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 can provide significant performance gains for applications. Two popular open-source JIT frameworks are MCJIT and JitBuilder. Both of these compilers 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 framework based on the Eclipse OMR compiler and runtime components. In this report we discuss the different approaches these two frameworks employ. In addition, we attempt to measure the overhead of each framework while compiling and then executing a small handful of test programs. We found that while LLVM required a larger memory footprint, in certain cases it was able to generate code more quickly. Furthermore, there were cases where the code LLVM generated offered higher throughput. JitBuilder did provide smaller disk and memory footprints and generated faster code than MCJIT in some cases.

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 profiling which code paths are executed most frequently, JIT compilers can generate code that is optimized for hot-paths [2]. A JIT compiler might also inline entire blocks of code, generate an inline cache to speed up the dispatch of polymorphic method calls [3], or generate hardware specific instructions.

Runtimes for the high-level language 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 want to 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 a runtime's performance by adding a JIT compiler, will have a large engineering task ahead of them. In addition, considering that JIT compilers often need to generate native, or architecture specific code, for multiple platforms, the effort required can 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 processor overhead does the JIT compiler have?
- What is the quality of the generated JIT code and what effect does this have on throughput?
- How is the resident state set (RSS) for the application effected?
- How much larger is the application binary with the inclusion of the JIT framework?
- How configurable is the JIT framework?
- 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 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 the report.

2 BACKGROUND

In this section we will discuss the background of the two JIT compiler frameworks we are interested in: LLVM MCJIT and Eclipse OMR JitBuilder. In particular, we will focus on the motivation for

[1] An example of a staged, or tiered compilation strategy, can be seen with the Testarossa JIT compiler (TRJIT) in the Eclipse 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.

```
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 returing x * y + z [13].

```
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 [19].

each framework, as well as discuss the techniques and features that 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 [12]. 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 (IR) called LLVM IR. The IR, which is initially generated by the front end, can be packaged with the target architecture binary that includes profiling instructions for later runtime compilation (JIT) as well as more aggressive offline optimizations. Several important characteristics of LLVM IR are 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, while operations between registers and memory use 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 application programming interface (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 [14]:

- Analysis Passes: collects 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.

Generated IR can then be used by the ExecutionEngine to generate code for the target architecture. Before a function is executed by the ExecutionEngine, it first checks if the code cache, or ObjectCache, contains a copy. If the function could not be found, the compiler will generate the JIT code and store it in the ObjectCache before execution [15]. 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 both lazy and concurrent compilations [16].

2.2 JitBuilder

JitBuilder is the embeddable framework for interacting with the Eclipse OMR JIT compiler Testarossa (TRJIT) [11]. 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 the TRJIT compiler [6, 17]. Much of the infrastructure driving Eclipse OpenJ9, a popular, open-source Java Virtual Machine, link to OMR components.

Through JitBuilder's API, users generate IR upon which optimizations are applied and from which native code is generated [18]. Based on basic-blocks, the IR is arranged into directed acyclic graphic (DAG) structures called trees, composed of nodes which each contain an opcode. The children of these opcode nodes are in turn operands. While the number of available opcodes is high – there are several hundred, most of the general instructions (add, load, compare) have a specific opcode for each data type: integer, pointer, double, float, vector, etc...[2] Trees with side-effects, which cannot be reordered, belong to a list of elements called treetops [19]. 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 [20, 21].

