MicroJIT: A Case for Templated Just-in-Time Compilation in Constrained Environments

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

Scott Young
Faculty of Computer Science
University of New Brunswick
Fredericton, NB, Canada
scott.young@unb.ca

Harpreet Kaur
Faculty of Computer Science
University of New Brunswick
Fredericton, NB, Canada
harpreet.bamrah@unb.ca

Julie Brown
Faculty of Computer Science
University of New Brunswick
Fredericton, NB, Canada
julie.brown@unb.ca

Marius Pirvu
Java JIT Compiler Development
IBM Canada
Toronto, ON, Canada
mpirvu@ca.ibm.com

Kenneth B. Kent
Faculty of Computer Science
University of New Brunswick
Fredericton, NB, Canada
ken@unb.ca

ABSTRACT

Modern software libraries and applications often need to be shared across environments that do not always share common architectures. The solution to this code sharing has often been to target managed runtime environments, or High-Level Language Virtual Machines, such as the Java Virtual Machine. These runtimes are often implemented using a process called interpretation. Interpreters are significantly slower than natively executed code. One way that runtimes have addressed this problem is by including a compiler that compiles the input programs into native code at runtime. These Just-in-Time compilers can be categorized into two classes, optimizing and templated. Optimizing compilers take longer to compile and require many supporting data structures to allow for their optimizations, but usually produce (often significantly) faster code. Templated compilers always generate the same code for a given block of input. This template-generated code is faster than interpreting, but usually (often significantly) slower than the code generated by an optimizing compiler. However, these compilers do not require the heavier data structures of optimizing compilers, and produce code much more quickly. For constrained environments, such as those found in containers and Internet of Things devices, where the resources available to the managed runtime are limited, the ability to perform lightweight JIT compilation may be desirable. In this paper, we introduce a templated compiler called MicroJIT into the Eclipse OpenJ9 JVM and compare it to an interpreter-only solution in a resource constrained environment. Although our bytecode coverage is not complete, we demonstrate significant performance improvements over the interpreter for our micro-benchmarks, while at the same time, compiling with less overhead than the default JIT compiler in Eclipse OpenJ9.

CCS CONCEPTS

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

CASCON'20, November 10–13, 2020, Toronto, Canada

KEYWORDS

bytecode, constrained environment, Internet of things, Java virtual machine, Just-in-Time compilation, optimization

ACM Reference Format:
Eric Coffin, Scott Young, Harpreet Kaur, Julie Brown, Marius Pirvu, and Kenneth B. Kent. 2020. MicroJIT: A Case for Templated Just-in-Time Compilation in Constrained Environments. In *Proceedings of 30th Annual International Conference on Computer Science and Software Engineering (CASCON'20).* ACM, New York, NY, USA, 10 pages.

1 INTRODUCTION

With rise of the Internet of Things (IoT), the number of embedded, connected devices has seen a significant surge in growth over the past decade [15]. For instance, farmers are now employing low-powered sensor networks to help monitor crop health, while many electrical utilities are installing smart meters to assist with managing peak demand. With the increase in IoT applications comes an increased need for software development. According to a recent survey, the most critical issues for IoT software developers are security and connectivity [24]. According to the same survey, while C remains the most popular language for developers writing code for the most constrained devices[1], Java is the preeminent language for both gateways, which connect networks of constrained devices, and for server-based applications.

While software written in C may offer the best performance with the lowest overhead, it has its drawbacks: one must manage memory explicitly, there is a lack of memory security, and portability is limited to hosts that match the target architecture and provide the necessary runtime dependencies such as a compatible C Standard library. The value of operating software correctly within heterogeneous contexts should not be underestimated: the same code may need to execute on a workstation and a server[2], on x86 and ARM platforms, and its lifetime may span decades, making it likely that the underlying platform will change.

[1] In the survey, Java was ranked number four among most popular languages used for constrained devices [24].
[2] The platform-spanning capabilities that the JVM provides can also aid in the architecture of systems: the same code could be reused between clients and servers potentially reducing development effort.

Java, along with its underlying Java Virtual Machine (JVM), has addressed these challenges: memory is automatically managed and not directly accessible—increasing both safety and security, while portability is available to any host with a compatible JVM. That said, the memory safety, security and portability provided by the JVM adds additional overhead cost to the application. Given that Java is the programming language of choice for majority of the industry [41], for many applications, it is evident that this overhead is worth paying.

Given the importance placed on security for IoT applications, the JVM could potentially provide benefits for constrained applications too. Any constrained system executing JVM-based workloads will be concerned with overhead due to many sources, such as dynamic class loading, interpretation of the application, mapping of program state to memory, initial overhead associated with the JIT compiler to compile methods into native instructions [2, 10], safety checks of certain memory access operations during runtime (such as null-reference and index out-of-bounds checks) and automatic memory management (which requires a garbage collector) [25].

In this work, we will consider environments that have the resources to run a modern, Java SE compatible JVM, yet are constrained enough to pay careful attention to these sources of overhead, particularly, to the JIT compilation. The JIT compiler in Eclipse OpenJ9—Testarossa (TRJIT)—is a highly-tunable, method-based, optimizing compiler. While TRJIT supports fast compilation, generating non-optimized code, it requires an intermediary stage where an intermediate language (IL) is generated. It is possible that the IL generation phase may introduce too much overhead for constrained environments where low-optimized code is sufficient. One approach to solve this issue could be to add a second, lighter-weight JIT compiler to the virtual machine, which could be used in place of TRJIT. Instead of generating an IL, this JIT compiler would translate bytecodes using predefined machine code templates, which would be stored and could be executed later. By eliminating the IL phase, we expect that JIT compilation will have a smaller footprint and will take less time to perform[3].

We propose adding this lightweight JIT compiler to Eclipse OpenJ9, which could be used instead of, or alongside the regular JIT compiler for constrained environments. In the spirit of the original MicroJIT [42], this compiler will be template-based with the goal of generating native methods as efficiently and quickly as possible. It should be noted that while our previous work focused on porting the original MicroJIT from IBM J9 to Eclipse OpenJ9 [4], we have since realized that this would entail a near-complete rewrite to achieve the required compatibility. With that said, we will continue using the moniker MicroJIT for this new, template-based JIT compiler. To guide the order of bytecode-template implementation, we will record the execution frequency of bytecodes in several standard benchmarks and to establish a baseline, we will measure the overhead of the interpreter operating without any JIT compiler. Then, we will measure the overhead associated with TRJIT and MicroJIT and compare the throughput of several applications while running in interpreter-only mode against the interpreter with MicroJIT. Finally, we will analyze the results to identify the window where template-based JIT begins to outperform interpretation, but where optimizing compilation overtakes template-based compilation.

This paper reads as follows. Section 2 provides an overview of the background and Section 3 discusses the related work. In Section 4, we focus on the design of MicroJIT and how we integrated it into Eclipse OpenJ9. Section 5 details our evaluation and results. Section 6 outlines the future work, and finally, Section 7 concludes the work, summarizing our findings and offering recommendations.

