# MicroJIT: A Template-Based Just-in-Time Compiler for Constrained Environments

by

Eric Douglas Coffin

**Bachelor of Computer Science, UNB, 2003**

**A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF**

**Master of Computer Science**

In the Graduate Academic Unit of Computer Science

| | |
|---|---|
| Supervisor(s): | Kenneth B. Kent, Ph.D., Computer Science |
| Examining Board: | Eric Aubanel, Ph.D., Computer Science, Chair |
| | Luigi Benedicenti, Ph.D., Computer Science |
| | Eduardo Castillo-Guererra, Ph.D., Electrical and Computer Engineering |

This thesis is accepted by the
Dean of Graduate Studies

**THE UNIVERSITY OF NEW BRUNSWICK**

**August, 2020**

# Abstract

The widespread adoption of Java Virtual Machine (JVM) runtime environments has increased application portability and security while making programming much more accessible. Programs that run on the JVM are initially interpreted, however, to improve performance, a Just in Time (JIT) compiler may be employed at run-time to translate parts of the program to native code. Later, the generated code can be executed directly, bypassing the need to interpret the code. Before code generation, some JIT compilers build an intermediate representation (IR) to apply transformations to, allowing for further performance improvements. For some constrained runtime environments, the overhead required by an optimizing JIT compiler may be too high to be useful. An alternative approach is to employ a non-optimized, template-based JIT compiler that copies predefined machine-code templates instead of generating and manipulating IR. In this work, we present such a JIT compiler, MicroJIT, geared toward generating native code as fast as possible for Eclipse OpenJ9, an open-source Java runtime environment. Our initial results for compilation time and memory overhead are promising. For our custom benchmarks, we can compile with significantly less overhead than the default optimizing JIT compiler in OpenJ9.

# Dedication

To my loving wife Melissa. Thank you for all the support.

# Acknowledgements

It has been a great experience getting to this point.

Dear reader, you will note that I chose to avoid writing from the first person. This choice was intentional, as there is a group of individuals that helped make this work a reality: Scott Young, Harpreet Kaur, and Julie Brown. Through 2019 and more recently (and remotely), you all pulled together to make MicroJIT on Eclipse OpenJ9 a possibility. Thank you all. Beyond this immediate group, I would also like to mention the exceptional previous work on MicroJIT by Federico Sogaro. His work paved much of the way for us.

Next, I would like to thank my supervisor, Dr. Kent, for his guidance on this journey. From day one, you helped me plot the chart that led me here. The clarity with which you operate is astounding, Sir. From IBM Canada, I would also like to acknowledge the guidance and expertise provided by Marius Pirvu while working on this project. From the CASA lab, I would like to thank Stephen MacKay for his constant attention to detail, and to both Aaron and DeVerne for keeping the systems up and running!

Finally, I would like to acknowledge the funding support provided by the Atlantic Canada Opportunities Agency through the Atlantic Innovation Fund program. I would also like to thank the Centre for Advanced Studies—Atlantic for providing facilities to conduct this research. Last but not least, I am grateful to the New Brunswick Innovation Foundation for their contribution to this project.

# Table of Contents

## Vita

# List of Tables

# List of Figures

# Chapter 1

# Introduction

> *It has long been an axiom of mine that the little things are infinitely the most important.*
>
> —Sir Arthur Conan Doyle
>
> *The Memoirs of Sherlock Holmes [1]*

With the rise of the Internet of Things (IoT) [2], the number of embedded, connected devices has seen a significant surge in growth over the past decade. One research firm forecasted that by 2020 there would be 5.8 billion IoT endpoints deployed in industrial, commercial, and automotive markets—an increase of 21% over 2019 [3]. 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 [4]. According to the same survey, while C remains the most popular language for developers writing code for the majority of 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,

[1] In the survey, Java was ranked number four among most popular languages used for constrained devices [4].

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.

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 add additional overhead cost to the application. Given that Java is the programming language of choice for the majority of the industry [5], 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. Previous work such as the Squawk VM [6], and more recent works such as Darjeeling [7], and CapeVM [8] have looked at designing such virtual machines for constrained systems such as those found in wireless sensor networks. Any constrained system executing JVM-based workloads will be concerned with the following sources of overhead:

- The modules of the runtime and the application are loaded and linked during execution. This process, called dynamic class loading, is part of the JVM specification [9].

- Interpretation of the application, which involves fetching and decoding bytecodes may be an order of magnitude slower than executing a native binary

[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.

image [10].

- As a stack-based virtual machine designed for portability, the reliance on physical registers by the JVM is minimal. Instead, program state is typically mapped to memory, adding significant overhead to the process[3].

- Finding the right balance between performing work and compiling code dynamically for future gains in performance can be a challenging problem for any workload running on a virtual machine. This issue is exacerbated when dealing with constrained environments, where the overhead associated with the JIT compiler may be too high to be useful. Just-in-Time (JIT) compilation, which can improve performance over interpretation, initially adds overhead to the overall execution of the program as it compiles methods into native instructions [12,13].

- The JVM specification requires that VMs check the safety of certain memory access operations during run-time. For example, operations involving objects require null-reference checks, and array operations require index out-of-bounds checks. While this adds overhead, the requirement can eliminate an entire class of security vulnerabilities [4].

- Automatic memory management requires a garbage collector to independently and safely clean the program memory, or heap, during run-time [14].

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—in particular, to JIT compilation. In particular, we will investigate the overhead of the existing JIT compiler on the Eclipse OpenJ9 JVM,

[3]Recall that in a typical memory hierarchy, as we move further away from the CPU core, the time to access memory increases and the size of the memory increases. Registers provide the fastest access but offer the smallest storage. As such, it is desirable to make full use of the available registers [11].

[4]JIT compilers might remove safety checks that it can show are redundant.

an open-source, high-performance JVM designed with modern cloud applications in mind [15]. Looking at the JIT compiler in Eclipse OpenJ9, Testarossa—or TRJIT—we see a highly-tunable, method-based, optimizing compiler. While TRJIT supports fast compilation, generating non-optimized code, it requires an intermediary stage where an intermediate representation, or intermediate language (IL) is generated[5]. It is possible that the IL generation phase may introduce too much overhead for constrained environments where low-optimized code is good enough. One approach to solving 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 then later be executed. By eliminating the IL-phase, we expect that JIT compilation will have a smaller footprint and take less time to perform[6].

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 [16], this compiler will be template-based with the goals 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 [17], 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.

This work will involve the following items:

- We will incorporate a second, lightweight, template-based JIT compiler (MicroJIT) into Eclipse OpenJ9.

[5]The terms intermediate representation (IR) and intermediate language (IL) are used interchangeably.

[6]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.

- To guide the order of bytecode-template implementation, we will record the execution frequency of bytecodes in several standard benchmarks.
- To establish a baseline, we will measure the overhead of the interpreter operating without any JIT compiler.
- We will measure the overhead associated with TRJIT and with MicroJIT.
- We will 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 thesis reads as follows: first, Chapter 2 provides an overview of the background, looking at JIT compilation, the JVM, Eclipse OpenJ9, as well as constrained devices. Next, Chapter 3 discusses the related work. In Chapter 4, we focus on the design of the MicroJIT and how we integrated it into Eclipse OpenJ9. Chapter 5 lists our results and provides an analysis. Chapter 6 outlines the future work. Finally, Chapter 7 concludes the work, summarizing our findings and offering our recommendations.

# Chapter 2

# Background

> *The most amazing achievement of the computer software industry is its continuing cancellation of the steady and staggering gains made by the computer hardware industry.*
>
> —Henry Petrosky

In this chapter, we will discuss several background concepts, including the Java Virtual Machine, Just-in-Time (JIT) compilation, Eclipse OpenJ9, and finally, constrained devices. In the section on the JVM, we will discuss how the JVM specification allows for portability and the challenges it poses for performance. In the section on JIT compilation, in addition to discussing the importance of it for performance on the JVM, we will provide an overview of different techniques used to answer the questions: what should we compile, and when should we compile it. The section on Eclipse OpenJ9 will provide a brief background on the open-source JVM our work will focus on, delving briefly into its optimizing JIT compiler, Testarossa. In the last section, we will investigate some concerns involving constrained devices, why Java could be a good fit, and the hurdles typically encountered when designing a JVM for such devices.

## 2.1 The 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 [5,18]. The designs of enterprise applications, often with complex domain models, rules, and requirements, are aided by these high-level, object-oriented languages and by 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:

- Programs written in C# compile to the Common Intermediate Language (CIL), which is designed to be executed by the Common Language Runtime (CLR) platform [19]—most commonly found on Windows-based operating systems. We will not focus on C# in this work.

- Programs written in Java compile to Class files and execute on Java Virtual Machines (JVMs). See Figure 2.1 for a depiction of this process[1]. For an in-depth discussion of the Java language, please refer to the Java Language Specification [20].

This focus on portability gave rise to the phrase "write once, run anywhere [21]." To ensure cross-platform compatibility, the JVM must adhere to the Java Virtual Machine Specification [9], which describes the following items:

- Program layout - How the application should look once loaded in memory.

- Linking and Loading - The process by which the JVM should dynamically load and link a program.

[1] Java programs are compiled using the Java Compiler or `javac`

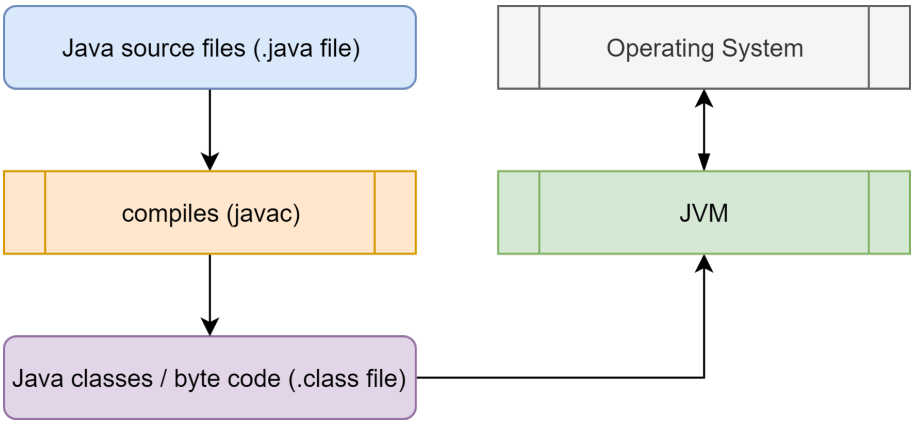


Figure 2.1: A Java program is compiled to *class* files containing metadata and bytecode instructions. Any compatible JVM can then execute this program.

- Program verification - For program safety, as programs execute, specific instructions must be verified, such as performing array-out-of-bounds checks when a program reads from or writes to an array object.
- Stack machine descriptions - The JVM uses a stack-based architecture to represent a program during operation.
- Class file format - The file layout to which a class file must adhere.
- Bytecode listing - In a program, the actual instructions of a program, or the bodies of methods are streams of bytecodes. The specification describes each of the possible instructions.