3 QUESTIONS AND METHODS

To answer the questions we outlined in the introduction, we wrote three simple programs for both LLVM MCJIT, and JitBuilder. For

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

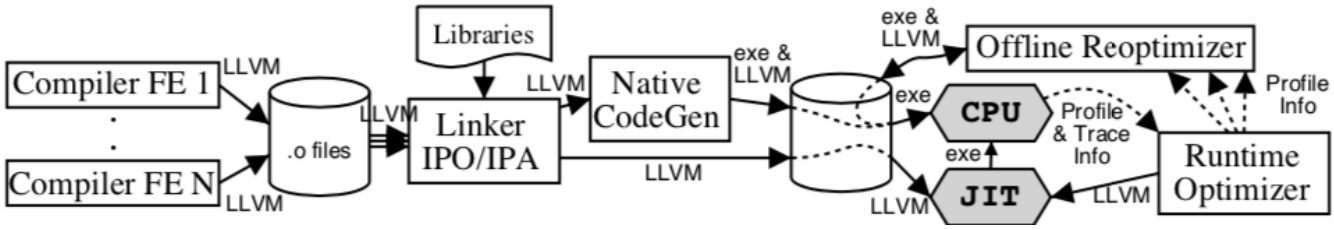

Figure 1: The LLVM Compiler Framework Architecture [12].

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

Listing 3: Generating JitBuilder IR for the increment program.

the sake of comparison, we also wrote the programs using standard C++ without any 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 IR was built using the provided APIs, then native code was generated and later used for the actual function calls. We built and benchmarked the programs on an x86-64 Linux workstation running Ubuntu 19.04 with 32GB of RAM and an Intel i7-8700 (6-core, 12 thread) processor. Both LLVM, and OMR were built from source from their respective GitHub repositories [22, 23]. All the source code for this project can be found online[24].

4 RESULTS

4.1 Compilation Time

The first question we looked at was how long each framework took to compile a function. This time includes running the program from start to finish and includes the setup and teardown time of the JIT. Each program was executed 20 times.

In Table 1 we see that for the increment task JitBuilder compiled the function 61.5% faster than MCJIT. Looking into this further, the initial run to compile the function was roughly 900,000 nanoseconds. Repeated executions made by our benchmarking framework[25] were closer to 500,000 nanoseconds as the instructions for this relatively small example were likely be cached. 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).

4.2 Execution Time

With JIT compilation 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

```
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.

function 1000 times. For each program, the process was repeated 20 times (see Table 2). Aside from the increment program, which favoured JitBuilder, both the recursive and iterative fib functions executed more quickly with LLVM generated code (0.53% faster for recursive, and 0.37% faster for iterative). Compiling the test programs with GCC 5.3, having an optimization level of O3 (large binary, fast code), we saw extremely low execution times for the native increment program as our function calls were essentially optimized away. It is worth noting that the code generated by the LLVM recursive-fib outperformed the native optimized code. Given two of our programs performed more quickly with MCJIT than with JitBuilder, it is interesting to examine the disassembly of the generated code.[3]

Examining the recursive-fib disassembly for JitBuilder (code listing 6), we see it is using the stack for storage. We expect these additional loads and stores, used for passing arguments and return values, are the reason for the performance disparity. The LLVM disassembly (code listing 5) on the other hand, passes arguments using registers. It's also interesting to see the condition for the jump comparison: `jg 0x2` with LLVM and `jl 0x2` in JitBuilder. We would expect the latter to be the better optimization considering

[3]We used GDB to print instructions starting at the function pointer address. The disassembly is using Intel style.

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	536,741	537,103	2620
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 20 times with each JIT framework.