2 BACKGROUND

In this section we discuss the following background topics: the Java Virtual Machine, JIT compilation, Eclipse OpenJ9, and finally constrained devices.

2.1 Java Virtual Machine

In the early 2000s, type-safe, runtime-based languages such as Java and C#, rose to prominence in the realm of enterprise applications [40, 41]. The designs of enterprise applications, often with complex domain models, rules and requirements are aided by these high-level, object-oriented languages and the virtual machines that execute them. Rather than compiling directly to executable code, applications written in these languages are designed to be portable, compiling to intermediary formats designed to be interpreted on any machine providing a compatible runtime environment. Typically, the programs written in the Java language compile to class files[4] and execute on Java Virtual Machines. For an in-depth discussion of the Java language, please refer to the Java Language Specification [14].

This focus on portability popularized the phrase "write once, run anywhere" [5]. To ensure cross-platform compatibility, the JVM must adhere to the Java Virtual Machine Specification [29], which describes the program layout, linking and loading, program verification, stack machine descriptions, class file format and bytecode listing. In addition to portability, Java prevents the application from explicitly managing or interacting with the underlying memory. While this limitation may preclude developers from choosing Java to solve lower-level tasks, it works well for high-level applications. By removing explicit memory management, something that takes increased care as application complexity grows, developer productivity can improve and the chances of inadvertently introducing memory errors decrease.

A runtime that implements the Java Virtual Machine Specification and provides the required Application Program Interfaces (APIs) should be able to run any compatible Java application. For maximum performance, these virtual machines are typically written using system-level languages, such as C, C++ and assembly. The underlying host supports these virtual machines: process controls, virtual memory, concurrency, exceptions and I/O must map from the JVM to the underlying operating system (OS) and, in turn, to the underlying Instruction Set Architecture (ISA) [39]. With this view in mind, interactions are made from the application to the underlying OS through the Java API libraries, such as when calling `System.out.println` and having ASCII characters print to the command-line. Alternatively, interactions between the JVM and the underlying OS can be made through the Application Binary

[3] In terms of throughput or work performed during a period of time, we expect that the code generated by the lightweight compiler will be faster than the interpreter, but an order of magnitude slower than the code generated by TRJIT.

[4] Java programs are compiled using the Java compiler or `javac`.

Interface (ABI), such as when making a system call such as *fork* on Linux to spawn a new thread.

Some examples of production-grade JVMs include Eclipse OpenJ9 [33], Oracle's HotSpot [8] and Azul Zing [22]. It is worth noting that the JVM has become an essential platform for languages beyond Java. Languages such as Scala, Kotlin and Clojure are all popular languages that compile to JVM compatible meta-data and bytecode. Class files, the format of which is defined by the JVM specification, contain the program structure, behaviour, data structures and various metadata required by the JVM for the program to execute. The executable code, contained in parameterized functional units called *methods*, is defined by the series of bytecode instructions. Each bytecode is one or more bytes in length, with the first byte containing the instruction opcode and the following bytes containing the instruction arguments. Data types include primitives such as integers, doubles and booleans, as well as object reference types. The JVM provides a stack-based machine for the application to operate within. Each application thread has a stack, which in turn contains stack frames. Each stack frame contains an operand stack, a program counter and a local storage array. The exact size or shape of the stack frame is described in the meta-data contained in the class file. Operations described by the bytecodes affect the operand stack, which maintains the state of the method.

During execution, an application can store values in two places: in the current stack frame, or on the heap. For values of known sizes, such as primitives, the stack is an ideal location for allocation as the memory will be automatically released when the stack frame is popped. Any values that are dynamic in size, or non-local, will be allocated on the program heap. As this memory cannot be explicitly freed, over time, as the program continues to execute, the heap will continue to fill. Garbage collection is the process by which the objects in the heap that are no longer referenced are freed [25]. For a collector to be viable, it must satisfy the requirement that all garbage will eventually be "collected".

2.2 Just-in-Time Compilation

Although Java applications were at one time entirely interpreted, advances in Just-in-Time (JIT) compilation have improved performance by orders of magnitude, allowing them to be much closer to that of their Ahead-of-Time (AOT) compiled counterparts written in C or C++ [26]. In addition to portability and safety, the high-performance garbage collectors and JIT compilers found in production-grade JVMs have helped place Java at the top of languages used for server-based workloads [41].

According to Rau [36], application binaries can be divided into three groups: those composed of directly executable machine instructions, those composed of higher-level instructions that must be parsed and interpreted before execution and those composed of directly interpretable instructions. While the latter two groups offer opportunities for portability and memory safety, the performance penalty incurred by interpretation may preclude them from specific workloads. This work will focus on the final group, i.e., programs composed of directly interpretable instructions.

While interpreters should be designed to be efficient—minimizing the number of indirect branches [11, 39]—at their simplest, they involve a cycle of fetching a single instruction, decoding that instruction, and then dispatching the instruction for execution using native instructions [39]. Even when the same instruction is executed repeatedly, this same cycle is performed. One way to improve the performance of such workloads is to add a runtime compiler to the execution framework. By compiling, or translating blocks of instructions into native instructions, and then later executing those generated blocks instead of interpreting them, significant performance gains may be achieved. This process, known as Just-in-Time compilation, is often combined with profiling and/or analysis to optimize the generated code [2, 39].

Considering JIT compilation as an expensive activity, in order to allow for the continued execution of the workload, Just-in-Time compilation is often performed selectively on only the most frequently executed blocks of code [28]. As described by Hansen in his work on Adaptive Fortran [17], one way to perform selective compilation is to associate with each code block a counter containing an invocation threshold value. Each time the code block is executed, the counter is decremented. When the counter reaches zero, the block can be compiled. For a block to be JIT-compiled means that for a source block X, there is a generated block of machine instructions, Y, that lives in a code storage area called the code cache [18, 39]. During execution, when the execution engine encounters a call to block X, it performs a check to see if Y exists in the code cache, and if it does, the interpreter transitions to the native code instead. This invocation threshold should be made tuneable for the user, as setting a low threshold will result in more JIT compilation during the startup of the application, which may be undesirable for some workloads. Within the context of Java, we will use entire methods as our code block or unit of compilation (see Figure 1).

JIT compilation can be viewed on a continuum: on one end, there is unoptimized code that is fast to generate but slow to execute. In contrast, on the other end, there is optimized code that is slow to generate but fast to execute. To lessen the impact on the startup time of an application, one could perform initial compilation with little or no optimization, that is, generating slow code quickly. As a code block continues to execute, higher thresholds may be reached, potentially triggering more expensive, optimized recompilations [20]. For JIT compilers with multiple optimization levels, it may prove useful to allow a user to specify heuristics per optimization level.

In order to aid in performing optimizations, it is typical for a JIT compiler to build an intermediate representation of the code to analyze and transform. Some of the optimizations we see for JVM-based optimizing JIT compilers are similar to those commonly found in AOT compilers [7, 27, 39], such as dead-code elimination, constant propagation, constant folding, strength reduction, code hoisting, loop unrolling, inline substitution and peephole optimizations.

