A Comparison of API-based JIT Compiler Overhead during Run-time

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 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?[2]
- 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.

[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.

[2] 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].

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

```
define i32 @mul_add(i32 %x, i32 %y, i32 %z) {
    entry:
    %tmp = mul i32 %x, %y
    %tmp2 = add i32 %tmp, %z
    ret i32 %tmp2
}
```

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.

2.1 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 it's 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 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 thats 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.

```
treetop--> istore a
            |
            imul--- isub-------+
            |                  |
            iadd               |
            |                  |
            +-----> iload b <--+
            |                  |
            +-----> iload a <--+
}
```

Listing 2: OMR IR representation for (a+b)*(a-b). Note that the iload nodes are reused [20].

This optimized IR will then be used by the ExecutionEngine when generating code for the target architecture. Before a function is executed by the ExecutionEngine, it first checks if the ObjectCache contains a copy. If the function could not be found, the compiler will generate code and store it in the ObjectCache before execution [16]. It is worth noting that an newer JIT API, called ORCJIT, is also part of the LLVM project. ORCJIT, or On-Request-Compilation JIT, is intended to compliment the MCJIT API – which compiles eagerly, by adding support for lazy, and concurrent compilations [17].

2.2 JitBuilder

JitBuilder is the embeddable framework for interacting with the Eclipse OMR JIT compiler Testarossa [12]. Eclipse OMR is an open-source collection of components for building language runtime environments. Some of OMR's components include a garbage collection framework, a thread library, cross-platform port support, virtual machine building blocks, and a JIT compiler [6, 18]. Much of the infrastructure driving Eclipse OpenJ9, a popular, open-source Java Virtual Machine, links to OMR components.

Through JitBuilder's API, users generate IR upon which optimizations are applied and from which native code is generated [19]. Based on basic-blocks, the IR is arranged into directed acyclic graphic (DAG) structures called trees, composed of nodes which each contain an opcode. OMR IR has several hundred opcodes, where typically with each type operation has a code for each data type: integer, pointer, double, float, vector, etc...[3] The children of these opcode nodes are in turn operands. Trees with side-effects, which cannot be reordered, belong to a list of elements called treetops [20]. Existing nodes can be reused within a given tree, making optimizations such as common subexpression elimination relatively simple (see Listing 2). Though Testarossa supports several levels of optimization ranging from *cold*, with roughly 20 optimizations applied, all the way to *scorching*, where as many as 170 optimizations are applied, as of writing, JitBuilder is locked at the *Warm* optimization level [21, 22].

[3] This can be contrasted with LLVM's roughly 31 opcodes, where the data type code is instead embedded within the instruction.

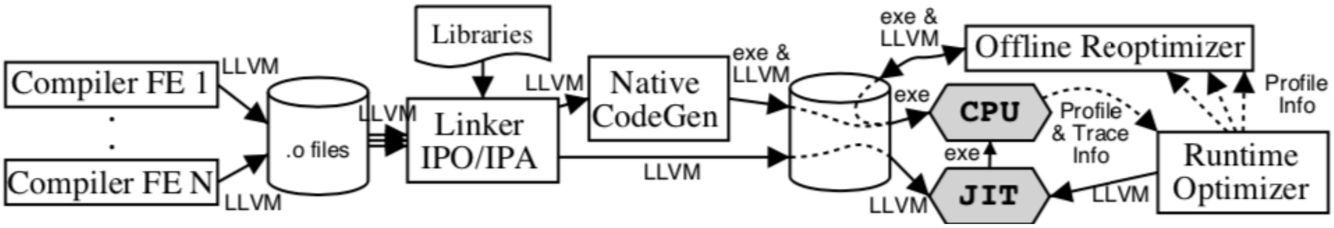

Figure 1: The LLVM Compiler Framework Architecture [13].

3 QUESTIONS AND METHODS

To answer the questions we outlined in the introduction, we wrote three simple programs for each of LLVM MCJIT, JitBuilder, and a straight forward native implementation without a JIT framework. The programs are as follows:

- increment - Calls a single function that adds one to an integer argument.
- recursive-fib - Calls a recursive fibonacci function for *fib(20)*.
- iterative-fib - Calls an iterative fibonacci function for *fib(20)*.

For the JIT implementations, the functions were created using the provided API's, then native code was generated and later used for the actual function call. We built and benchmarked the programs on an x86-64 Linux workstation with 32gb of RAM and an Intel 6 Core (12 thread) processor. Both LLVM, and OMR were built from source from their respective Github repositories [23, 24]. All the source code for this project can be found online through Github[25].

4 RESULTS

4.1 Compilation Time

The first question we looked at was how long each framework took to compile a function 25 times. This time includes running the program from start to finish and thus includes the setup and teardown time of the JIT.

In Table 1 we see that for the increment task JitBuilder's compiled the function 61.5% faster than MCJIT. For the other two tasks, we see that MCJIT was able to compile the functions more quickly than JitBuilder (recursive-fib compiled 16.5% faster, and iterative-fib compiled 40.1% faster). Looking at the JitBuilder code to generate the function (see Listing **??**), we see a much smaller function body compared to the function used to generate IR for LLVM (see Listing 4). There is no clear reason why this disparity exists. One idea is JitBuilder has a lower base overhead than LLVM, but the work required to generate code for more complex IR favours LLVM. We will revisit this question when we look at the call-stack flame graphs.

