

# LLVM and JitBuilder: Measuring and Comparing the Overhead of two API-based JIT Frameworks

Eric Coffin

Faculty of Computer Science  
University of New Brunswick  
Fredericton, NB, Canada  
eric.coffin@unb.ca

## ABSTRACT

Just-in-Time compilation has allowed for significant performance gains during the run-time of applications. Two popular open-source JIT frameworks are LLVM MCJIT and OMR JitBuilder, both of which can be embedded within applications, offering interfaces to define and generate native code at run-time. LLVM is a collection of modular compiler and toolchain components, while MCJIT is a framework based on these components to provide JIT compilation. Similarly, JitBuilder is a JIT framework based on the OMR compiler and runtime components. In this report we discuss the different approaches these two frameworks employ. In addition, we measure the overhead of each framework while compiling and then executing a small handful of functions. We found that while LLVM required a larger memory footprint, in certain cases it was able to generate code more quickly. Furthermore, the code LLVM generated typically offered high throughput. JitBuilder, a relatively young project compared to LLVM, does not currently expose the underlying configuration of TRJIT. Instead, it locks the compilation level to the *warm* setting, thus limiting the opportunities for optimizations.

## CCS CONCEPTS

• **Software and its engineering** → **Compilers.**

## KEYWORDS

compilers, just-in-time, optimization

## 1 INTRODUCTION

Just-in-time compilation, or JIT compilation, is a technique to improve the run-time binary translation of an application [1]. Using information collected from the running application, JIT compilers can further optimize generated code. For instance, by collecting profile information on executed code paths, JIT compilers can generate code that is optimized for hot-paths [2]. A JIT compiler might also inline entire blocks of code, or generate an inline cache to speed up the dispatch of polymorphic method calls [3].

Popular high-level language runtimes for Java make heavy use of JIT compilers, allowing their workloads, to execute much faster than if they were entirely interpreted [4]. Given that JIT compilation can add significant overhead to a workload, compilation is typically applied selectively to the code executed most frequently. Furthermore, given that compilation can be viewed as a continuum, where slow, unoptimized code is cheap to generate, and where

fast, optimized code is expensive to generate, a compiler will often support multiple optimization levels [2]. A runtime with such an optimizing JIT compiler can then employ a staged compilation strategy which would allow the runtime to apply compilation and optimization according to heuristics [5].<sup>1</sup>

Considering JIT compilers may also need perform some of the optimizations found in a static compiler such as common sub-expression elimination, loop unrolling and constant propagation [9], an application developer, or language designer interested in enhancing run-time performance by adding a JIT compiler, will have a large engineering task ahead of them. In addition, for projects targeting multiple architectures, considering JIT compilers generate native, or architecture specific code, the effort required will increase significantly. It is no surprise then, that libraries or frameworks, encapsulating proven, high-performance JIT compilers are available to software engineers today. For an engineer interested in incorporating an existing JIT framework, it is useful to consider the following questions when making their selection:

- How much larger is the application binary with the inclusion of the JIT framework?
- How is the resident state set for the application effected?
- How is the working state set of the application effected during JIT compilation?<sup>2</sup>
- How much processor overhead does the JIT compiler have?
- How configurable is the JIT framework?
- What is the quality of the generated JIT code and what effect does this have on throughput?
- How easy is it for the programmer to incorporate and use the JIT framework?

In this report, we will consider those questions while looking at two such JIT frameworks: LLVM MCJIT [11] and OMR JitBuilder [12]. In Sections 2, and 3 we will discuss the background of each compiler and consider the techniques they employ. In Section 3, we outline the methods we used to answer those questions. In Section 4, we will consider the results. In Section 5, we will discuss related work. In Section 6, we will look at potential future work, and finally in Section 7, we will summarize our findings.

<sup>1</sup> An example of a staged, or tiered compilation strategy can be seen with the Testarossa JIT compiler (TRJIT) in the OpenJ9 JVM [6, 7]. Here an invocation threshold must be met before the JIT compiler will compile a method. Additionally, separate thresholds can be associated with each optimization level [8]. We will discuss TRJIT in more detail when we look at JitBuilder in Section 2.2.

<sup>2</sup> An application's working set size (WSS) can be defined as the subset of resident set size pages (RSS) that are active specific time interval [10].

```

1 define i32 @mul_add(i32 %x, i32 %y, i32 %z) {
2   entry:
3     %tmp = mul i32 %x, %y
4     %tmp2 = add i32 %tmp, %z
5     ret i32 %tmp2
6 }

```

**Listing 1: LLVM IR for a function multiplying  $x * y$  and adding  $z$  [14]**

## 2 BACKGROUND

In this section we will discuss the background of the two JIT compiler frameworks we are interested in: LLVM MCJIT and OMR JitBuilder. In particular, we will focus on the motivation for each framework, as well as discuss the techniques and features they provide.