2.3 Eclipse OpenJ9

This work will expand upon the JVM implementation of Eclipse OpenJ9 for Java 8 [33]. In particular, we will focus on its implementation for Linux on the x86-64 architecture. This open-source project, which implements the OpenJDK specification and is available under the AdoptOpenJDK project [1], is designed for low overhead, quick startup and high throughput. Also, OpenJ9, which originated from

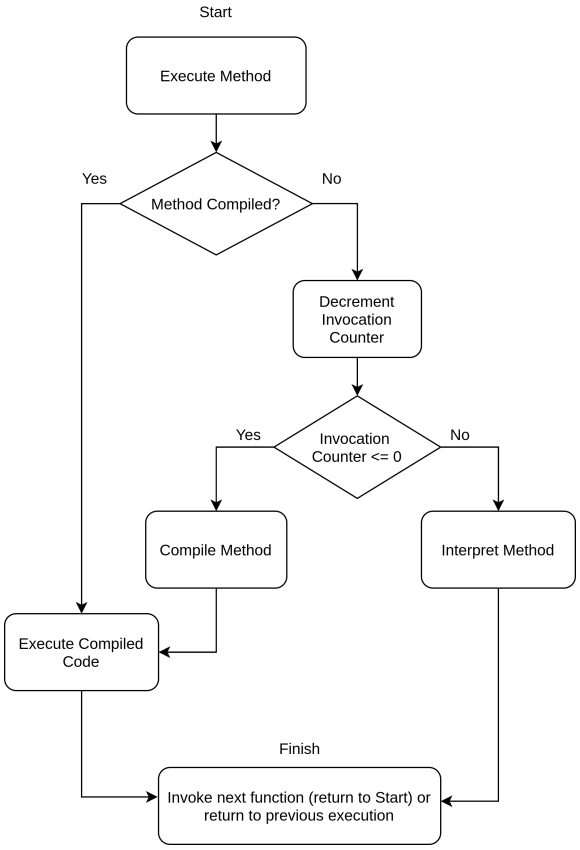


Figure 1: When a method is to be executed, the execution engine first checks if the method has already been JIT-compiled. In this example, compilation and, in turn, execution occurs synchronously, potentially occurring on the currently executing application thread.

IBM's J9 JVM, is built upon the open-source runtime component framework named Eclipse OMR. The OMR framework allows an existing runtime to integrate the production-grade components such as garbage collection, JIT compiler, an API for quickly utilizing the JIT compiler, named JitBuilder, cross-platform support for threads and signals, as well as threading support [13, 30]. Much of the infrastructure driving Eclipse OpenJ9 links to OMR components.

The JIT compiler found in this project, and thus in OpenJ9, is named Testarossa (TR) [43]. In TR, the compilation is method-based and triggered using invocation thresholds for regular methods as well as methods containing loops. TR supports several levels of optimization ranging from the initial optimization level *cold*, with roughly 20 optimizations applied, through *warm*, then *hot* and *very-hot* levels, all the way to the highest level—*scorching*, where as many as 170 optimizations can be applied [34, 37]. There is another compilation level named *noOpt*, which applies no optimizations and may be used to improve application startup time in large applications [34].

Command-line arguments for TR can be passed to the JVM with the `-Xjit` option group [32]. Some options significant to this work are listed as follows:

- `count` - The invocation threshold for standard methods. The default value is 3000.
- `bcLimit` - The bytecode limit size for methods to compile in bytes.
- `codetotal` - Available memory for generated code in KB.
- `limit` - A debugging option to list the methods that TR should include and at what compilation level it should compile them.
- `exclude` - A debugging option to list the methods that TR should exclude.
- `breakOnEntry` - A debugging option generating a breakpoint instruction (`int3` on x86-64) at start of the generated code.

During compilation, fewer resources are available on the platform for executing workload. For certain constrained environments, the overhead of the TR JIT compilation may be too high to be effective. In this work, we will add a second, lighter-weight JIT compiler, MicroJIT, to Eclipse OpenJ9.

2.4 Constrained Devices

In this work, we consider *constrained devices* to be any computing device that has limited processing capability compared to other devices performing work. We use the term *constrained environment* to denote a context within which work is done on a constrained device. Thus, this term can encapsulate embedded systems, connected IoT devices, and even containers that are running in the cloud. This limited capability may be the result of several things: having limited CPU cores and processing speed, having limited memory, maintaining optimal power efficiency, having limited I/O bandwidth, or having limited connectivity [19]. The type of workload performed on the device may also be indicative of a constrained environment.

As the market for IoT continues to increase, the number of these constrained devices is growing substantially. Indeed, IoT devices are being installed for nearly every conceivable scenario where data can help inform decisions. With this increase in devices, comes the increased need for software and thus software developers. Meeting this demand in a timely fashion with secure and robust software will be a significant challenge for the industry. Revisiting the Eclipse IoT developer survey [24] from 2019, we see that although C is the most popular language for most of the constrained devices, Java is in the top five. Furthermore, developers use Java over any other language for less constrained devices—gateways and servers. With the benefits of Java and the JVM in mind, it is conceivable that interest in using them for running workloads on constrained devices will continue to increase. For many of these workloads, performance will be a crucial factor, and as such, JIT compilation will play an essential role.

3 RELATED WORK

Different solutions have been proposed in the past in this direction. One such work was the experimental support for WebAssembly—a new runtime and compilation target for the web—in V8 [47]. This implementation used the TurboFan compiler—V8's powerful

optimizing compiler—and much of the existing JavaScript virtual machine infrastructure. After roughly a year, the V8 team launched a new JavaScript execution pipeline for V8 v5.9 [45], which led to improvements in performance and memory consumption of JavaScript applications. This pipeline used Ignition for interpretation and TurboFan for compilation. Liftoff is a baseline compiler for WebAssembly, which is included in V8 v6.9 and now enabled by default on desktop systems [16]. It reduces the startup times of WebAssembly applications significantly by adding another compilation tier. The new compilation pipeline with Liftoff is much simpler compared to the existing compilation pipeline with TurboFan. Even if the existing compilation process is straightforward, it still consumes considerable time and memory. So, Liftoff is able to generate code much faster than TurboFan.

SpiderMonkey—Mozilla's JavaScript runtime—uses IonMonkey for JIT optimization [6]. It applies various strategies to optimize operations, such as property accesses and function calls. Shudo et al. [38] developed a Java Just-in-Time compiler involving low compilation and development costs. The optimization methods implemented in the compiler included instruction folding, exception handling with signals and code patching. Another work by Iliasov [21] demonstrated that dynamic code generation from templates created using a C compiler can be employed to build a simple, portable JIT compiler. At the same time, a template-based compiler requires more memory than an interpreter and also, it implies certain limitations on the instruction sets.