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 chance of inadvertently introducing memory errors decreases.

Similar to how Fortran or C made programming more accessible than writing assembly, these high-level languages have done the same over C or C++, that is—reducing

the required low-level knowledge to start programming in the language [22]. Some argue that by allowing developers to program without a low-level, or system-based understanding of the application, the quality will decrease, or the essence of the craft will be lost. However, successful systems are designed with successive layers providing an increased level of abstraction, and high-level languages running on virtual machines are no different. As we shall see, for an application with little control of the underlying hardware to achieve high-performance, the performance of the underlying runtime, or virtual machine (VM), is critical.

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 VMs: 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) [10]. 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 Interface (ABI), such as when making a system call such as `fork` on Linux to spawn a new thread (see Figure 2.2).

Some examples of production-grade JVMs include Eclipse OpenJ9 [23], Oracle's HotSpot [24], and Azul Zing [25]. It is worth noting that the JVM has become an essential platform for languages beyond Java. Languages such as Scala [26], Kotlin [27], and Clojure [28] 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

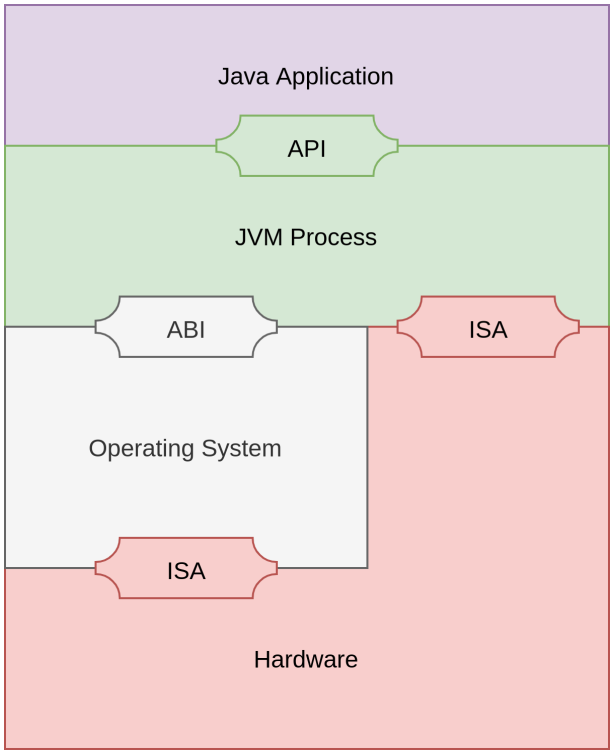


Figure 2.2: Interactions between the application and the underlying system are made through API calls. Interactions between the JVM and underlying host are made across the system ABI—as systems calls to the operating system, or through instructions directly to hardware [10].

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. For example, provided the following bytecode stream `[iload1,iload2,iadd]`: `iload1` would push the integer value from local storage index `1` onto the operand stack; `iload2` would push the value at index `2` onto the stack; then `iadd` would pop the two values and push the resulting sum back onto the operand stack.

Before calling a method, instructions are issued to push the method arguments and the target for the method onto the operand stack. For instance methods, the bytecode `invokevirtual` is executed, which pops the arguments and target off the operand stack and pushes a new stack frame onto the stack before calling the target method. Once the called method is ready to return, the `ret` instruction is executed: the stack frame is popped, and the return result is pushed onto the operand stack of the caller.

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.[2] 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 (GC) is the process by which the objects in the heap that are no longer referenced are freed [14]. For a collector to be viable, it must satisfy the requirement that all garbage will eventually be "collected." Although, GC is a rich topic involving many aspects of computing, and an in-depth discussion is beyond the scope of this work, we will note two things:

- The time spent performing GC is time spent not executing the user workload. Therefore, the garbage collector should be as efficient as possible while performing cleanup. For instance, an inefficient collector might leave the heap space fragmented, leading to allocation failures earlier.
- The executing application must support GC: in between method calls, checks are performed to see if GC is required. Also, each stack frame must support GC by exposing its root set, or pointers into the heap, so that collection can take place.

In addition to GC, the other process that JVMs perform that may add significant overhead to the executing application is Just-in-Time (JIT) compilation which we will discuss in the next section.

[2]Recall that a stack is a common mechanism for maintaining the state of a running application. Each function call is represented by a stack-frame that contains the program counter, or what the currently executing line of code is, any argument values for the function, a frame pointer, any local variables, as well as return address of the calling function. The x86-64 architecture has registers designed explicitly for this: RIP stores the program counter, and RBP stores the base pointer of the current stack frame. When function calls are made, any state that needs to persist can be pushed on the stack before the call, and after the function returns, these values can be restored by popping them off the stack. Typically only programmers writing programs in assembly language need to manage the stack manually.

## 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 inch closer to that of their Ahead-of-Time (AOT) compiled counterparts written in C, or C++ [29]. In addition to portability and safety, the high-performance garbage collectors and JIT compilers found in production-grade JVMs has helped place Java at the top of languages used for server-based workloads [5].

According to Rau, application binaries can be divided into three groups: those that are composed of directly executable machine instructions, those that are composed of higher-level instructions that must be parsed and interpreted before execution, and those that are composed of directly interpretable instructions [30]. 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: programs composed of directly interpretable instructions.

While interpreters should be designed to be efficient—minimizing the number of indirect branches [10,31], 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[3] [10]. 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 run-time 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

[3]Indirect branching occurs when the address operand for an instruction is located in memory—that is, the value must be obtained through dereferencing, or accessing memory. While this additional I/O will add to the overhead of an interpreter, performance of the CPU may also suffer due to the instruction pipeline and branch prediction. When the target of a branch is not known or cannot easily be guessed, the instruction pipeline, which in many modern architectures supports prediction, will stall until the target is known.

may be achieved. This process, known as Just-in-Time (JIT) compilation, is often combined with profiling and or analysis to optimize the generated code [10,12].

Considering JIT compilation is an expensive activity, in order to allow for the continued execution of the workload, JIT compilation is often performed selectively on only the most frequently executed blocks of code [32]. As described by Hansen in his work on Adaptive Fortran [33], 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 then decremented. Once 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 [10,34]. 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 2.3).

An alternative to using entire methods as compilation units is to use basic blocks. Basic blocks are snippets of code that begin at a jump or branch target and end at either branch or a jump. By compiling basic blocks, only the most frequently executed code paths are compiled, which should lead to less required storage and may be beneficial in constrained environments [35]. These basic blocks can later be further optimized by combining them into superblocks or traces. Trace-based compilation units, which can span methods, circumvent the overhead associated with invocation calls, as the JIT compiler translates entire paths of execution through the program [36]. Traces can be further combined into larger trace-graphs providing

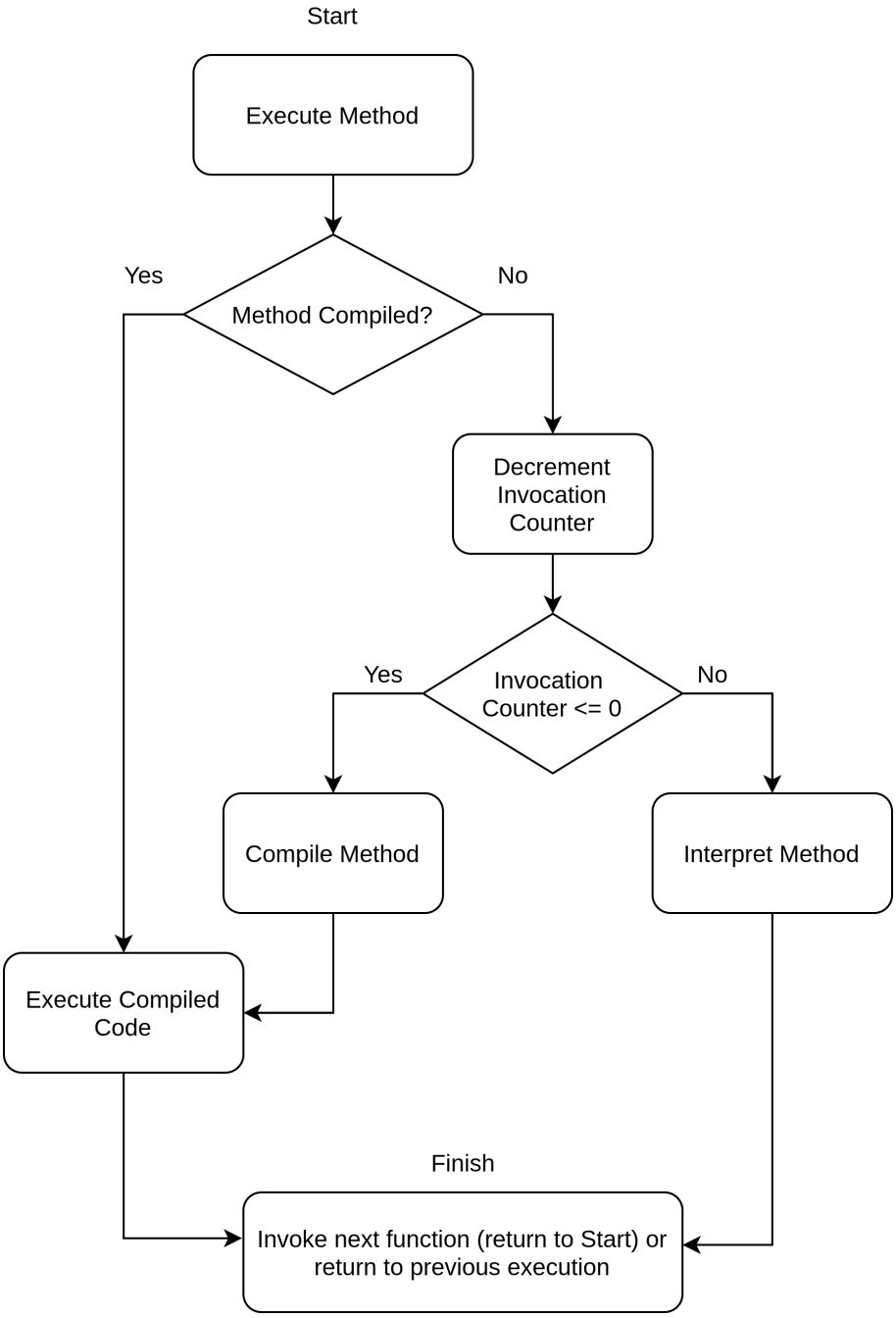


Figure 2.3: When a method is to be executed, the execution engine first checks if the method has already been JIT-compiled. In this example, compilation and, in turn, execution occurs synchronously, potentially occurring on the currently executing application thread.

further performance improvements [37]. While only the original code paths are compiled, potentially leading to less time spent compiling, and less code generated, in some scenarios, trace-based compilers can be detrimental. One issue arises when a code path diverges from the JITed trace, such as when an `else` clause must execute, but it is not part of the trace, and execution must return to the interpreter. Another issue that must be considered is a potential code explosion, as a single block of code may be part of numerous traces. These issues can add considerable complexity to a JIT compiler.