## 3 LLVM

LLVM, which at one time stood for Low Level Virtual Machine, is a popular set of open-source, modular compiler and toolchain components [13]. The compiler framework was originally designed to provide analysis and transformation for an application throughout its entire lifetime: from initial compilation and linking, through to runtime and even while the application was offline (see Figure 1). To achieve this ambitious goal, the framework utilizes a well defined, human-readable, intermediate representation called LLVM IR. The IR, which is initially generated by the front end can be packaged with the target architecture binary along with profiling instructions for later runtime compilation (JIT) as well as more aggressive offline optimizations. Several important characteristics of LLVM IR as follows:

- The IR maintains Static Single Assignment (SSA) form with unlimited virtual registers.
- Each register is of one of four primitive types: boolean, integer, floating-point or pointer.
- Similar to RISC, memory operations are carried out in registers, and between registers and memory using Load and Store instructions.
- The IR is limited to 31 opcodes.
- The IR is organized into basic blocks which must be composed into valid control flow graphs, simplifying the work required for various optimizations.

This report will focus on LLVM’s JIT component, which can be accessed through the MCJIT API. The MCJIT framework provides an API that accepts IR, generates optimized machine code, and provides a function pointer for calling the generated code. The JIT compiler offers several levels of optimization: none, less, default, and aggressive. It should be noted that the JIT compiler by default does not perform any IR optimizations or transformations. Instead, a developer must pass the generated IR to a PassManager with specific optimizations they intend to apply. These passes can be categorized as analysis passes, or transformation passes [15]:

- Analysis Passes: collect information about IR for use later by transformations, for debugging or for visualization. A few examples are *print-callgraph*, *print-function*, and *iv-users* for printing the users of a particular induction variable. There are roughly 40 such passes available.

- Transformation Passes: These typically modify IR. Examples include *adce* for dead code elimination, *instcombine* for combining redundant instructions, or *peephole* optimizing, and *tailcallelim* for eliminating tail calls. There are roughly 60 such passes available.

This optimized IR will then be used by the ExecutionEngine when generating code for the target architecture.

## 4 METHODOLOGY

Methodology

### 4.1 Results

## 5 RELATED WORK

Related work.

## 6 FUTURE WORK

Future Work

## 7 SUMMARY

Summary

## 8 ACKNOWLEDGEMENTS

This research was conducted within the Centre for Advanced Studies—Atlantic, Faculty of Computer Science, University of New Brunswick. The authors are grateful for the colleagues and facilities of CAS Atlantic in supporting our research. The authors would like to acknowledge the funding support provided by the Atlantic Canada Opportunities Agency (ACOA) through the Atlantic Innovation Fund (AIF) program. Furthermore, we would also like to thank the New Brunswick Innovation Foundation for contributing to this project.

## REFERENCES

- [1] John Aycock. A Brief History of Just-in-time. *ACM Comput. Surv.*, 35(2):97–113, jun 2003.
- [2] Jim Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.
- [3] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP ’91, page 21–38, Berlin, Heidelberg, 1991. Springer-Verlag.
- [4] Iffat H. Kazi, Howard H. Chen, Berdenia Stanley, and David J. Lilja. Techniques for Obtaining High Performance in Java Programs. *ACM Comput. Surv.*, 32(3):213–240, September 2000.
- [5] Mark Leone and R. Dybvig. Dynamo: A staged compiler architecture for dynamic program optimization. 04 1999.
- [6] Eclipse Foundation. Eclipse OMR, 2018. <https://projects.eclipse.org/projects/technology.omr>, Last accessed on 2019-06-19.
- [7] IBM Knowledge Center. How the JIT compiler optimizes code. [https://www.ibm.com/support/knowledgecenter/en/SSB23S\\_1.1.0.15/com.ibm.java.v80.doc/docs/jit\\_optimize.html](https://www.ibm.com/support/knowledgecenter/en/SSB23S_1.1.0.15/com.ibm.java.v80.doc/docs/jit_optimize.html), Last accessed on 2019-06-19.
- [8] IBM Knowledge Center. Selectively disabling the JIT or AOT compiler. [https://www.ibm.com/support/knowledgecenter/en/SSYKE2\\_7.0.0/com.ibm.java.zos.70.doc/diag/tools/jitpd\\_disable\\_some.html](https://www.ibm.com/support/knowledgecenter/en/SSYKE2_7.0.0/com.ibm.java.zos.70.doc/diag/tools/jitpd_disable_some.html), Last accessed on 2019-06-19.
- [9] Keith Cooper and Linda Torczon. *Engineering a Compiler: International Student Edition*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.
- [10] Brendan Gregg. How To Measure the Working Set Size on Linux, 2018. <http://www.brendangregg.com/blog/2018-01-17/measure-working-set-size.html>, Last accessed on 2020-01-06.
- [11] LLVM Project. LLVM Compiler Infrastructure. <https://llvm.org/>, Last accessed on 2020-01-06.