Sogaro et al. [42] investigated whether using two different JIT compilers in the same JVM can improve startup times. They concluded that integrating MicroJIT—a lightweight JIT system—with the default J9 JIT, could improve startup times in some configurations of the JVM. Later, another work was carried out by Coffin et al. [4] by incorporating MicroJIT into OpenJ9 to offer similar improvements to startup times, while offering applications in resource constrained environments a lightweight JIT alternative. Some changes were proposed to the earlier implementation of MicroJIT to improve its flexibility and performance, such as porting MicroJIT to the 64-bit x86 architecture, extending bytecode support, added support for asynchronous compilation and profiling to the generated code.

In our current work, an effort has been made to improve the throughput of certain workloads in constrained environments running with MicroJIT versus running in interpreter-only mode. While trying to port the project, we realized that it could involve more effort and risk than rewriting the compiler from scratch mainly because the mechanism in the previous MicroJIT for returning to the interpreter when an unsupported bytecode was encountered would not work in Eclipse OpenJ9 and also, the shape of the stackframe the original MicroJIT maintained was not compatible with Eclipse OpenJ9. While these issues signalled that a rewrite of the previous MicroJIT was necessary, we did borrow several aspects from its design such as continuing to use methods as the unit of compilation for MicroJIT, iterating or walking the bytecodes, copying a snippet of machine code to the code cache for each bytecode it encountered while generating machine code for the body of a method, and use of `extra2` field similar to the previous MicroJIT.

An alternative approach to reduce JVM overhead from JIT compilation is to utilize AOT-compiled code. Considering that the Java Virtual Machine specification states that linking and loading must happen dynamically [14], the quality of AOT-compiled code will be limited as the compiler must assume that all references are unresolved. The quality of AOT-compiled code can be improved by storing dynamically generated code from the JIT compiler and metadata in an offline store, or cache. This cache can improve startup time and performance over the interpreter, potentially leading to less reliance on the JIT compiler for some workloads. OpenJ9 offers this improvement through the use of the shared class cache, a memory-mapped file, enabled automatically through the use of the `-Xshareclasses` command-line option. The cache can be configured to be persistent and shared between multiple instances of the JVM [9]. For long-lived environments capable of persisting runtime generated code between executions of the Java application, this approach provides a very efficient mechanism for reducing startup time [35]. On the other hand, for short-lived environments, such as those running ephemeral containers where applications will only execute once, this approach may not easily offer a viable solution [46], although options are available to pre-warm an image during the building of a Docker Image [23].

4 DESIGN

MicroJIT is a template-based JIT compiler for Eclipse OpenJ9 designed to generate code quickly with low overhead. Using methods as its unit of compilation, MicroJIT translates the bytecodes of a Java method-body into a block of generated binary instructions stored for later execution in a code cache. The compilation is performed selectively through the use of invocation counters. Once the number of times a method has been interpreted reaches a specified threshold, the method will be passed to MicroJIT for compilation. If the method is able to be compiled by MicroJIT, later, when the method is invoked again by the interpreter, execution will instead transition to the JITed code.

4.1 Integrating MicroJIT and Eclipse OpenJ9

While MicroJIT may be viewed as an independent module, it is closely integrated with Testarossa (TR). This coupling allows us to take advantage of numerous facilities provided by TR, including:

- **Code Cache Management**: The code cache management facility provides us with a convenient, cross-platform mechanism for allocating and managing executable memory. While we could have directly used the system call `mmap` to provide our process with an executable memory space to copy our templates, this would have limited us to the Linux platform. By leveraging the Code Cache Manager API, MicroJIT can interact indirectly with the memory systems of other operating systems as well.
- **Asynchronous Compilation**: Rather than compiling directly in the same thread as the executing method, which would delay the execution of the workload—especially for large methods—a method is placed on a queue for later compilation, allowing the interpreter to continue interpreting the method. The compilation queue allows increased parallelism, as we can now perform compilation on multiple threads and also to prioritize some methods over others. By utilizing the existing compilation process up to but not including the IL-phase, we gain support for asynchronous compilation.

- **Debugging Utilities**: TR provides command-line options to limit compilation to particular methods. TR's ability to log the generated code in a human-readable format was also valuable when designing the MicroJIT code-generator. By sharing much of the pre-compilation and post-compilation code paths with TR, MicroJIT benefits from these debugging and tracing facilities.
- **Bytecode Iterator**: The Bytecode Iterator class, which implements the iterator pattern [12], provides a convenient mechanism for iterating through a method's bytecode stream. The class saved us the effort of parsing Java bytecode instructions and operands while providing us with their associated mnemonics for both programming and debugging[5].
- **Exceptions**: Finally, the close integration of MicroJIT with TR provides us with cross-platform support for interrupt handling. Through the Port library, provided through Eclipse OMR, signal handlers are registered for JITed code. In the event our JITed instructions generate an exception, for example, when attempting to divide by zero, the registered handler will receive the signal, and then begin the process of unwinding the stack from the JITed frame to find the appropriate Java exception handler.

In order to enable and configure MicroJIT within Eclipse OpenJ9, we added the following command-line options to the `-Xjit` top-level option:

- `mjitEnabled` - Set to 1 to enable MicroJIT.
- `mjitCount` - The number of invocations a method requires before triggering a compilation.

4.2 Architecture

In the following sections we discuss how we select and generate code, as well as considerations for our target platform—Linux on x86-64.

4.2.1 Selective Compilation. The pre-compilation phase begins when the interpreter executes a method that has not been JIT-compiled. The invocation counters are adjusted, thresholds are checked and compilation is triggered when the counters have reached zero. MicroJIT adds a second field, `extra2`, to each method metadata structure `J9Method`. This field performs a role similar to the `extra` field, which is used by TR to store the invocation count, the address of the JIT-compiled method, or special values indicating that compilation should not be attempted again. When compilation is triggered, and asynchronous compilation is enabled, the method is placed in the compilation queue. The compilation thread then dequeues a pending method compilation and continues the pre-compilation phase during which metadata structures are populated, and various compiler options are initialized. Once the pre-compilation completes, the compilation proceeds to MicroJIT.

4.2.2 Code Generation. When MicroJIT compilation begins, a 1024-byte segment of memory is first allocated through the code cache manager. Currently, if the number of generated bytes exceeds this, the compilation will fail, and the segment will be freed through the code cache manager. Next, we inspect the incoming parameters to generate code for populating the stack frame and the local storage area, and for ensuring later root-set compatibility with garbage collection. While the slots in our operand stack are each eight bytes, i.e., large enough to satisfy any primitive datatype we will encounter, for certain `wide` operations involving floats and doubles, we use two slots. Likewise, following the Java specification, for a local array, we allocate two slots for both float and double types in order to simplify compatibility.

The initial machine code generated by MicroJIT contains the preprologue followed by the prologue. The preprologue contains code to check for stack overflow as well as to potentially relinquish control to the interpreter to perform any necessary JVM processes. The prologue contains code to push the previous base pointer and the preserved registers onto the stack and then sets up the new base and stack pointers. Instructions are then generated to copy the incoming parameters to the local array, after which the next step is to generate the body of the method.