An alternative approach to selective-compilation is to profile executing code to detect the "hot-spots" or sections executing most frequently by inserting generated code within the application bytecode to increment profile counters. Once a counter reaches its threshold, code executing on a separate thread will note the source block or method and compile it. This approach may be combined with the less-granular, method-based invocation counters to detect when execution is focused on a single method for some time, for instance, during the execution of a large loop. In such a case, the JITed block may replace the currently executed block through a process called on-stack-replacement[4] (OSR) [38,39].

One additional heuristic to consider when choosing "what" to compile is the size of a particular method [40]. For a method-based JIT compiler, the method must be compiled in its entirety before compilation can continue. For large methods, this has two impacts on the system. First, large methods will likely take more time to compile. Second, larger methods will likely produce more machine code, placing more pressure on the code cache. Similar to the invocation threshold, exposing this parameter, say as bytes, to the user for configuration allows the user to customize the JIT compiler to their particular workload and environment. For constrained systems, targeting smaller methods should result in a smaller code cache, which in

[4]OSR may also be used for debugging purposes, replacing the JITed target stack frame with an interpreted version.

turn could lead to less management overhead. One challenge in this approach is knowing the value of compiling large methods. The VM would conceivably spend more time in large methods, meaning targeting large methods could be valuable. In many cases, though, large methods have code paths with few executed instructions, such as an `if` statement surrounding the majority of the method statements. In the case that the `if` block is not executed, the program might only execute a return statement. Considering the general nature of the JVM, providing the method size as a tunable parameter to the JIT compiler is a good option. Providing the JIT compiler with a list of methods or packages to compile or ignore could also allow the user to tune the JVM for a specific application.

With the drive toward massively multi-core systems, an essential consideration for compilation is "when" a method should be compiled. A method can be compiled synchronously on the same thread executing application code, or it could be placed in a queue to be compiled asynchronously by one or more compilation threads. With asynchronous compilation, the use of a queue can allow us to defer compilation and even add prioritization to the process [32]. For systems with many cores, it may be beneficial to choose lower invocation thresholds, while for systems with fewer cores, the inverse may be true [32].

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 [41]. For JIT compilers with multiple optimization levels, it may prove useful to allow a user to specify heuristics per optimization level. For instance, the lowest compilation level might require 200 invocations and be limited

to methods of 150 or fewer bytes. In comparison, the next higher compilation level might require 1000 invocations and be limited to methods of 500 or fewer bytes.
In order to aid in performing optimizations, it is typical for a JIT compiler to build an intermediate representation (IR) 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 [10,42,43]:

- Dead-code Elimination: If certain statements are guaranteed never to execute, or have no impact on the program result, then they may not need to be generated. By eliminating them, fewer operations are executed, and the program size can decrease.

- Constant Propagation: This optimization involves updating variable references to constant values with the actual constant value, which may eliminate memory accesses by encoding the constant value directly in the machine instruction.

- Constant Folding: Certain expressions involving constants can be calculated at compile-time rather than during run-time, allowing the expression to be calculated once instead of multiple times.

- Strength Reduction: This optimization involves replacing operations within loops that involve an invariant and an induction variable with less expensive operations, such as replacing a multiply operation with an addition.

- Code Hoisting: Similar to strength reduction, this optimization involves moving a computation involving a loop invariant expression above the loop, so it is not repeatedly computed.

- Loop Unrolling - The time spent executing the branching and jumping in a loop may be performed more quickly by repeatedly generating the body of the loop.

- Inline Substitution - Inlining involves copying the body of a called method into the body of the caller, eliminating the need for the call. If the body of a method contains fewer instructions than the instructions to transition into and out of the method, it may be a good candidate for inlining.

- Peephole optimizations - By storing machine code instructions in a buffer before they are written to the code cache, we can scan ahead to look for redundant instructions that can be combined or removed. For instance, an instruction that stores a value from a register to memory, only to load it again into the same register, might be removed [44].

Other optimizations only make sense within the context of a dynamic run-time environment [45]. For instance, polymorphic-inline-caches can improve the performance of dispatching in object-oriented languages by short-circuiting a lookup to the dispatch table with a simple address check [46]. Code reorganization is another such optimization that can improve cache locality: the code generated for methods that are commonly called in sequence can be arranged within memory in the same order. Building on this concept, methods that are part of multiple sequences (have multiple callers), could have their code generated in multiple places. However, the overhead of this process may be considerable, both in terms of code-growth and in terms of bookkeeping [10].

Compatibility is of primary concern when dealing with executing generated code. We will look at this from two different viewpoints: garbage collection and exceptions. Recall that a garbage collector must eventually recover all the garbage in the heap—that is, it will eventually free any memory that is no longer to by the program. In order for the garbage collector to fulfill this duty, the root-set, or the pointers into the heap, must be discoverable by walking the thread stack frames. Therefore, the stack frames generated by JITed code must provide a compatible mechanism for the garbage collector to discover any object references (addresses) stored in the local

array, on the operand stack, or stored in registers.

The Java language provides a rich exception handling system, allowing a running application to recover from an unexpected state. When an exception occurs, it is *thrown* from the violating instruction. When this occurs, the VM will search the currently executing method for a compatible exception handler defined in an enclosing `catch` statement. If no valid handler is found, then the VM continues to "unroll" the stack frame, searching each caller for a valid handler. Finally, if no handler is found, the program is halted. If the exception occurs in the generated code, say by a divide by zero, the processor may generate an exception that is trapped, passing control back to the operating system. The VM will have had to register a handler to deal with the potential exception, to which the operating system then passes control. The state of the original methods may need to be rebuilt at this point as knowing the index of bytecode and, in turn, the line of source code that triggered the exception is critical. Restoring this state may also involve rebuilding the operand stack and local array with values that had previously been stored in registers, or whose operations had been optimized away, which may require interpreting instructions up to the exceptional statement.

## 2.3 Eclipse OpenJ9

The JVM implementation this work will apply to is Eclipse OpenJ9 for Java 8 [23]. In particular, we will focus on its implementation for Linux on the x86-64 architecture[5,6]. This open-source project, which implements the OpenJDK specification and is available under the AdoptOpenJDK project [49], is designed for low overhead, quick startup, and high throughput. Also, OpenJ9, which originated from 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 following production-grade components: 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 [15,50]. 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) [51]. 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 *cold*, with roughly 20 optimizations applied, all the way to *scorching*, where as many as 170 optimizations can be applied [52,53]:

- The initial optimization level, applied when methods are first compiled, is the *cold* level. The goal of the *cold* level is to get code generated quickly for the broadest range of methods.

[5]As of writing, other versions of the JVM also exist, including support for Java 9 through Java 13. Implementations also exist for other platforms such as z/Architecture, PowerPC, ARM32, and AARCH64, as well as for the macOS and Windows operating system.

[6]The x86-64 instruction set is considered Complex Instruction Set Computing (CISC) based. CISC provides an instruction set with high orthogonality, providing several addressing mechanisms for instructions. Many operations, such as adding integers, can typically be performed between registers, or directly between a register and memory [47]. This can be contrasted with Reduced Instruction Set Computing (RISC) architectures where such operations are strictly register based [48]. CISC-based architectures typically support more instructions, but may result in smaller programs, while RISC-based architectures have fewer instructions, but may result in slightly larger programs.

- After the application has run for some time and methods reach their invocation threshold, they may be recompiled for the *warm* level.
- Beyond the *warm* level, a sampling thread is used to detect methods that have high CPU utilization.
- Methods that use more than 1% CPU time are compiled at the *hot* level.
- Methods that use more than 12.5% CPU time are compiled at the *very-hot* level, with profiling instructions to guide future optimizations when the methods are later recompiled to the highest level—*scorching*.
- There is another compilation level named *noOpt*, which applies tree simplification and may apply low-level optimizations to the generated code. This option may be used to improve application startup time in large applications [53].

Command-line arguments for TR can be passed to the JVM with the `-Xjit` option group [54]. While dozens of options exist, several of significance 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. The default value is 65,535, or the 16-bit value 0xFFFF.
- `codetotal` - Available memory for generated code in KB. The default value for this is 256MB (256*1024*1024).
- `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.

```
1 treetop--> istore a
2             |
3             imul--- isub-------+
4             |                  |
5             iadd               |
6             |                  |
7             +-----> iload b <--+
8             |                  |
9             +-----> iload a <--+
```

Listing 2.1: OMR IR representation for (a+b)*(a-b). Note that the iload nodes are reused [55].

- `breakOnEntry` - A debugging option generating a breakpoint instruction (`int3` on x86-64) at the start of the generated code.

The JIT compiler in OpenJ9 builds an intermediary representation (IR) called *trees*, on which it performs optimizations before generating native code. Based on basic-blocks, the IR is arranged into directed acyclic graphic (DAG) structures called trees, composed of nodes that 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 such as integer, pointer, double, float, and vector. Trees with side-effects, which cannot be reordered, belong to a list of elements called treetops [55]. Existing nodes can be reused within a given tree, making optimizations such as common subexpression elimination relatively simple (see Listing 2.1).

During compilation, fewer resources are available on the platform for executing the 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 also 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 [11].

The type of workload performed on the device may also be indicative of a constrained environment. That is, for very constrained devices, the work will likely be limited to only basic operations. For example, sensor-devices may have constraints for energy efficiency, processing power, memory, connectivity, and I/O. For less-constrained devices, the work will be more involved but still relatively singular. For example, a sensor aggregator or gateway may collect readings from distributed sensor-devices over some time [56]. Such a device would require more memory, more processing power, and would likely have a dedicated power supply. Finally, we reach servers, which have the most varied workloads and have the least constrained resources[7]. These servers may serve user requests for a web application, run machine-learning algorithms, or serve any number of other purposes.

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—forecasted to be 5.8 billion for industrial and commercial applications by the end of 2020 alone [3], comes the increased need for software and thus software developers. Meeting this demand in a timely fashion with secure and robust software

[7]While servers may be the "least-constrained," they too operate under constraints.

will be a significant challenge for the industry. Revisiting the Eclipse IoT developer survey [4] from 2019, we see that although C is the most popular language for the majority of constrained devices, Java is in the top 5. 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 an crucial factor, and as such, JIT compilation will play an essential role.

# Chapter 3

# Related Work

*Somewhere, something incredible is waiting to be known.*

—Carl Sagan

In the following sections, we discuss the related work. First, we discuss works that explore the use of multiple JIT compilers in a single runtime environment. Next, we look at Ahead-of-Time compilation and how it can be used in OpenJ9 to reduce the overhead in constrained environments. Finally, we look at other efforts to customize runtime environments for Java for constrained devices.

