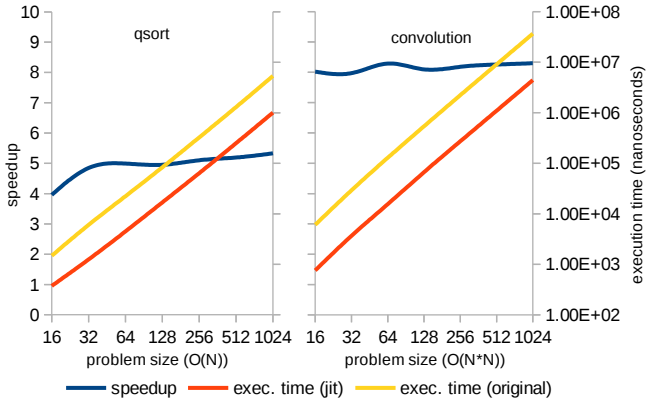

Figure 1: Execution times and speedup. QSort(left). Convolution(right).

Table 1: Just-In-Time compilation overheads.

benchmark	compilation time (ms)	cache-hit time (ms)
convolution	67	0.00338
qsort	129	0.00276

Following section presents some performance results regarding this approach.

3 EXPERIMENTS

We ran our experiments under Linux 4.13 on a Intel Core i7-6600U processor at 2.60GHz. We ran two examples, i) a convolution benchmark, where the main kernel is specialized for the image bounds and mask values and bounds, and ii) a quick sort benchmark, where specialization is done on the compare function used.

Figure 1 shows the obtained speed-up over the statically compiled version with -O2 of the code and the time taken by the benchmarks.

The results show a clear performance advantage by the kernels using runtime code specialization. The specialized versions execute around 8× faster for the convolution kernel and 5× faster for the quick-sort kernel. This speedup remains independent of the input size.

Table 1 depicts, for each benchmark, the time taken by the JIT compiler to obtain a specialized version of the function and the time taken by the code cache lookup.

The time taken by the JIT compiler to generate specialized versions of the code remains very high. Nevertheless, this cost can be compensated by the use of a code-cache, as the performance impact of a hit in the cache is negligible.

4 CONCLUSION & FUTURE WORK

There is still room for improvement for statcially compiled code when performances are strongly tied to runtime parameters.

Our framework provides a clean interface for on-demand JIT compilation, that allows to take advantage of program properties only known at runtime. The experiments presented in this article show that our approach remains profitable, nevertheless, work must be done to reduce the introduced runtime overhead.

We foresee more advanced optimizations enabled by our framework. Among them, we highlight: devirtualization of virutal function calls; composition of dynamically generated functions; and code generation from data structures, by specializing an eval function for an AST.

ACKNOWLEDGMENTS

We would like to thank Quarkslab for its support in the development of the project.

REFERENCES

[1] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. 2000. Overview of the IBM Java Just-in-time Compiler. *IBM System Journal* 39, 1 (Jan. 2000), 175–193.

[2] Scott Thibault, Charles Consel, Julia l. Lawall, Renaud Marlet, and Gilles Muller. 2000. Static and dynamic program compilation by interpreter specialization. *Higher-Order and Symbolic Computation* 13, 3 (Sept. 2000), 161–178.

[3] R. Clint Whaley and Jack J. Dongarra. 1998. Automatically Tuned Linear Algebra Software. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC '98)*. IEEE Computer Society, Washington, DC, USA, 1–27.