While iterating the method bytecodes, if an unsupported bytecode is encountered, the compilation is aborted, the code cache memory is freed, and a value is stored in the `extra2` field to signal the method failed and should not be compiled by MicroJIT again. On the other hand, if all the bytecodes of the method body were supported, the code buffer will now contain the machine code templates for each bytecode instruction. Branching is supported through the use of two structures: a table mapping each bytecode index to the generated code address, and a jump table containing an entry for each branch instruction. Each jump table entry contains the branch or jump's target bytecode index and the address in the generated code to target with a patch operation after the template is copied. After the bytecodes have been iterated, we iterate each jump table entry finding the generated address for the target bytecode in the bytecode index table and then patching the generated branch or jump address with it. After the body has been generated and patched, the epilogue instructions are generated, which clean up the stack and restore the preserved registers. Finally, the address of the preprologue is written to the `extra2` field within the method metadata structure. Later, when the method is invoked by the interpreter, this field is checked, and if it contains a valid program counter address, execution then transitions to the JITed code.

4.2.3 Platform. Our first target platform for MicroJIT is Linux running on the 64-bit x86 architecture. The choice of this platform stems from our previous work attempting to port MicroJIT to Linux from Windows while extending its instructions from 32-bit to 64-bit. With the choice to rewrite the compiler, we maintained the target platform. Values are passed to and from the generated code via registers according to the following calling convention defined by Eclipse OpenJ9:

- `RAX, RSI, RDX, RCX` - Argument registers for the first four integer method parameters where the first argument is found in `RAX`. Other parameters will be found in the caller stack frame.
- `XMM0-XMM7` - Floating point method parameters.

[5]The iterator pattern is a design pattern for iterating over a collection of elements. The methods provided are `first()`, to get the first element, `next()`, to get the next element of the iterator, and `hasNext()`, returning a boolean value if there is another element to iterate over.

```
template_start iAddTemplate
mov r11, [r10] ; pop first value off java stack
add r10, 8 ; reduce stack size by 1 slot
mov r12, [r10] ; copy second value to the value reg
add r11, r12 ; add the values
mov [r10], r11 ; write the accumulator to the stack
template_end iAddTemplate
```

Listing 1: Bytecode x86-64 template for the `iadd` bytecode.

- `EAX, RAX, XMM0` - Return value registers; `EAX` is used for 32-bit integers, `RAX` for 64-bit integers and `XMM0` for float or double values.
- The return address will be already on the stack. After the epilogue executes, this value will be used by the action to return to the previous frame.

Within the generated code, similar to the JVM specification, we use a memory-based operand stack for state, and a local array for storage. We also provide a side-table for internal mapping between the bytecode and generated code, which is used for patching branch and jump instructions. We use the following x86-64 registers within the generated code:

- `RSP`: Base pointer for the Java stack frame
- `R10`: Stack pointer for the Java stack
- `R11`: Stores the accumulator or stores a pointer to an object
- `R12`: Stores any value which will act on the accumulator, stores the value to be written to an object field or stores the value read from an object field
- `R13`: Holds addresses for absolute addressing; used when loading references or fields
- `R14`: Holds a pointer to the start of the local array
- `R15`: Stores values loaded from memory for storing on the stack

4.3 Bytecodes

The majority of the bytecodes we implemented map to a unique template written in NASM-style assembler, although several bytecodes, notably `load`, `store` and `ret` were handled generically. Listing 1 shows the assembler for the `iadd` template. We use the `template_start` and `template_end` macros to simplify calculating the size of the template. During code-generation, as we iterate through a method's bytecode stream, we simply copy these templates into the allocated code cache segment. For those bytecodes that require index-based addressing, we write placeholder bytes and patch them after the body has been completely generated. With the assistance of side-tables, a similar mechanism is used for branching as well as for the `goto` instruction. The operands of these bytecodes specify signed offsets from the current bytecode, though we translate them into indexes from 0.

4.3.1 Implementation Strategy. At the time of writing, MicroJIT does not provide full bytecode coverage. As mentioned, when we encounter an unsupported bytecode, we prevent further attempts at compiling the method with MicroJIT. While we have enough bytecodes implemented to compile the Fibonacci programs described in the Results section, MicroJIT lacks support for many important bytecodes including `invokevirtual`, `new`, `invokespecial`, `newarray` and `athrow`. To guide our implementation, and to track our progress, we test our bytecode coverage, against the DaCapo benchmark suite [3] (see Table 1 for results with `avrora`). Across all the benchmarks, `invokevirtual` and `invokespecial` are the most common unsupported bytecodes.

Bytecode	Count	% of Unsupported	% of Total
JBinvokevirtual	2518	18.53	5.25
JBputfield	1647	12.12	3.44
JBinvokespecial	1530	11.26	3.19
JBifeq	773	5.69	1.61
JBnewdup	632	4.65	1.32
JBldc	575	4.23	1.20
JBaconstnull	366	2.69	0.76
JBathrow	351	2.58	0.73
JBifnull	338	2.48	0.70
JBnop	267	1.96	0.55
Total Bytecodes	47,877		
Unsupported	13,585		

Table 1: The bytecode frequency for DaCapo is used to help guide implementation choices. Listed are the 10 most used unsupported bytecodes for `avrora` with `mjitCount=20`.

4.3.2 Testing. To aid in the development of MicroJIT, we designed a regression test framework. This framework allows us to perform assertions on the results of Java test methods while at the same time confirming that they were compiled by MicroJIT. These tests are then executed with the aid of the JUnit framework [44] on Eclipse OpenJ9. We ensure our test methods are executed by providing the same invocation threshold to both the test framework and to the JVM. When the tests have completed, we have the expected results from JUnit, and we scan the compiler log for special messages signalling that compilation was completed for each particular method. A failure could thus be caused by one of two things: either a method was not compiled by MicroJIT, which would indicate a failure within the code generator, or the result was incorrect, indicating an issue with a bytecode template. The test framework is designed to execute automatically on a continuous integration server before a branch is merged into the mainline, or trunk.

5 RESULTS

We tested the performance of MicroJIT with bespoke micro-benchmarks that calculated the Fibonacci series iteratively and recursively using static methods, for which we provided full bytecode coverage. `IterativeFib.fib(I)I`, which compiles to 37 bytecodes, computes the series in a single method call using a for loop and a temporary variable, while `RecursiveFib.fib(I)I` compiles to 21 bytecodes, and computes through repeated calls to itself. The micro-benchmarks were run on an embedded board with an Intel Atom 1.44 GHZ 4-core processor, 4GB of RAM, running Debian 5.6.7-1.