## 3.1 Multiple JIT Compilers

The idea of using two JIT compilers in a single JVM was put forth by Detlefs and Agesen. They suggested that one compiler performs fast, non-optimized compilations, while the other compiler focuses on performing slow, optimized compilations [57]. They found they were able to achieve the best results on the SPECJVM98 benchmark suite [58] when combining the two compilers: the top-performing configuration compiled all methods with the fast compiler except for the top 10 frequently called methods, which were instead compiled with the slow compiler.

Work by Sogaro et al. incorporated the previous MicroJIT into IBM J9, comple-

menting the default optimizing JIT compiler [16]. They were able to show that their template-based JIT compiler could improve the startup times of programs in the DaCapo Benchmark Suite while allowing TRJIT to take over compilation to further improve performance in the throughput phase.

Recent work on the JavaScript V8 runtime environment to improve the startup time for WebAssembly (WASM) [59] has led to the development and addition of a second, non-optimizing JIT compiler, called Liftoff to V8 in v6.9 [60]. Once the application has been compiled by Liftoff and is executing, TurboFan—the default, optimizing compiler in V8, can take over and recompile the hottest functions.

## 3.2 Ahead-of-Time Compilation

For constrained devices, reducing the amount of work performed by the JIT compiler will reduce the JVM overhead. One way to achieve this reduction is to utilize Ahead-of-Time (AOT) compiled code. Considering that the Java Virtual Machine specification states that linking and loading must happen dynamically [9], 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 improve 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 [61]. For long-lived environments capable of persisting run-time generated code between executions of the Java application, this approach provides a very efficient mechanism for reducing startup time [62]. 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 [63], although options are available to pre-warm an image during the building of a Docker Image [64].

## 3.3 Constrained Java Virtual Machines

Previous work has looked at customizing Java applications and or Java Virtual Machines specifically for constrained environments. Squawk VM [6], from Sun Microsystems, was written in Java, and designed to run on bare metal without the need of an operating system. Also, to maximize performance, the platform did not provide any class loading subsystem, and instead, Java classes were compiled into an intermediate format that could be packed and loaded more efficiently at runtime.

Native Image is work from Oracle that aims to reduce application startup time and memory overhead by generating Java-based native applications using GraalVM. They accomplish this by AOT-compiling the entire application and bundling this with the necessary VM runtime components called SubstrateVM [65] [66]. Naturally, with AOT-compiling the entire application, dynamic class loading is not possible. Although the native application cannot benefit from JIT compilation at run time, the AOT compilation can be informed by profile data gathered during previous non-AOT executions.

# Chapter 4

# Design

> *All parts should go together without forcing. Therefore, if you can't get them back together again, there must be a reason. By all means, do not use a hammer.*
>
> —IBM maintenance manual 1975

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.

The motivation behind adding a template-based JIT compiler to Eclipse OpenJ9 is to explore the potential of providing a lighter-weight alternative for constrained environments. By replacing the tree-based IL-generation and transformations with non-optimized, fast code generation, we hope to reduce the time required to perform compilations while still offering improvements over the performance of the inter-

preter. We recognize the tradeoff here: by reducing the quality of the generated code, the loss to potential throughput will be significant. Still, for some workloads operating in constrained environments, having low-optimized code that is quick to generate may be desirable.

In this chapter, we will discuss the design of MicroJIT. We begin by providing an overview of our original goal to port the previous MicroJIT to Eclipse OpenJ9, as well as the discoveries we made along the way that led to us rewriting the compiler. We then discuss how we integrated MicroJIT with the existing JIT compiler, Testarossa, as well as several of the benefits we inherit from the integration. Next, we look at architectural considerations for our target platform, x86-64 Linux, and discuss features of the generated code. Finally, we look at our existing bytecode support, our implementation strategy moving forward, and our framework for performing regression testing.

## 4.1 From Port to Rewrite

This work initially began with us trying to port the previous MicroJIT [16] for IBM J9 to Eclipse OpenJ9 for Linux x86-64. Once ported, our motivation was to then expand upon it: extending its bytecode support, adding support for asynchronous compilation, and adding support for profiling [17]. However, as we dove further into the work, we eventually realized that porting the project could involve more effort and risk than rewriting the compiler from scratch. The main issues we encountered were as follows. First, the mechanism in the previous MicroJIT for returning to the Interpreter when an unsupported bytecode was encountered would not work in Eclipse OpenJ9. This mechanism allowed control to pass from the execution of a JIT-compiled method back to the Interpreter to interpret a single bytecode and then switch back to the generated code for further execution of the JITed method. Eclipse OpenJ9, on the other hand, uses what it calls "actions" to change the state of the Interpreter [67]. These actions are hooks provided by the JVM to ensure the state of the program at the time of a transition is correct. For us, this meant that the original mechanism of jumping back and forth between the Interpreter and the generated code could not work. It also meant that we would need to support all of the bytecodes of a method to compile it. If we encountered a single bytecode that we did not support, the compilation would fail, and the interpreter would take over. Second, the shape of the stackframe the original MicroJIT maintained was not compatible with Eclipse OpenJ9. In the transition of open-sourcing parts of J9 into Eclipse OpenJ9 with its modularized Eclipse OMR components, the stack frame shape—or the layout of memory used by a JITed function at the time of execution, had changed. In order to provide interoperability with the JVM, we would need to adopt a compatible stack frame shape. While these two issues signalled that a rewrite of the previous MicroJIT was necessary, we did borrow several aspects from its design.

First, we would continue to use methods as the unit of compilation for MicroJIT, as method-based compilation would provide us with the most straightforward path toward interoperability with the JVM and with any existing Testarossa functionality we might leverage. Second, when generating machine code for the body of a method, the previous MicroJIT would iterate, or walk the bytecodes, copying a snippet of machine code to the code cache for each bytecode it encountered. This mechanism for generating the body of a method is central to our design. Third, the previous MicroJIT added a field, called `extra2`, to each JVM method structure, which contains the invocation count and is used to trigger compilations for MicroJIT. We utilized the same mechanism for the new implementation of MicroJIT. When a method is first loaded, the field is initialized with the MicroJIT threshold subtracted from default JIT threshold. Similarly, Testarossa maintains an `extra` field for each method, which is initialized to the default JIT threshold. During execution, each time the interpreter invokes the method, the `extra` value is decremented. When the `extra` field is less than the `extra2` field, the method is ready for compilation with MicroJIT. For instance, if the JIT threshold was 3000 and the MicroJIT threshold was 100, then the `extra2` field would be initialized with 2900. After compilation with MicroJIT, the `extra2` field is overwritten with the start address of the generated code in the code cache. If the field contains an address the next time the method is invoked, the JITed code is executed instead.

Finally, the previous MicroJIT performed limited optimizations. There was minimal support for method inlining, as well as "fast-path" checks, designed to remove unnecessary context switching back to the interpreter between calls to JIT-compiled functions [68] [69]. For our implementation, initially, we will forgo any optimizations, including the inlining and fast-path checks.

## 4.2 Integrating MicroJIT and Eclipse OpenJ9

While it may be convenient to view MicroJIT 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 its debug and tracing options, mechanisms for asynchronous compilation, code cache management, stack-walking mechanisms, interrupt handling, as well as bytecode utilities. While we are currently targeting only Linux on x86-64, leveraging Testarossa's cross-platform functionality should ease any future development effort toward other platforms. By leveraging these facilities, we reduced our development effort considerably.

### 4.2.1 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.

### 4.2.2 Asynchronous Compilation

Having the ability to compile asynchronously is another valuable mechanism that was previously one of our goals during the port of the previous MicroJIT [17]. 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 to interpret the method. The compilation queue offers us the opportunity for increased parallelism, as we can now perform compilation on multiple threads. The compilation queue also

allows us 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.

### 4.2.3 Debugging Utilities

Another useful set of capabilities that this integration affords us is the rich set of tools for debugging and tracing provided by Eclipse OpenJ9. While developing MicroJIT, we need to focus on a small subset of methods, for which we can provide full bytecode coverage for, at any one time. As mentioned previously, TR provides command-line options to limit compilation to particular methods, which helped us focus our effort. 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 facilities.

### 4.2.4 Bytecode Iterator

Another useful utility Testarossa provided was the BytecodeIterator class. This class, which implements the iterator pattern [70], 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 associated mnemonics for both programming and debugging.[1]

### 4.2.5 Exceptions

The close integration of MicroJIT with TR also provides us with cross-platform support for interrupt handling. Through the Port library, provided through Eclipse

[1]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.

OMR, signal handlers are registered for JITed code. In the event of our JITed instructions generating 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.

### 4.2.6 Command-Line Options

Finally, in order to enable and configure MicroJIT within Eclipse OpenJ9, we added two 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.

## 4.3 Architecture

In the following sections, we look more closely at how selective compilation is performed, how code is generated and stored for MicroJIT, as well as discuss the target architecture for the generated code.

### 4.3.1 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 checked, and compilation is triggered when the counters have reached zero. As mentioned, MicroJIT adds a second field, `extra2`, to each method metadata structure `J9Method`. This field performs a similar role as 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 set up. Once the pre-compilation completes, the compilation proceeds to MicroJIT.

### 4.3.2 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. Improving this design is left for future work, though one option could be to store the generated code in a separate buffer and then allocate a segment of known length once the generation has completed. One downside to this approach is

the additional I/O required to copy the code to the segment, which could reduce the speed of the compilation.

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 8-bytes—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 the local array, we allocate 2-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 relinquishing 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 process is aborted, the code cache memory is freed, and a value is stored in `extra2` field to signal the method failed and should not be compiled by MicroJIT again. In such a case, future invocations of the method will continue to be performed by the Interpreter. 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 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.

### 4.3.3 Platform

Our first target platform for MicroJIT is Linux running on a 64-bit x86 architecture. The choice of this initial platform stems primarily from our previous work attempting to port MicroJIT to Linux from Windows while extending its instructions from 32-bit to 64-bit and our familiarity with both the platform and the architecture. In addition, in terms of constrained environments, the x86-64 architecture is common among edge-devices, or gateways [71], as well as for container-based systems operating in the cloud [72].

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.
- EAX, RAX, XMM0 - Return value registers. For a 32-bit integer, EAX is used, while RAX is used for 64-bit integers and XMM0 for float or double values.

- The return address will already be 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, 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: Stores the base pointer for the Java stack frame.
- R10: Stores the stack pointer for the Java stack.
- R11: Stores the accumulator, or stores a pointer to an object.
- R12: Stores any value that 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.

```
template_start iAddTemplate

mov r11, [r10]  ; pop first value off java stack into the accumulator
add r10, 8      ; which means reducing the stack size by 1 slot (8 bytes)
mov r12, [r10]  ; copy second value to the value register
add r11, r12    ; add the value to the accumulator
mov [r10], r11  ; write the accumulator over the second arg.

template_end iAddTemplate
```

Listing 4.1: Bytecode x86-64 Template for the `iadd` bytecode.

## 4.4 Bytecodes

The majority of the bytecodes we implemented map to a unique template written in NASM-style assembler[2] although several bytecodes, notably `load`, `store` and `ret` were handled generically. Listing 4.1 shows the assembly 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 template has been copied. 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.