```
(gdb) x/24i $1
  0x7ffff7fcd000 <fib>:        push   rbp
  0x7ffff7fcd001 <fib+1>:      push   r14
  0x7ffff7fcd003 <fib+3>:      push   rbx
  0x7ffff7fcd004 <fib+4>:      cmp    edi,0x2
  0x7ffff7fcd007 <fib+7>:      jg     0x7ffff7fcd013 <fib+19>
  0x7ffff7fcd009 <fib+9>:      mov    eax,0x1
  0x7ffff7fcd00e <fib+14>:     pop    rbx
  0x7ffff7fcd00f <fib+15>:     pop    r14
  0x7ffff7fcd011 <fib+17>:     pop    rbp
  0x7ffff7fcd012 <fib+18>:     ret
  0x7ffff7fcd013 <fib+19>:     mov    ebx,edi
  0x7ffff7fcd015 <fib+21>:     lea    edi,[rbx-0x1]
  0x7ffff7fcd018 <fib+24>:     movabs r14,0x7ffff7fcd000
  0x7ffff7fcd022 <fib+34>:     call   r14
  0x7ffff7fcd025 <fib+37>:     mov    ebp,eax
  0x7ffff7fcd027 <fib+39>:     add    ebx,0xfffffffe
  0x7ffff7fcd02a <fib+42>:     mov    edi,ebx
  0x7ffff7fcd02c <fib+44>:     call   r14
  0x7ffff7fcd02f <fib+47>:     add    eax,ebp
  0x7ffff7fcd031 <fib+49>:     pop    rbx
  0x7ffff7fcd032 <fib+50>:     pop    r14
  0x7ffff7fcd034 <fib+52>:     pop    rbp
  0x7ffff7fcd035 <fib+53>:     ret
```

Listing 5: LLVM MCJIT recursive-fib disassembly (Optimization level 3)