First we will look at the time spent compiling the methods with MicroJIT, and with TR with the compilation levels of `noOpt` (TR-noOpt) and `cold` (TR-cold) respectively. The times, shown in microseconds, are based on 200 fresh executions of the Java test programs for each of the three compiler settings, with the 20 lowest and

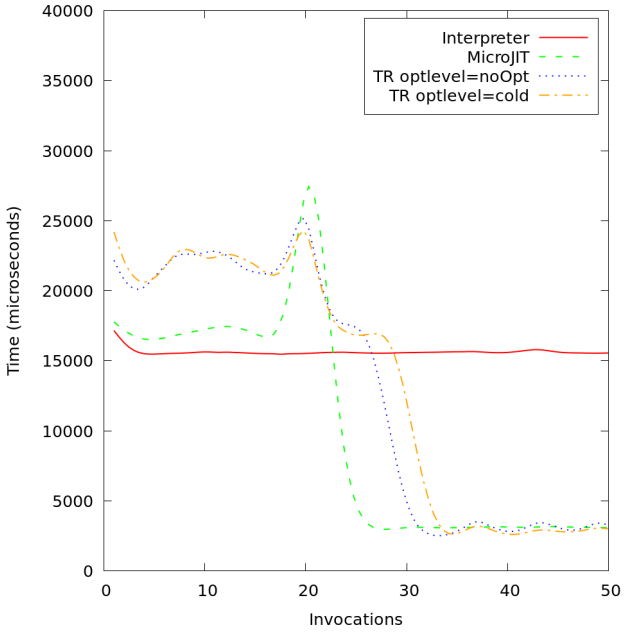
20 highest results discarded to help eliminate outliers. Also, using the `limitFile` option, each benchmark was limited to compiling the single bespoke method. Table 2 shows that by eliminating the IL phase, MicroJIT was able to compile the methods roughly 2.3 times faster than TR-noOpt, with `RecursiveFib` almost 5 times faster than TR-cold level. Considering that we are copying machine code templates instead of building and transforming an intermediate form, this is the expected result. Table 3 shows the memory in KB used by JIT compilers to compile the methods in the same experiment[6]. We see that MicroJIT uses a fraction of the memory that either TR-noOpt or TR-cold use: it is able to compile the micro-benchmark methods with just a single 64 KB segment. One result that stands out is TR-noOpt requiring more memory than TR-cold for `IterativeFib`, but less for `RecursiveFib`. Looking at the optimization logs from TR, we see that the generated code size is 147 bytes for TR-noOpt and 141 bytes for TR-cold. We also see that in the post-optimization IL-trees, TR-noOpt, which had only a single tree-simplification optimization, has 44 nodes, while TR-cold, which had several optimizations performed, has 38 nodes. Comparing this to the `RecursiveFib` results, we see that the generated code size is 103 bytes for TR-noOpt and 265 bytes for TR-cold. Also, looking at the post-optimization IL-trees, we see that TR-noOpt has 23 nodes, while TR-cold, which had several optimizations performed, including inlining, has 103 nodes. One possible explanation for the larger footprint when compiling `IterativeFib` at the `noOpt` level is the overhead associated with the larger IL-trees.

Figure 2a shows the execution time of calculating the 30th Fibonacci number using `IterativeFib` from 100 fresh executions of the Java program. With each compiler using 20 as its invocation threshold, we see that MicroJIT completes its compilation first and thus sees the earliest improvement in throughput. Figure 2b shows similar results for the recursive method, however we note that the invocation count is reached earlier due to recursion. In both figures, we see that the baseline performance of interpreter remains fairly static. The interpreter runs were executed in interpreter-only mode without TR (`-Xint`), meaning that no profiling data was gathered. On the other hand, for the TR-specific runs—TR-noOpt and TR-cold—the interpreter did collect profile information, which is then used later to inform the JIT compilations.

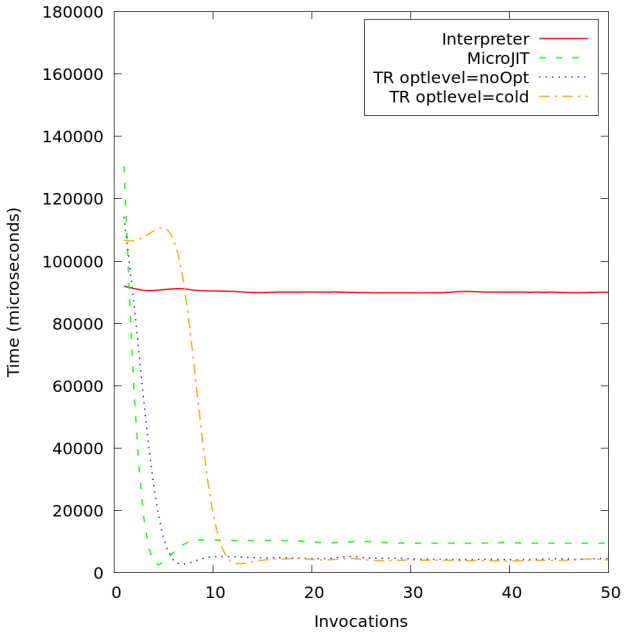
Finally, Tables 4 and 5 show the throughput of over 1 million invocations for `IterativeFib` and `RecursiveFib` respectively. The experiment was run 100 times. The % column shows the percentage of time spent relative to the interpreter. In Table 4, we see that MicroJIT is able to complete 1 million calculations in 1.38 seconds, or 9.55% of the time it took for the interpreter. As expected both TR-noOpt and TR-cold are able to perform the same task significantly faster, with TR-noOpt completing 1 million calculations in 0.897 seconds. It may be worth noting that, despite its name, TR-noOpt performs tree simplification and the code generation may include low-level optimizations.

In Table 5, we see that more computational effort is required to calculate the Fibonacci sequence recursively, with the interpreter requiring 88.55 seconds to complete 1 million calculations of the 10th number. We can also see that significant improvement in throughput can be achieved through TR and its IL phase. While MicroJIT is

[6]The memory values did not change across the repeated executions.



(a) `IterativeFib` calculating 30th number in sequence, i.e., fib(30)



(b) `RecursiveFib` calculating 10th number in sequence, i.e., fib(10)

Figure 2: Comparison of Execution Times (in microseconds) of first 50 iterations of `IterativeFib` and `RecursiveFib` with invocation threshold 20

able to complete the task in 8.69% of the time that was required by the interpreter, TR-noOpt required 2.67% of the time and TR-cold required just 2.10% of the time. This performance discrepancy can be attributed to optimizations of `invokestatic` in TR's generated

	MicroJIT			TR-noOpt			TR-cold		
	mean	median	std.dev	mean	median	std.dev	mean	median	std.dev
`IterativeFib`	1025.44	1253.00	356.84	2446.13	2498.00	258.99	3100.34	3132.50	239.79
`RecursiveFib`	1016.49	1266.00	368.59	2376.59	2432.00	247.69	4976.59	4980.00	271.50

Table 2: JIT Compilation Time (in microseconds)

	MicroJIT	TR-noOpt	TR-cold
`IterativeFib`	64	1024	960
`RecursiveFib`	64	960	1280

Table 3: Memory (in kilobytes) for JIT Compilation

	Time (in seconds)			Operations per second	
	mean	std.dev	%	mean	std.dev
Interpreter	14.426	0.0324	100.00	69,315.85	153.76
MicroJIT	1.378	0.0047	9.55	725,672.55	2428.24
TR-noOpt	0.897	0.0039	6.22	1,114,194.82	4849.61
TR-cold	0.847	0.0045	5.87	1,180,377.12	6299.74