### 4.4.1 Implementation Strategy

As of writing, we do not provide full bytecode coverage for MicroJIT. 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

[2]NASM, or Netwide Assembler, is the assembler used in the build process of OpenJ9 for Linux on x86-64. The assembly language, also called NASM, is line-based with full-support for Intel x86-64 (among others) and is designed for portability [73].

support for many important codes including `invokevirtual`, `new`, `invokespecial`, `newarray` and `athrow`. To guide our implementation, and to track our progress, we measured our bytecode coverage against the DaCapo Benchmark Suite version 9.12 [74] with `mjitCount=20` (see Table 4.1 for the results with `avrora`). The DaCapo benchmark suite was chosen as its programs provide a diverse range of functionality. The following list briefly describes five of the ten programs we analyzed.

- **Avrora** - Runs a number of programs on a simulated grid of AVR microcontrollers.
- **Eclipse** - Executes several non-graphical Java Development Tool tests in the Eclipse IDE.
- **PMD** - Runs source code analysis against several Java classes.
- **Sunflow** - Generates a raytrace-based render of an image.
- **Xalan** - Transforms an XML document to HTML.

Across all the benchmarks, `invokevirtual`, and `invokespecial` are the most common unsupported bytecodes.

Please see Appendix A for a full listing of the bytecodes as well as our progress.

### 4.4.2 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 [75] 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

| Bytecode | Count | % of Unsupported | % of Total |
|---|---|---|---|
| JBinvokevirtual | 2518 | 18.535 | 05.259 |
| JBputfield | 1647 | 12.124 | 03.440 |
| JBinvokespecial | 1530 | 11.262 | 03.196 |
| JBifeq | 773 | 05.690 | 01.615 |
| JBnewdup | 632 | 04.652 | 01.320 |
| JBldc | 575 | 04.233 | 01.201 |
| JBaconstnull | 366 | 02.694 | 00.764 |
| JBathrow | 351 | 02.584 | 00.733 |
| JBifnull | 338 | 02.488 | 00.706 |
| JBnop | 267 | 01.965 | 00.558 |
| **Total Bytecodes** | 47,877 | | |
| **Supported Bytecodes** | 34,292 | | |
| **Unsupported Bytecodes** | 13,585 | | |
| **Supported Methods** | 205 | | |
| **Unsupported Methods** | 1557 | | |

Table 4.1: The 10 most used unsupported bytecodes for `avrora`.

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.

In the next chapter we evaluate MicroJIT, comparing the performance of template-generated code to that of the interpreter.

# Chapter 5

# Evaluation

> *Science, my boy, is made up of mistakes, but they are mistakes which it is useful to make, because they lead little by little to the truth.*
>
> —Jules Verne, Journey to the Center of the Earth

This work aims to improve the throughput of constrained workloads when running with MicroJIT versus running in interpreter-only mode while minimizing any additional overhead. To this end, MicroJIT should perform compilations with less overhead—in terms of memory and computation time—than TRJIT would require. Considering that MicroJIT does not yet provide full bytecode coverage for the JVM specification and that it can only compile methods that it fully supports, in order to evaluate its performance, we concentrated on methods that it can compile.
With this in mind, our evaluation encompasses the following items:

- Selecting specific Java-based "microbenchmark" methods that MicroJIT can compile.
- Measuring the time required for the JIT compilation of the microbenchmarks with MicroJIT and with TRJIT.
- Measuring the memory required while JIT compiling the microbenchmarks with MicroJIT and with TRJIT.

- Measuring the execution time for the microbenchmarks over time.
- Gauging the throughput performance of MicroJIT over time when compared to the Interpreter and with TRJIT.

## 5.1 Microbenchmarks

For our microbenchmarks, we wrote two `static` methods to compute the *nth* number in the Fibonacci series iteratively and recursively. These methods were good candidates for microbenchmarks for several reasons. First, the algorithms should be familiar to people with a background in mathematics or computer science. Second, the algorithms are small, and as such, were good targets to implement bytecode support. Lastly, the two algorithms use two different computational approaches. The iterative style focuses on jumping within a single block of code, while the recursive style makes repeated calls to the same function until the base case is reached, and then the result is calculated as the stack frames unwind.

The first method, `IterativeFib.fib(I)I`[1], computes the series iteratively, relying on a `for` loop and temporary variables to calculate the result. Listing 5.1 shows the Java code for the method, while Listing 5.2 shows the bytecode for the method, disassembled using the command `javap -c IterativeFib.class`.

Anyone familiar with a high-level language such as C should be able to work through the Java code listing. The bytecode listing, on the other hand, is more challenging as it requires us to maintain a mental model of the operand stack. When the method begins, the integer method parameter is loaded onto the operand stack (`iload_0`), which is found in index 0 of the local array.[2] If the argument is equals to zero,

[1]The JVM specification defines shorthand symbols for the types of parameters and return values within the bytecode listings [9]. Here, `I` stands for Integer: the method accepts a single Integer parameter and returns an Integer result.

[2]For an instance method, the first parameter would instead be found in the first index of the local array and be loaded using `iload_1`, while index zero would contain the reference to the instance i.e. `this`.

```
1  public class IterativeFib {
2      public static int fib(int n) {
3
4          if(n == 0) {
5              return 0;
6          }
7
8          int x_1 = 0;
9          int x_2 = 1;
10         int temp = 0;
11
12         for(int i = 1; i < n; i++) {
13             temp = x_1 + x_2;
14             x_1 = x_2;
15             x_2 = temp;
16         }
17
18         return x_2;
19     }
20 }
```

Listing 5.1: Java class and method to iteratively compute the *nth* number of the Fibonacci series.

```
1  public static int fib(int);
2  Code:
3      0: iload_0
4      1: ifne          6
5      4: iconst_0
6      5: ireturn
7      6: iconst_0
8      7: istore_1
9      8: iconst_1
10     9: istore_2
11     10: iconst_0
12     11: istore_3
13     12: iconst_1
14     13: istore        4
15     15: iload         4
16     17: iload_0
17     18: if_icmpge     35
18     21: iload_1
19     22: iload_2
20     23: iadd
21     24: istore_3
22     25: iload_2
23     26: istore_1
24     27: iload_3
25     28: istore_2
26     29: iinc          4, 1
27     32: goto          15
28     35: iload_2
29     36: ireturn
```

Listing 5.2: Bytecode listing for `IterativeFib.fib(I)I`.

the method is done, and returns zero. This check is accomplished in the JVM by comparing the top element on the stack with zero (`ifne`): if it is not equal, jump to bytecode index 6, else continue and execute bytecode 4—zero is pushed onto the stack and the method returns.[3]

When MicroJIT compiles this method, it parses the bytecode stream, and for each instruction, copies a native code template into the code cache. As described in Section 4.3.2, for each bytecode, we store the bytecode index and the address in the code cache the template was copied to in a side table. For those instructions that have targets, such as conditional bytecodes and the `goto` bytecode, the template contains a 4-byte placeholder (`0xDEADBEEF`) for patching. When these instructions are encountered, we add an entry to the jump side table that contains the target bytecode index and the address in the code cache following the placeholder bytes. After all the bytecode templates have been generated, we use these two side tables to patch the placeholder bytes with the target addresses of all the generated branch or goto instructions.

Listing 5.3 shows the partial disassembly of the generated MicroJIT code for `IterativeFib.fib(I)I`. While this listing omits the preprologue and eplilogue, and is relatively dense reading, it is worth highlighting several lines:

- Line 1 - R10 stores our operand stack pointer. Here we are adding a slot onto the stack, which we then populate with a value from the local array (offset added to R10).
- Lines 7 and 8 - To eliminate possible side effects from intermingling 32-bit and 64-bit operands in our accumulator (R11) and value (R12) registers, we zero out the bits using `XOR` operations. In the future, we will investigate alternatives to this operation.

[3]The branching instructions (`ifne`, `if_icmpge`) in Listing 5.2 have the bytecode index as an operand.

```
0x7fffd6b40196: sub    r10,0x8
0x7fffd6b4019a: mov    r15,QWORD PTR [r14+0x20]
0x7fffd6b401a1: mov    QWORD PTR [r10],r15
0x7fffd6b401a4: sub    r10,0x8
0x7fffd6b401a8: mov    r15,QWORD PTR [r14+0x0]
0x7fffd6b401af: mov    QWORD PTR [r10],r15
0x7fffd6b401b2: xor    r11,r11
0x7fffd6b401b5: xor    r12,r12
0x7fffd6b401b8: mov    r12,QWORD PTR [r10]
0x7fffd6b401bb: add    r10,0x8
0x7fffd6b401bf: mov    r11,QWORD PTR [r10]
0x7fffd6b401c2: cmp    r11d,r12d
0x7fffd6b401c5: jge    0x7fffd6b40263
0x7fffd6b401cb: sub    r10,0x8
0x7fffd6b401cf: mov    r15,QWORD PTR [r14+0x8]
0x7fffd6b401d6: mov    QWORD PTR [r10],r15
0x7fffd6b401d9: sub    r10,0x8
0x7fffd6b401dd: mov    r15,QWORD PTR [r14+0x10]
0x7fffd6b401e4: mov    QWORD PTR [r10],r15
0x7fffd6b401e7: xor    r11,r11
0x7fffd6b401ea: xor    r12,r12
0x7fffd6b401ed: mov    r11,QWORD PTR [r10]
0x7fffd6b401f0: add    r10,0x8
0x7fffd6b401f4: mov    r12,QWORD PTR [r10]
0x7fffd6b401f7: add    r11d,r12d
0x7fffd6b401fa: mov    QWORD PTR [r10],r11
0x7fffd6b401fd: xor    r15,r15
0x7fffd6b40200: mov    r15,QWORD PTR [r10]
0x7fffd6b40203: add    r10,0x8
0x7fffd6b40207: mov    QWORD PTR [r14+0x18],r15
0x7fffd6b4020e: sub    r10,0x8
0x7fffd6b40212: mov    r15,QWORD PTR [r14+0x10]
0x7fffd6b40219: mov    QWORD PTR [r10],r15
0x7fffd6b4021c: xor    r15,r15
0x7fffd6b4021f: mov    r15,QWORD PTR [r10]
0x7fffd6b40222: add    r10,0x8
0x7fffd6b40226: mov    QWORD PTR [r14+0x8],r15
0x7fffd6b4022d: sub    r10,0x8
0x7fffd6b40231: mov    r15,QWORD PTR [r14+0x18]
0x7fffd6b40238: mov    QWORD PTR [r10],r15
0x7fffd6b4023b: xor    r15,r15
0x7fffd6b4023e: mov    r15,QWORD PTR [r10]
0x7fffd6b40241: add    r10,0x8
0x7fffd6b40245: mov    QWORD PTR [r14+0x10],r15
0x7fffd6b4024c: mov    r11,QWORD PTR [r14+0x20]
0x7fffd6b40253: add    r11,0x1
0x7fffd6b40257: mov    QWORD PTR [r14+0x20],r11
0x7fffd6b4025e: jmp    0x7fffd6b40196
0x7fffd6b40263: sub    r10,0x8
0x7fffd6b40267: mov    r15,QWORD PTR [r14+0x10]
0x7fffd6b4026e: mov    QWORD PTR [r10],r15
0x7fffd6b40271: mov    rbx,QWORD PTR [rsp+0x88]
0x7fffd6b40279: mov    r9,QWORD PTR [rsp+0x80]
0x7fffd6b40281: mov    rbp,QWORD PTR [rsp+0x78]
0x7fffd6b40289: mov    rsp,QWORD PTR [rsp+0x70]
0x7fffd6b40291: add    rsp,0x90
0x7fffd6b40298: mov    rax,QWORD PTR [r10]
0x7fffd6b4029b: ret
```

Listing 5.3: Partial MicroJIT disassembly listing for `IterativeFib.fib(I)I`.