4.2 Execution Time

Once compilation has completed we turn to measuring how much CPU time is spent actually executing the function. To measure this we compiled the function once and then executed the compiled function 1000 times. For each program, the process was repeated 20 times (see Table 2). Compiling the test programs GCC 5.3 with an optimization level of O3 resulted in extremely low execution

```
bool
IncrementMethod::buildIL()
{
  Return(
    Add(
        Load("value"),
        ConstInt32(1)));
  return true;
}
}
```

Listing 3: Generating JitBuilder IR for the increment program.

```
  static Function *CreateIncrementFunction(
                    Module *M,
                    LLVMContext &c) {
    FunctionType *f = FunctionType::get(Type::getInt32Ty(c),
                        {Type::getInt32Ty(c)},
                        false);

    Function *incrementF = Function::Create(f,
                        Function::ExternalLinkage,
                        "increment",
                        M);

    BasicBlock *BB = BasicBlock::Create(c,
                        "EntryBlock",
                        incrementF);

    Value *One = ConstantInt::get(Type::getInt32Ty(c), 1);

    Argument *ArgX = &*incrementF->arg_begin();
    ArgX->setName("AnArg");

    Value *Sum = BinaryOperator::CreateAdd(ArgX,
                                  One,
                                  "addresult",
                                  BB);

    ReturnInst::Create(c, Sum, BB);
    return incrementF;
  }
}
```

Listing 4: Generating MCJIT IR for the increment program.

times for the native increment program as our function calls were optimized away.

4.3 Max Resident State Set

4.4 Binary File Size

4.5 Considerations

Considering the JIT compiler behind JitBuilder was originally designed for dynamic runtime compilation, it makes sense that it has a lighter footprint than LLVM – both in terms of RSS and linked file size. Unlike LLVM's IR, which is meant for life-long usage, the

Program	LLVM MCJIT			Eclipse OMR JitBuilder		
	mean (ns)	median	std. dev.	mean (ns)	median	std. dev.
increment	1,378,060	1,387,338	39,774	533,581	531,840	4911
recursive-fib	1,548,316	1,547,491	3925	1,854,393	1,850,322	21,754
iterative-fib	2,509,502	2,519,808	60,429	4,192,213	4,191,730	10,419

Table 1: Results of compiling each function 25 times with each JIT framework.

Program	LLVM MCJIT			Eclipse OMR JitBuilder			Native (C++)		
	mean (ns)	median	std. dev.	mean (ns)	median	std. dev.	mean (ns)	median	std. dev.
increment	1,463,310	1,463,032	1660	524,723	524,763	718	0.876	0.875	0.002
recursive-fib	13,674,832	13,664,199	26,548	28,707,381	28,699,013	29,480	31,404,538	31,319,334	142,705
iterative-fib	2,657,528	2,668,792	83,388	4,227,967	4,190,232	102,744	37,379	37,516	583

Table 2: Results of compiling each function and executing generated function 1000 times.

design of OMR's IR does not have the goals in mind. Perhaps this may somewhat limit the overhead of JitBuilder when analyzing and transforming the IR.

5 RELATED WORK

- LLVM IR -> JitBuilder

6 FUTURE WORK

Future Work

7 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.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] 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.

[12] 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.

[13] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[14] LLVM Project . A First Function . http://releases.llvm.org/2.6/docs/tutorial/JITTutorial1.html, Last accessed on 2020-01-06.

[15] LLVM Project . LLVM's Analysis and Transform Passes . https://llvm.org/docs/Passes.html, Last accessed on 2020-01-06.

[16] LLVM Project . MCJIT Design and Implementation . https://llvm.org/docs/MCJITDesignAndImplementation.html, Last accessed on 2020-01-06.

[17] LLVM Project . ORC Design and Implementation . https://llvm.org/docs/ORCv2.html, Last accessed on 2020-01-06.

[18] Matthew Gaudet and Mark Stoodley. Rebuilding an airliner in flight: a retrospective on refactoring IBM testarossa production compiler for Eclipse OMR . pages 24–27, 10 2016.

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

[20] Eclipse OMR . Testarossa's Intermediate Language: An Intro to Trees . https://github.com/eclipse/omr/blob/master/doc/compiler/il/IntroToTrees.md, Last accessed on 2020-01-06.

[21] Ricardo Nabinger Sanchez, Jose Nelson Amaral, Duane Szafron, Marius Pirvu, and Mark Stoodley. Using machines to learn method-specific compilation strategies. In *International Symposium on Code Generation and Optimization (CGO 2011)*, pages 257–266. IEEE, 2011.

[22] Mark Stoodley . Github issue: JitBuilder only uses warm optimization level #1187 . https://github.com/eclipse/omr/issues/1187, Last accessed on 2020-01-06.

[23] Eclipse OMR . [mlir][Linalg] Reimplement and extend getStridesAndOffset . https://github.com/llvm/llvm-project/commit/d67c4cc2eb4ddc450c886598b934c111e721ab0c, Last accessed on 2020-01-06.

[24] Eclipse OMR . Merge pull request #4672 from aviansie-ben/tabortwc-be-fix . https://github.com/eclipse/omr/commit/e6f60a17270955f3312ec85f7c0f53a6e6d01957, Last accessed on 2020-01-06.

[25] Eric Coffin . Report Github Repository . https://github.com/edcoffin/vm-project, Last accessed on 2020-01-06.