Table 4: Time to execute 1 million iterative invocations of fib(30)

	Time (in seconds)			Operations per second	
	mean	std.dev	%	mean	std.dev
Interpreter	88.55	0.105	100.00	11,292.26	13.33
MicroJIT	7.70	0.320	8.69	129,925.61	4688.08
TR-noOpt	2.37	0.008	2.67	420,178.73	1534.97
TR-cold	1.86	0.014	2.10	536,712.94	4257.15

Table 5: Time to execute 1 million recursive invocations of fib(10)

code: while the code generated by TR has the `call` instruction addresses patched directly to the JITed code, the `call` instructions in MicroJIT first jump to an intermediary routine to check if the callee has been JITed by TR, by MicroJIT, or not compiled at all (control returns to the interpreter). This additional step is currently required to maintain interoperability between MicroJIT and TR, though in the future, a MicroJIT-only build may eliminate it. Looking at the quality of the generated code, TR-noOpt has far fewer instructions than MicroJIT (28 versus 115), and for `IterativeFib`, the numbers are similar (36 versus 116)[7]. While we present a large improvement over the interpreter, we will continue to look for inexpensive operations we can apply to reduce the number of instructions generated, as well as to reduce the overhead when calling.

6 FUTURE WORK

MicroJIT is currently a limited solution, which is missing support for several key bytecodes. Our top priority is improving bytecode coverage, so we can support more micro-benchmarks, eventually transitioning to majority method support in established benchmarks, such as DaCapo. We have been making steady progress towards this goal and expect to reach full, or near-full coverage within the next several months.

TR compiles methods with varying levels of optimizations. Some of these levels, particularly those at the highest level of optimization, rely on profiling data gathered during interpretation. By compiling code so early with MicroJIT, this data set becomes much smaller and therefore less representative. One of our future research objectives is to emit profiling instructions into MicroJITed code and see if a templated JIT can create equivalent profiling data sets faster than the interpreter, without negatively impacting the performance of TR as an optimizing compiler. These penalties for optimizations are a real cost in terms of development time, processing power and memory usage; and not having them is one of the reasons why MicroJIT compiles so fast. However, some compiler optimizations are fairly inexpensive to implement. For example, a peephole optimization can be used to eliminate unnecessary loads [31], drastically improving performance, but can be done with a single look ahead operation for every store instruction in the source. Some future work can be focused on this field of optimizations, like what cheap optimizations can be added to a templated code generator or can we actually make those optimizations cheaper?

As a replacement for TR in resource constrained environments, MicroJIT needs to support the architectures used most commonly in that field. Today, those architectures are the AArch32 and AArch64 architectures. As a tool to facilitate faster warm-up times for TR, MicroJIT needs to support the architectures that run server-based Java applications. Those architectures include the x86-64 Architecture, the 64-bit PowerPC Architecture and the z/Architecture. As MicroJIT evolves, we want to extend its support to cover all these architectures and make it a feature available for all Eclipse OpenJ9 platforms.

7 SUMMARY

For some constrained workloads limited by memory and/or processor power, a lightweight JIT compiler may be the ideal mechanism to achieve performance improvements at runtime. This paper presents our work of adding a template-based JIT compiler, MicroJIT, to Eclipse OpenJ9 towards that end. While our bytecode coverage is thus far limited, the compilation results are promising; for our micro-benchmarks, we demonstrate significant improvements in performance over the interpreter, while at the same time, compiling more quickly and with less memory overhead than the default JIT compiler in Eclipse OpenJ9.

[7]The difference in code-size can be attributed to our reliance on an in-memory operand stack, as well as to our operations for register preservation.

ACKNOWLEDGMENTS

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] AdoptOpenJDK. 2020. *Prebuilt OpenJDK Binaries for Free!* Retrieved 2020-02-27 from https://adoptopenjdk.net/

[2] John Aycock. 2003. A Brief History of Just-in-Time. *Comput. Surveys* 35, 2 (June 2003), 97–113. https://doi.org/10.1145/857076.857077

[3] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. *ACM SIGPLAN Notices* 41, 10 (Oct. 2006), 169–190. https://doi.org/10.1145/1167515.1167488

[4] Eric Coffin, Scott Young, Kenneth B. Kent, and Marius Pirvu. 2019. A Roadmap for Extending MicroJIT: a Lightweight Just-in-Time Compiler for Decreasing Startup Time. In *Proceedings of 29th Annual International Conference on Computer Science and Software Engineering (CASCON '19)*. IBM Corp., Riverton, NJ, USA, 293–298.

[5] ComputerWeekly.com. 2002. *Write Once, Run Anywhere?* Retrieved 2019-06-17 from https://www.computerweekly.com/feature/Write-once-run-anywhere/

[6] MDN Contributors. 2020. *JIT Optimization Strategies.* Retrieved 2020-06-12 from https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey/JIT_Optimization_Strategies

[7] Keith D. Cooper and Linda Torczon. 2011. *Engineering a Compiler* (2nd. ed.). Elsevier.

[8] Oracle Corporation. 2020. *Java SE HotSpot at a Glance.* Retrieved 2020-03-30 from https://www.oracle.com/technetwork/java/javase/tech/index-jsp-136373.html

[9] Ben Corrie and Hang Shao. 2018. *Class Sharing in Eclipse OpenJ9.* Retrieved 2020-04-19 from https://developer.ibm.com/tutorials/j-class-sharing-openj9

[10] L. Peter Deutsch and Allan M. Schiffman. 1984. Efficient Implementation of the Smalltalk-80 System. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL '84)*. Association for Computing Machinery, New York, NY, USA, 297–302. https://doi.org/10.1145/800017.800542

[11] M. Anton Ertl and David Gregg. 2003. The Structure and Performance of Efficient Interpreters. *Journal of Instruction-Level Parallelism* 5 (Nov. 2003), 1–25.

[12] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software* (1st. ed.). Addison-Wesley Longman Publishing Co., Inc.

[13] Matthew Gaudet and Mark G. Stoodley. 2016. Rebuilding an Airliner in Flight: A Retrospective on Refactoring IBM Testarossa Production Compiler for Eclipse OMR. In *Proceedings of the 8th International Workshop on Virtual Machines and Intermediate Languages (VMIL '16)*. Association for Computing Machinery, New York, NY, USA, 24–27. https://doi.org/10.1145/2998415.2998419

[14] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. 2014. *The Java Language Specification, Java SE 8 Edition* (1st. ed.). Addison-Wesley Professional.

[15] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. 2013. Internet of Things (IoT): A Vision, Architectural Elements, and Future Directions. *Future Generation Computer Systems* 29, 7 (Sept. 2013), 1645–1660. https://doi.org/10.1016/j.future.2013.01.010

[16] Clemens Hammacher. 2018. *Liftoff: A New Baseline Compiler for WebAssembly in V8.* Retrieved 2020-06-10 from https://v8.dev/blog/liftoff

[17] Gilbert Joseph Hansen. 1974. *Adaptive Systems for the Dynamic Runtime Optimization of Programs.* Ph.D. Dissertation. Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, USA.