- Line 13 - We compare our loop induction variable with our target value (30) and jump to `0x7fffd6b40263` (line 49) if the value is greater or equal.
- Line 48 - Continue iterating by jumping back to the start of the loop.
- Line 49 - Restore the registers, place the result in RAX, and return.

The second method, `RecursiveFib.fib(I)I`, computes the series recursively, relying on the invocation of itself to calculate the result. Listing 5.4 shows the Java code for the method, Listing 5.5 shows the bytecodes for the method, and Listing 5.6 shows the partial disassembly of the generated MicroJIT code. Several lines of assembly worth noting are:

- Line 6 - Load the base-case (1) onto the operand stack for a comparison (`jg` on line 13).
- Line 13 - If the value is less or equal to the base-case, our registers are restored, the stack pointer is restored, and the result is placed in RAX before returning.
- Line 24 - If the value is greater than the base-case then work continues here. Setup is performed for the recursive call by subtracting 1 from the argument (lines 28 and 34), and preserving registers.
- Line 46 - In order to make the `invokeStatic` call, we call an assembly helper `interpreterStaticAndSpecialGlue`, which redirects us to the proper target for the call: either JITed code from TR or MicroJIT, or a return to the Interpreter.

The evaluations were all performed on an embedded board with an Intel Atom 1.44 GHz x5-Z8350 4-core processor, 4 GB of DDR3 RAM, 64 GB of eMMC storage, running Debian Linux 5.6.7-1 in headless mode[4] [76] [77]. We selected this hardware

[4]Headless mode means the operating system provides only terminal-based access without a Graphical User Interface.

```
public class RecursiveFib {
    public static int fib(int n){
        if(n <= 1) {
            return n;
        }
        return RecursiveFib.fib(n - 1) + RecursiveFib.fib(n - 2);
    }
}
```

Listing 5.4: Java class and method to recursively compute the *nth* number of the Fibonacci series.

```
public static int fib(int);
Code:
    0: iload_0
    1: iconst_1
    2: if_icmpgt     7
    5: iload_0
    6: ireturn
    7: iload_0
    8: iconst_1
    9: isub
    10: invokestatic  #8                  // Method fib:(I)I
    13: iload_0
    14: iconst_2
    15: isub
    16: invokestatic  #8                  // Method fib:(I)I
    19: iadd
    20: ireturn
```

Listing 5.5: Bytecode listing for `RecursiveFib.fib(I)I`.

as it provided us with a good approximation of an edge, or a gateway-class IoT device. To disable processor throttling due to power consumption, the scaling governors for the CPU cores were all set to "performance" mode, allowing the cores to run at their maximum speed.

## 5.2 Compilation Time

First, we will look at the time spent compiling the two microbenchmark 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 compil-

```
0x7fffd6b400c8: mov    QWORD PTR [r14+0x0],rax
0x7fffd6b400cf: sub    r10,0x8
0x7fffd6b400d3: mov    r15,QWORD PTR [r14+0x0]
0x7fffd6b400da: mov    QWORD PTR [r10],r15
0x7fffd6b400dd: sub    r10,0x8
0x7fffd6b400e1: mov    DWORD PTR [r10],0x1
0x7fffd6b400e8: xor    r11,r11
0x7fffd6b400eb: xor    r12,r12
0x7fffd6b400ee: mov    r12,QWORD PTR [r10]
0x7fffd6b400f1: add    r10,0x8
0x7fffd6b400f5: mov    r11,QWORD PTR [r10]
0x7fffd6b400f8: cmp    r11d,r12d
0x7fffd6b400fb: jg     0x7fffd6b4013a
0x7fffd6b40101: sub    r10,0x8
0x7fffd6b40105: mov    r15,QWORD PTR [r14+0x0]
0x7fffd6b4010c: mov    QWORD PTR [r10],r15
0x7fffd6b4010f: mov    rbx,QWORD PTR [rsp+0x70]
0x7fffd6b40117: mov    r9,QWORD PTR [rsp+0x68]
0x7fffd6b4011f: mov    rbp,QWORD PTR [rsp+0x60]
0x7fffd6b40127: mov    rsp,QWORD PTR [rsp+0x58]
0x7fffd6b4012f: add    rsp,0x78
0x7fffd6b40136: mov    rax,QWORD PTR [r10]
0x7fffd6b40139: ret
0x7fffd6b4013a: sub    r10,0x8
0x7fffd6b4013e: mov    r15,QWORD PTR [r14+0x0]
0x7fffd6b40145: mov    QWORD PTR [r10],r15
0x7fffd6b40148: sub    r10,0x8
0x7fffd6b4014c: mov    DWORD PTR [r10],0x1
0x7fffd6b40153: xor    r11,r11
0x7fffd6b40156: xor    r12,r12
0x7fffd6b40159: mov    r12,QWORD PTR [r10]
0x7fffd6b4015c: add    r10,0x8
0x7fffd6b40160: mov    r11,QWORD PTR [r10]
0x7fffd6b40163: sub    r11d,r12d
0x7fffd6b40166: mov    QWORD PTR [r10],r11
0x7fffd6b40169: xor    rax,rax
0x7fffd6b4016c: mov    rax,QWORD PTR [r10]
0x7fffd6b4016f: add    r10,0x8
0x7fffd6b40173: mov    QWORD PTR [rsp+0x8],r10
0x7fffd6b4017b: mov    QWORD PTR [rsp+0x10],r11
0x7fffd6b40183: mov    QWORD PTR [rsp+0x18],r12
0x7fffd6b4018b: mov    QWORD PTR [rsp+0x20],r13
0x7fffd6b40193: mov    QWORD PTR [rsp+0x28],r14
0x7fffd6b4019b: mov    QWORD PTR [rsp+0x30],r15
0x7fffd6b401a3: movabs rdi,0x182798
0x7fffd6b401ad: call   0x7ffff5dc7170 <interpreterStaticAndSpecialGlue>
```

Listing 5.6: Partial MicroJIT disassembly listing for `RecursiveFib.fib(I)I`.

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

Table 5.1: Compilation times (in microseconds) for microbenchmark methods.

ing just the single microbenchmark method. The Java programs running the microbenchmarks used a simple `for` loop to ensure that the methods had enough invocations to trigger the JIT compilations. Finally, the times for the compilations were extracted from verbose logs provided by Eclipse OpenJ9 using the following option: `-Xjit:verbose=compilePerformance,vlog=vlogfile`. The times were extracted from the verbose log files using a Perl script and later aggregated with the `statistics` module using Python 3.7 [78]. Table 5.1 shows that by eliminating the IL phase, MicroJIT was able to compile the methods roughly 2.3 times faster than TRJIT with the noOpt compilation level (TR-noOpt), and compile `RecursiveFib.fib(I)I` almost 5 times faster than TR with the cold compilation level (TR-cold). Considering that we are copying machine-code templates instead of building and transforming an intermediate form—what we expect would be much less work—this result is both expected and promising.

## 5.3 Memory Consumption

Next, we consider the amount of heap-based memory required during the JIT compilation. For MicroJIT, we expect to see less memory used than TR as we are not building, or operating on any IL-trees. Using the same verbose log results that were generated in the previous experiment, Table 5.2 shows the memory in KB used by JIT compilers during compilation. We see that MicroJIT uses a fraction of the memory that either TR-noOpt or TR-cold use: it can compile the microbenchmark 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

| | MicroJIT | TR-noOpt | TR-cold |
|---|---|---|---|
| Iterative-fib | 64 | 1024 | 960 |
| Recursive-fib | 64 | 960 | 1280 |

Table 5.2: Memory (in Kilobytes) for a single JIT Compilation.

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. In the design of MicroJIT, we intentionally relied on stack-based allocations wherever possible. Thus when the primary MicroJIT function, `mjit()`, returns, any stack-based structures would be freed.[5] While these numbers report the amount of heap-based memory allocated for a single compilation, they do not take into account the amount of memory used by our stack frames. It is also worth noting that the memory required did not change across executions of the benchmark processes, thus we did not need to report the mean or standard deviation.

## 5.4 Execution Time

Next, we consider the time required to execute each microbenchmark. Initially, we expect the JITed results to be roughly the same as the Interpreter results, then

[5]The term freed here is a misnomer as no `free` operation actually occurs when the function returns. Instead, the old stack frame memory will be overwritten by subsequent function calls.

slower than the Interpreter as less computational resources are available during JIT compilation, and then finally faster than the Interpreter as the invocations are able to execute native code instead. The test programs invoked the microbenchmarks 1000 times, while printing the number of elapsed nanoseconds for each call before looping. These test programs were executed using 100 fresh JVM executions for each of the following: Interpreter, MicroJIT, TR-noOpt, and TR-cold. Figure 5.1 shows the time for the first 50 invocations of `IterativeFib.fib(30)`. 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 5.2 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 TRcold——the interpreter did collect profile information, which is then used later to inform the JIT compilations.

Figure 5.3 shows the execution of `IterativeFib.fib(30)` after 3000 invocations. At this point, we see that both TR-noOpt and TR-cold are faster than MicroJIT, while all three JITed methods are significantly faster than the Interpreter.

Figure 5.4 paints a similar picture for `RecursiveFib.fib(10)`, however, in this case, the gulf between the JITed methods and the Interpreter is much larger. The performance of this microbenchmark is dominated by the overhead of the `static` method calls, which, for the Interpreter, requires additional steps to find and call the target. The gulf between the JITed methods is also significantly larger, as MicroJIT is less optimized for making the recursive calls.

