# 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 interaces 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 oppourtunities 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].[1]

Considering JIT compilers may also need perform some of the optimizations found in a static compiler such as common subexpression 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 linked 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 run-time?
- 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 [10] and OMR JitBuilder [11]. In Section 2, 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.

## 2 BACKGROUND

### 2.1 LLVM

LLVM.

[1] An example of a staged, or tiered compilation strategy can 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.

*UNB FCS Directed Studies, Fall 2019, Fredericton, NB, CA*
2020.

### 2.2 JitBuilder

JITBuilder.

## 3 METHODOLOGY

Methodology

### 3.1 Results

## 4 RELATED WORK

Related work.

## 5 FUTURE WORK

Future Work

## 6 SUMMARY

Summary

## 7 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.vm.80.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, 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] LLVM Project . LLVM Compiler Infrastructure . https://llvm.org/, Last accessed on 2020-01-06.

[11] Daryl Maier and Xiaoli Liang. Supercharge a language runtime! In *Proceedings of the 27th Annual International Conference on Computer Science and Software Engineering*, CASCON '17, page 314, USA, 2017. IBM Corp.