[18] Kim Hazelwood and Michael D. Smith. 2002. Code Cache Management Schemes for Dynamic Optimizers. In *Proceedings of the Sixth Annual Workshop on Interaction between Compilers and Computer Architectures (INTERACT '02)*. IEEE Computer Society, Washington, DC, USA, 102–102.

[19] John L. Hennessy and David A. Patterson. 2011. *Computer Architecture: A Quantitative Approach* (5th. ed.). Elsevier.

[20] Urs Hölzle and David Ungar. 1996. Reconciling Responsiveness with Performance in Pure Object-Oriented Languages. *ACM Transactions on Programming Languages and Systems* 18, 4 (July 1996), 355–400. https://doi.org/10.1145/233561.233562

[21] Alex Iliasov. 2003. Templates-based Portable Just-in-Time Compiler. *ACM SIGPLAN Notices* 38, 8 (Aug. 2003), 37–43. https://doi.org/10.1145/944579.944588

[22] Azul Systems Inc. 2020. *Zing Runtime for Java.* Retrieved 2020-03-30 from https://www.azul.com/products/zing/

[23] Docker Inc. 2020. *Docker Image Build.* Retrieved 2020-06-22 from https://docs.docker.com/engine/reference/commandline/image_build/

[24] Eclipse Foundation Inc. 2019. *IoT Developer Survey 2019 Results.* Retrieved 2020-04-29 from https://iot.eclipse.org/community/resources/iot-surveys/assets/iot-developer-survey-2019.pdf

[25] Richard Jones, Antony Hosking, and Eliot Moss. 2016. *The Garbage Collection Handbook: The Art of Automatic Memory Management* (1st. ed.). CRC Press.

[26] Iffat H. Kazi, Howard H. Chen, Berdenia Stanley, and David J. Lilja. 2000. Techniques for Obtaining High Performance in Java Programs. *Comput. Surveys* 32, 3 (Sept. 2000), 213–240. https://doi.org/10.1145/367701.367714

[27] Ken Kennedy and John R. Allen. 2001. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach* (1st. ed.). Morgan Kaufmann Publishers Inc.

[28] Prasad A. Kulkarni. 2011. JIT Compilation Policy for Modern Machines. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications (OOPSLA '11)*. Association for Computing Machinery, New York, NY, USA, 773–788. https://doi.org/10.1145/2048066.2048126

[29] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. 2014. *The Java Virtual Machine Specification, Java SE 8 Edition* (1st. ed.). Addison-Wesley Professional.

[30] Daryl Maier and Xiaoli Liang. 2017. Supercharge a Language Runtime!. In *Proceedings of the 27th Annual International Conference on Computer Science and Software Engineering (CASCON '17)*. IBM Corp., Riverton, NJ, USA, 314–314.

[31] William M. McKeeman. 1965. Peephole Optimization. *Commun. ACM* 8, 7 (July 1965), 443–444. https://dl.acm.org/doi/pdf/10.1145/364995.365000

[32] Eclipse OMR. 2020. *OMR Command-line Options.* Retrieved 2020-04-19 from https://github.com/eclipse/omr/blob/master/compiler/control/OMROptions.cpp

[33] Eclipse OpenJ9. 2020. *Eclipse OpenJ9 Repository.* Retrieved 2020-02-27 from https://github.com/eclipse/openj9

[34] Eclipse OpenJ9. 2020. *The JIT Compiler.* Retrieved 2020-04-20 from https://www.eclipse.org/openj9/docs/jit/

[35] Marius Pirvu. 2018. Optimize JVM Startup with Eclipse OpenJ9. Retrieved 2019-06-19 from https://developer.ibm.com/articles/optimize-jvm-startup-with-eclipse-openjj9

[36] B. Ramakrishna Rau. 1978. Levels of Representation of Programs and the Architecture of Universal Host Machines. *ACM SIGMICRO Newsletter* 9, 4 (Nov. 1978), 67–79. https://doi.org/10.1145/1014198.804311

[37] Ricardo Nabinger Sanchez, José Nelson Amaral, Duane Szafron, Marius Pirvu, and Mark Stoodley. 2011. Using Machines to Learn Method-Specific Compilation Strategies. In *International Symposium on Code Generation and Optimization (CGO 2011)*. IEEE, 257–266. https://doi.org/10.1109/CGO.2011.5764693

[38] Kazuyuki Shudo, Satoshi Sekiguchi, and Yoichi Muraoka. 2004. Cost-Effective Compilation Techniques for Java Just-in-Time Compilers. *Systems and Computers in Japan* 35, 12 (Nov. 2004), 10–24. https://doi.org/10.1002/scj.10564

[39] James E. Smith and Ravi Nair. 2005. *Virtual Machines: Versatile Platforms for Systems and Processes* (1st. ed.). Elsevier.

[40] TIOBE The software quality company. 2019. *The C# Programming Language.* Retrieved 2019-06-17 from https://www.tiobe.com/tiobe-index/csharp/

[41] TIOBE The software quality company. 2019. *The Java Programming Language.* Retrieved 2019-06-17 from https://www.tiobe.com/tiobe-index/java/

[42] Federico Sogaro, Eric Aubanel, Kenneth B. Kent, Vijay Sundaresan, Marius Pirvu, and Peter Shipton. 2017. MicroJIT: A Lightweight, Just-in-Time Compiler to Improve Startup Times. In *Proceedings of the 27th Annual International Conference on Computer Science and Software Engineering (CASCON '17)*. IBM Corp., Riverton, NJ, USA, 140–150.

[43] Toshio Suganuma, Takeshi Ogasawara, Mikio Takeuchi, Toshiaki Yasue, Motohiro Kawahito, Kazuaki Ishizaki, Hideaki Komatsu, and Toshio Nakatani. 2000. Overview of the IBM Java Just-in-Time Compiler. *IBM Systems Journal* 39, 1 (Jan. 2000), 175–193. https://doi.org/10.1147/sj.391.0175

[44] The JUnit Team. 2020. *JUnit.* Retrieved 2020-05-12 from https://junit.org

[45] V8 Team. 2017. *Launching Ignition and TurboFan.* Retrieved 2020-06-10 from https://v8.dev/blog/launching-ignition-and-turbofan

[46] Mark Thom, Gerhard W. Dueck, Kenneth Kent, and Daryl Maier. 2018. A Survey of Ahead-of-Time Technologies in Dynamic Language Environments. In *Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering (CASCON '18)*. IBM Corp., Markham, Ontario, Canada, 275–281. http://dl.acm.org/citation.cfm?id=3291291.3291320

[47] Seth Thompson. 2016. *Experimental Support for WebAssembly in V8.* Retrieved 2020-06-10 from https://v8.dev/blog/webassembly-experimental