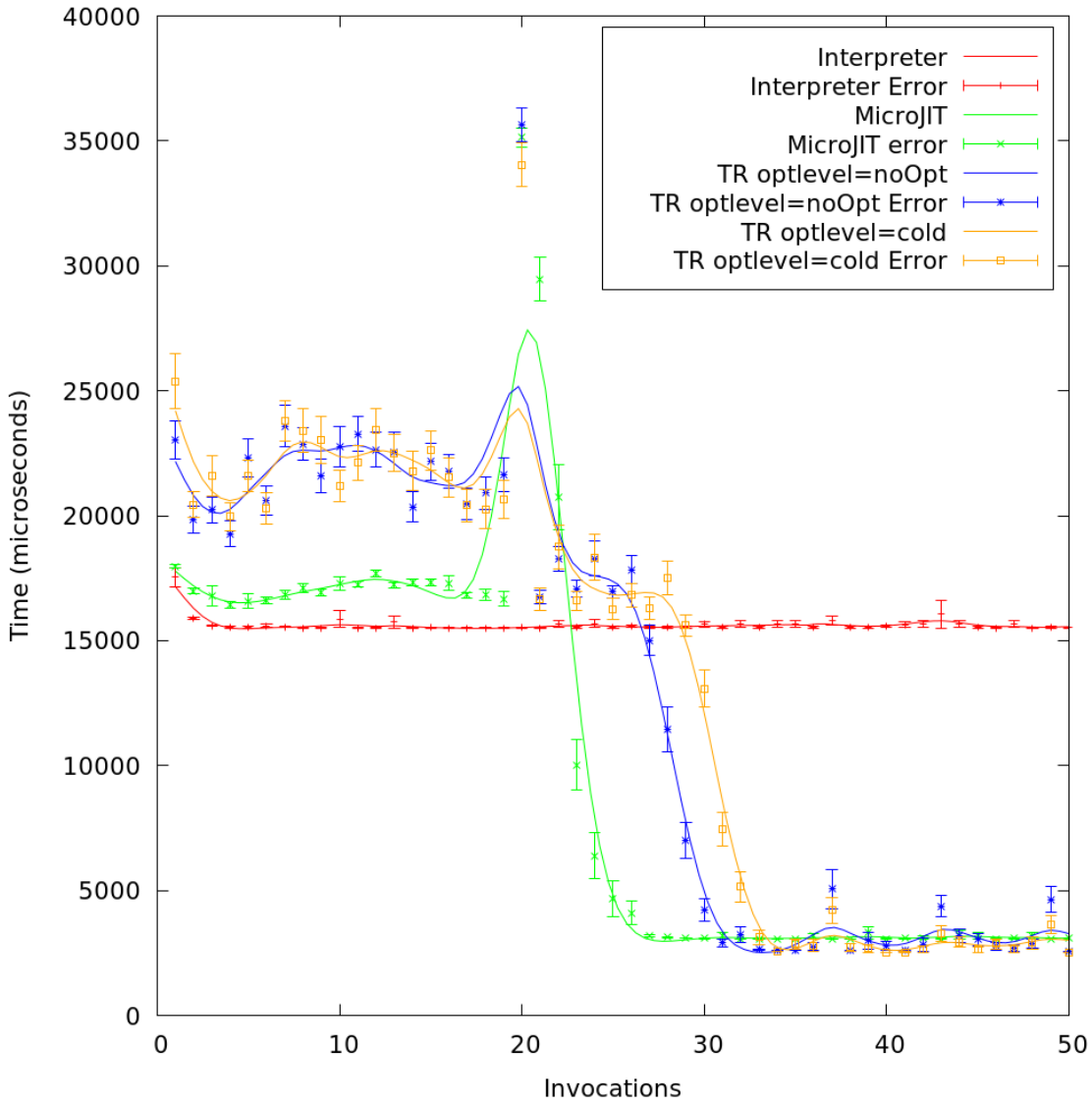


Figure 5.1: Comparing the execution time of the first 50 iterations of the microbenchmark IterativeFib.fib(30).

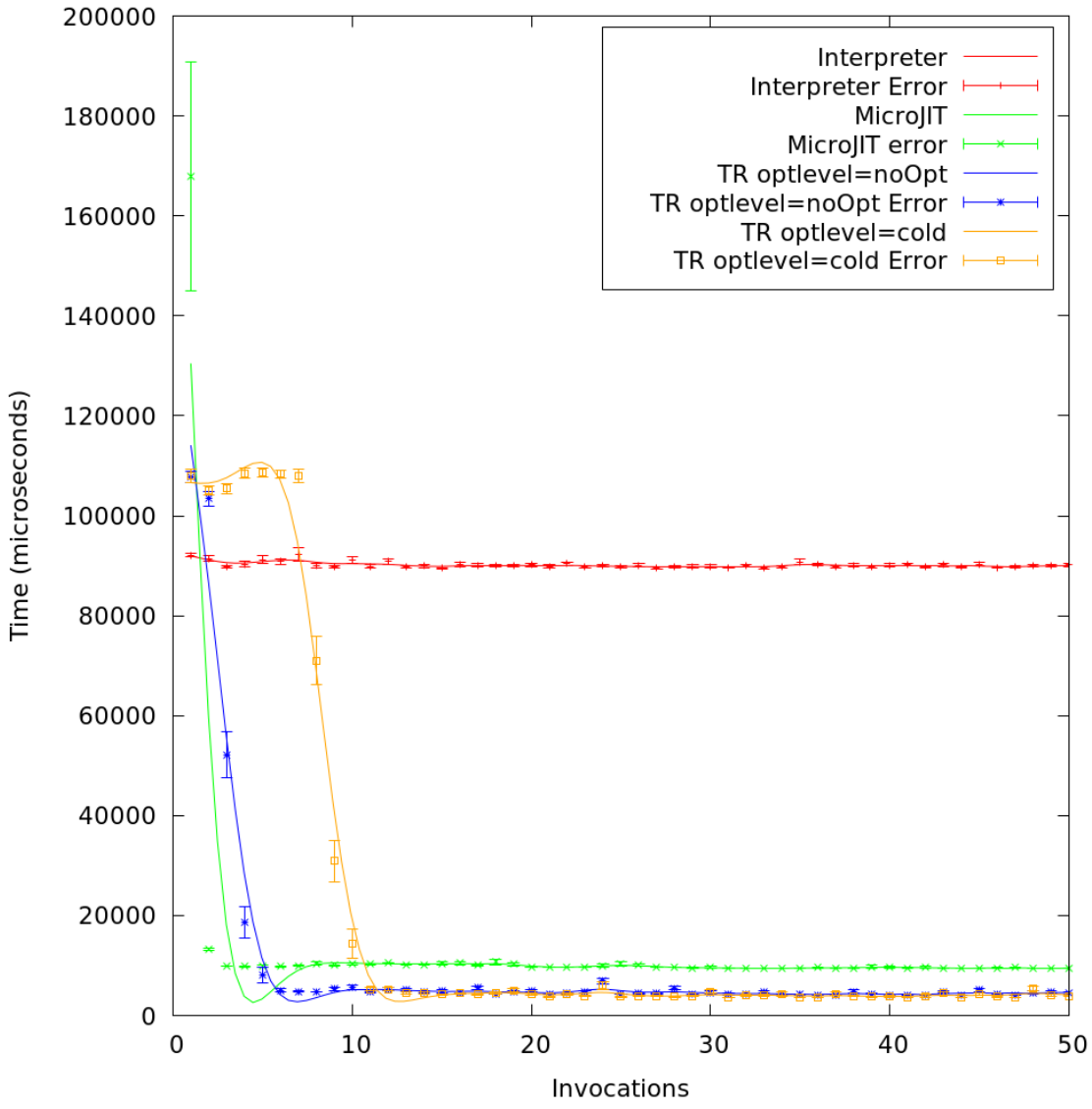


Figure 5.2: Comparing the execution time of the first 50 iterations of the microbenchmark RecursiveFib.fib(10).

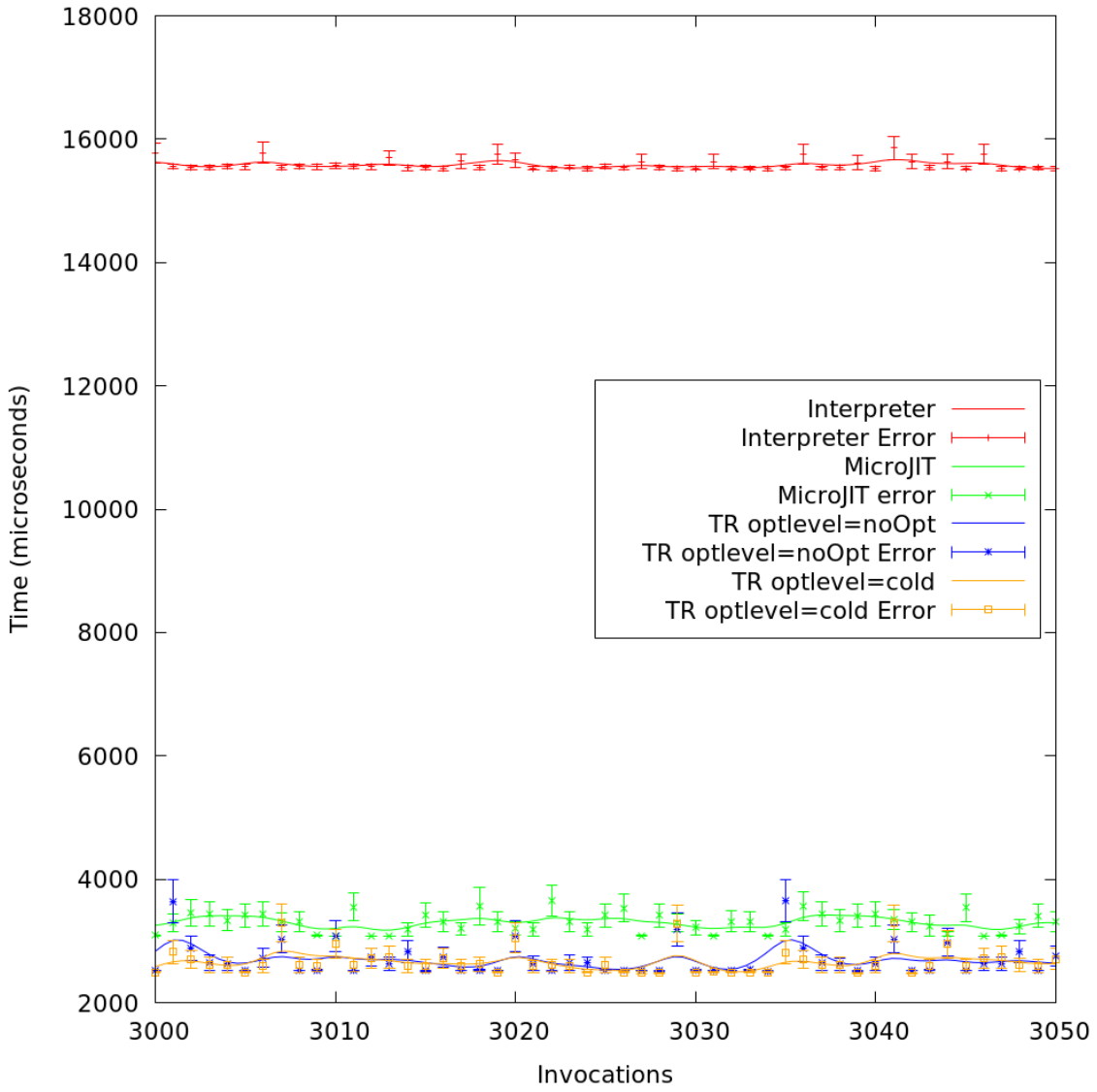


Figure 5.3: Comparing the execution time of microbenchmark IterativeFib.fib(30) after 3000 iterations.