```
(gdb) x/32i $1
  0x7ffff6a81034: sub    rsp,0x18
  0x7ffff6a81038: mov    QWORD PTR [rsp+0x10],rbx
  0x7ffff6a8103d: cmp    edi,0x2
  0x7ffff6a81040: jl     0x7ffff6a8106d
  0x7ffff6a81042: mov    rbx,rdi
  0x7ffff6a81045: lea    edi,[rbx-0x2]
  0x7ffff6a81048: call   0x7ffff6a81034
  0x7ffff6a8104d: mov    rdi,rbx
  0x7ffff6a81050: mov    rbx,rax
  0x7ffff6a81053: sub    edi,0x1
  0x7ffff6a81056: call   0x7ffff6a81034
  0x7ffff6a8105b: xchg   rbx,rax
  0x7ffff6a8105e: mov    rcx,rbx
  0x7ffff6a81061: add    eax,ecx
  0x7ffff6a81063: mov    rbx,QWORD PTR [rsp+0x10]
  0x7ffff6a81068: add    rsp,0x18
  0x7ffff6a8106c: ret
  0x7ffff6a8106d: mov    rax,rdi
  0x7ffff6a81070: mov    rbx,QWORD PTR [rsp+0x10]
  0x7ffff6a81075: add    rsp,0x18
  0x7ffff6a81079: ret
```

Listing 6: JitBuilder recursive-fib disassembly (Optimization level *Warm*)

most times n is greater than 2. The code in the LLVM listing would perform relative jumps more often, while the code in the JitBuilder would flow through the instructions more often.

Looking at the estimated execution time minus JIT compilation for 1000 functions calls (see Table 3), we can see that JitBuilder's generated code outperformed MCJIT both in the iterative-fib programs.[4]

4.3 Binary File Size

Looking at the size of the generated files (see Table 4), the linked LLVM MCJIT programs are roughly six times larger in size than the JitBuilder programs.[5] Part of the explanation for this may have to do with potentially over-linking the LLVM programs. With over 167 possible linkable modules when using LLVM, we found what we think is the minimum number of modules required to use MCJIT. These 8 modules are *core, executionengine, interpreter, passes, mc, mcjit, support, nativecodegen.*

4.4 Resident State Set

The resident state set (RSS) is amount of program memory during execution at any one time. To measure the maximum RSS, we used the `time` utility (`/usr/bin/time -v`). Looking at the results in Table 5, we see that LLVM consistently had the largest RSS. The memory overhead required by LLVM is roughly 4.5 times larger than JitBuilder's requirement. Looking at the number of times inactive pages were reclaimed during execution (minor page faults), we see that LLVM had roughly 4 times the number of inactive pages collected.

4.5 Developer Experience

One additional aspect not typically included when considering overhead is the usability of the frameworks from a developer's perspective. Under this banner we looked at two items: ease of use, and how configurable the frameworks were.

4.5.1 Usability. While ease of use is not likely to be the highest priority when considering which JIT framework to select, research has shown that improved API usability can positively impact developer productivity, while improving code quality and reducing errors [26]. Overall, we found that JitBuilder took less time to integrate, and provided a more intuitive API for generating IR. When considering usability we took note of the following items:

- Both of the frameworks provide public Github repositories [27, 28].
- The instructions for building the frameworks were readily available and simple to follow.
- Linking LLVM took considerable effort as it was unclear which of the 167 objects were required to use MCJIT. After some trial and error, we discovered that LLVM provides a utility, `llvmconfig`, which simplifies this task by generating

[4]In our tests, the compile time plus the execution time of the JIT-ed function was less than the measurement for just the compilation. As noted earlier, we suspect this performance has to do with a high degree of cache locality. We subtracted the mean from Table 1 from the mean from Table 2 without accounting for the variance. This may be incorrect. Unfortunately, time constraints limited our ability to properly benchmark 1000 executions of the JIT-ed code.

[5]The JIT frameworks were compiled without debug symbols.

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	534,192	534,064	869	0.876	0.875	0.002
recursive-fib	13,674,832	13,664,199	26,548	28,707,381	28,699,013	29,480	16,016,325	15,996,370	60,459
iterative-fib	2,657,528	2,668,792	83,388	4,227,967	4,190,232	102,744	0.876	0.875	0.003

Table 2: Results of JIT compiling each function once and executing the generated code 1000 times.

Program	LLVM MCJIT	Eclipse OMR JitBuilder
increment	85,250	0
recursive-fib	12,126,516	26,852,988
iterative-fib	148,026	35,754

Table 3: Estimated time to execute JIT-ed function 1000 times.

Program	LLVM MCJIT	Eclipse OMR JitBuilder	Native (C++)
increment	60,640,216	10,148,608	16,616
recursive-fib	60,671,176	10,149,272	16,600
iterative-fib	60,652,728	10,149,184	16,600

Table 4: Total size in bytes of linked binary test programs.

Program	LLVM MCJIT	Eclipse OMR JitBuilder	Native (C++)
	max RSS (kb)	max RSS (kb)	max RSS (kb)
increment	43,544	9416	1548
recursive-fib	45,472	9768	1532
iterative-fib	44,452	10,044	1684

Table 5: Maximum resident state set (RSS) in kilobytes during execution of a single compilation and execution of the generated function. A minor page fault occurs when the OS reclaims inactive pages during execution. Page size was 4096 bytes.

header and linking arguments based on a list of provided objects[6]

- JitBuilder was relatively simple to integrate, requiring only a single header file and single module to link: `jitbuilder`, where as LLVM required us to include 20 header files.
- We found that writing the IR generators took less time with JitBuilder than with LLVM. The most challenging was writing the iterative-fib generator for LLVM which included an induction value, a Phi node, several basic blocks for before, during and after the loop, condition values, as well as allocation and store pointers.[7].
- In every case the number of lines of code to generate the IR were fewer for JitBuilder. This can be seen in the increment code listing for JitBuilder2, and for LLVM MCJIT1.
- The in-memory objects generated by LLVM MCJIT can contain debug symbols (DWARF format) allowing debugging to take place in GDB 7.0 and higher [29].

4.5.2 Configuration. While the API of LLVM MCJIT may have an overabundance of controls, it does provide an high level of configuration compared to JitBuilder, including which optimization and analysis passes to run, control over the memory structure of the generated code, control over the object cache, and what level of optimization to generate the code at. As mentioned, JitBuilder on the other hand is currently locked at the *Warm* optimization level, limiting the number of optimizations performed on the IR.

[6]Commands for building the programs can be found in `makefiles` in the accompanying source code under `src/<framework>/` [24].

[7]Source code for the LLVM iterative-fibonacci program can be found in the CreateFibFunction within `src/llvm/iterative-fib/fib.cpp`

5 RELATED WORK

Given the overhead involved in fetching, decoding, and executing instructions, interpreters are a common place we see JIT compilers employed. Java Virtual Machines [4, 18] have been employing JIT compilers for over two decades, while the Python world has both Numba, a Python JIT compiler based on LLVM [30, 31], and the PyPy project – a Python interpreter with it's own tracing JIT [32].

JitBuilder has been integrated in several demonstration interpreters [33–35], while work is ongoing to build new runtimes, or to bolster existing runtimes with it's parent project, Eclipse ORM [36].

There is also work being done to allow JitBuilder to ingest LLVM IR [37]. Not only would this allow JitBuilder to bolt onto existing LLVM based projects, it could help provide deeper insight into the quality of the generated code between the two frameworks.

6 FUTURE WORK

To provide a deeper glimpse into the quality of the generated code, the number of test programs could be expanded to cover such scenarios as arrays, large vector operations, and very large functions. It would be interesting to compare the performance of repeated JIT compilations rather than tearing down the JIT infrastructure between calls. The JIT compiler in Eclipse OMR, TRJIT, supports many levels of compilation, as well as support for synchronous, and asynchronous compilation schemes, however, these options have not yet been exposed to JitBuilder. While we believe JitBuilder should be expanded upon to provide such controls, special attention would have to be given to maintaining the simplicity and usability of the API. Finally, testing the compilers under more complex scenarios such as making JIT-to-JIT function calls and dealing with code cache limitations would provide more real world results.

7 SUMMARY

Considering the compiler behind JitBuilder was designed specifically for dynamic run-time compilation, it makes sense that it has a lighter footprint than MCJIT, which shares it's codebase with the static compiler for LLVM. Unlike LLVM's IR, which is meant for life-long usage, the design of OMR's IR does not necessarily have ahead of time, or offline goals in mind. We believe this simplifies the work required when IR interactions are made in JitBuilder and may give it a performance edge as a dynamic compiler. While the code generated by MCJIT did outperform JitBuilder is some cases, we were unable to enable all of the optimizations for JitBuilder.

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.

[12] 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.

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

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

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

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

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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

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

[23] 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.

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

[25] Google . Benchark . https://github.com/google/benchmark, Last accessed on 2020-01-06.

[26] Brad A. Myers and Jeffrey Stylos. Improving api usability. *Commun. ACM*, 59(6):62–69, May 2016.

[27] LLVM Project . The LLVM Compiler Infrastructure . https://github.com/llvm/llvm-project, Last accessed on 2020-01-08.

[28] Eclipse OMR . JitBuilder . https://github.com/eclipse/omr/tree/master/jitbuilder/release, Last accessed on 2020-01-08.

[29] LLVM Project . Debugging JIT-ed Code With GDB . https://llvm.org/docs/DebuggingJITedCode.html, Last accessed on 2020-01-06.

[30] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, New York, NY, USA, 2015. Association for Computing Machinery.

[31] Anaconda . Numba . http://numba.pydata.org/, Last accessed on 2020-01-06.

[32] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. Tracing the meta-level: Pypy's tracing jit compiler. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, ICOOOLPS '09, page 18–25, New York, NY, USA, 2009. Association for Computing Machinery.

[33] Leonardo2718 . Lua Vermelha . https://github.com/Leonardo2718/lua-vermelha, Last accessed on 2020-01-06.

[34] wasmjit-omr . WABT: The WebAssembly Binary Toolkit . https://github.com/wasmjit-omr/wasmjit-omr, Last accessed on 2020-01-06.

[35] b9org . Base9 . https://www.base9.xyz/index.html, Last accessed on 2020-01-06.

[36] IBM . Ruby+OMR JIT Compiler: What's next? . https://developer.ibm.com/code/2017/03/01/ruby-omr-jit-compiler-wh, Last accessed on 2020-01-06.

[37] Nazim Bhuiyan . LLJB . https://github.com/nbhuiyan/lljb, Last accessed on 2020-01-06.