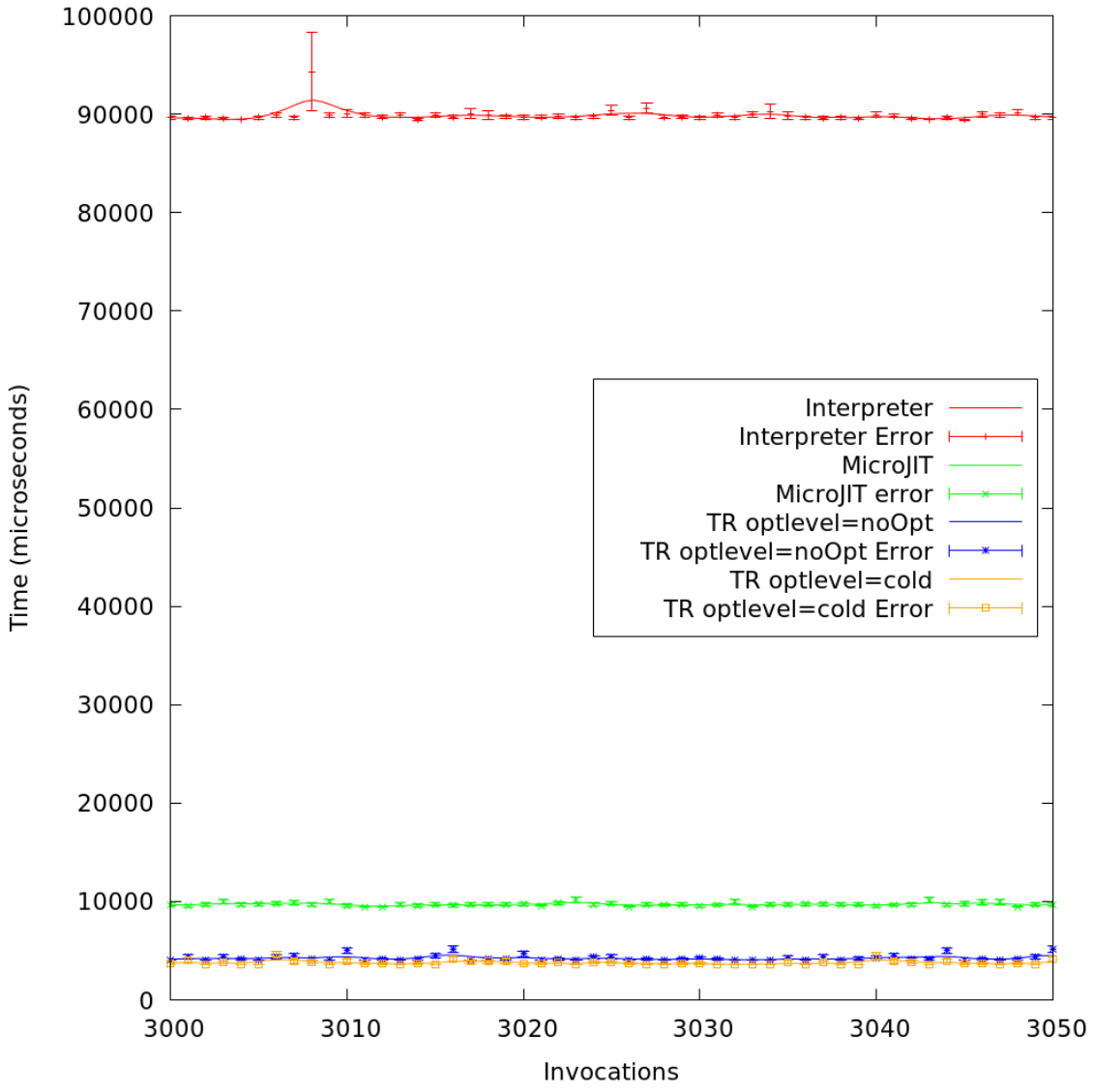


Figure 5.4: Comparing the execution time of microbenchmark RecursiveFib.fib(10) after 3000 iterations.

## 5.5 Throughput

Finally, to measure throughput, or operations per second, we executed the benchmarks 1 million times and measured the elapsed time. We expect that MicroJIT will have higher througput than the Interpreter, but have lower throughput than either of the TR optimization levels. This experiment was repeated 100 times for each of the Interpreter, MicroJIT, and for TR-noOpt TR-cold, while the JVM was restarted between executions (400 individual executions in total). Tables 5.3 and 5.4 show the throughput of 1 million invocations for `IterativeFib.fib(30)` and `RecursiveFib.fib(10)` respectively. The % column shows the percentage of time spent relative to the Interpreter. In Table 5.3, we see that MicroJIT can complete 1 million calculations in 1.38 seconds, or 9.55% of the 14.426 seconds 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. Recall that, despite its name, TR-noOpt performs tree simplification and the code generation may also include low-level optimizations.

In Table 5.4, 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. We can also see that significant improvement in throughput is achieved by TR and its IL-phase. While MicroJIT can 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 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

| | Time (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 5.3: Time to execute 1 million invocations of `IterativeFib.fib(30)`. We ran the experiment 100 times. The % column shows the percent of time spent relative to the interpreter. MicroJIT is able to complete 1 million invocations in 9.55% of the time it takes the Interpreter.

| | Time (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 5.4: Time to execute 1 million invocations of `RecursiveFib.fib(10)`. While MicroJIT is able to complete the task in 8.69% of the time required by the Interpreter, the code generated by TR is significantly faster for this benchmark.

build may eliminate it. Looking at the quality of the generated code, TR-noOpt has fewer instructions than MicroJIT (28 versus 115), and for `IterativeFib`, the numbers are similar (36 versus 116)[6]. 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 investigate mechanisms to reduce the overhead when invoking static methods.

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

# Chapter 6

# Future Work

*The more you look, the more you see.*

—Rober Pirsig, Zen and the Art of Motorcycle Maintenance

While we hope this work has shown the promise of integrating a template-based JIT compiler into a modern Java Virtual Machine for constrained environments, much work remains. Primary amongst this is the need to extend and eventually complete the bytecode support for MicroJIT. While MicroJIT can operate without full bytecode coverage, until support is added for common bytecodes such as `new`, `invokespecial`, `invokevirtual`, and `throw`, the usefulness of the compiler will remain limited. Once complete bytecode coverage is implemented, several other avenues of future exploration emerge, including adding support for a MicroJIT-only mode, extending support for MicroJIT to other architectures, investigating inexpensive optimizations, and finally using MicroJIT in tandem with TRJIT to reduce startup time while adding profiling instructions to MicroJIT-generated code.

## 6.1 MicroJIT-only Mode

Currently, when MicroJIT is included in the build for Eclipse OpenJ9, it operates alongside TRJIT. In the future, we would like to investigate providing an additional

build flag to replace the default TR compiler with MicroJIT. When MicroJIT is included, the `extra2` field, a 64-bit unsigned integer, is part of every `J9Method` metadata structure. Considering a Java project could have many thousands of classes and many more associated methods, storing an additional 8-bytes for each method could add significant overhead. Instead, MicroJIT would use the standard `extra` field. Any additional overhead required to check if the callee is compiled by TR or MicroJIT when transitioning between JIT code could also be eliminated. An alternative approach would be to add MicroJIT as an optimization level to TR that is even lower than `noOpt`. This optimization level, called `template`, would tightly integrate MicroJIT with the rest of Testarossa, and the existing command-line option `optLevel=<level>` would provide for the MicroJIT-only mode.

## 6.2 Extending Architectural Support

Considering that a significant portion of the world's IOT devices are now based on low-powered, ARM architecture [71], extending the architecturual support for MicroJIT beyond x86-64 to AArch64 could be interesting future work. While ARM has a RISC-based[1] instruction set architecture (ISA), x86 has a CISC-based[2] ISA [11]. While the RISC versus CISC debate has historically sided with ARM for low power consumption and with x86 for pure throughput, recent work has shown that the architectures are now even with respect to these when operating on modern workloads [79]. Given the nature of the ARM ISA, we would expect the number of generated native instructions to be relatively large compared to x86. Also, given its large number of registers, there may be additional opportunities involving register optimizations.

[1]Reduced Instruction Set Computing - the instruction set is relatively simple, primarily register-based, with independent load and store commands

[2]Complex Instruction Set Computer - the instruction set is relatively complex—capable of multiple operations, including addressing memory, within a single instruction

## 6.3 MicroJIT and TRJIT

While this work has focused on the scenario where MicroJIT is the only JIT compiler in Eclipse OpenJ9, there is future work to investigate how MicroJIT might work alongside TRJIT to provide inexpensive, fast compilation earlier than the `noOpt` optimization level. For less-constrained environments, such as servers, where the startup time, or the time to serve the first request is a priority, MicroJIT, as previously pointed out by Sogaro et al. [16], could, provided a lower invocation threshold than TR, decrease startup time. One of the goals from our last work—porting the MicroJIT to OpenJ9 [17]—was adding profiling instructions to our generated code. Without these profiling instructions, the higher quality code eventually generated by TRJIT would need to execute for longer in order to generate enough profile data to help with further optimizations.

## 6.4 Support for Larger Generated Code

As mentioned is Section 4.3.2, we limited the size of our generated code to 1024 bytes. Considering that each method contains generated code for the preprologue, prologue, the epilogue, this may not leave much space to generate the method body. Some work remains to choose an appropriate default size, or to design an alternative mechanism for handling scenarios where the code generated exceeds this amount.

# Chapter 7

# Conclusion

The rise of IoT devices and container-based microservices has renewed the need for virtual machines and their workloads to operate efficiently and effectively in constrained environments. For some of these workloads, the overhead of an optimizing JIT compiler may be too high to be effective, yet having access to JIT compilation may still be necessary for performance. In this work, we propose that our template-based JIT compiler, MicroJIT, could help satisfy this need. With our limited bytecode support and bespoke microbenchmarks, we have shown that both JIT compilation time and memory overhead can be reduced on Eclipse OpenJ9 and that significant improvements over the baseline performance of the Interpreter can be achieved. With the promise shown by these early results, we will continue the work to complete our bytecode support. In the future, our goals will be to provide standard benchmark performance results, while eventually expanding our effort to include multiple target platforms and architectures.

# Bibliography


[1] A. C. Doyle, *The Memoirs of Sherlock Holmes*. Oxford University Press, 1993.

[2] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of Things (IoT): A vision, architectural elements, and future directions," *Future generation computer systems*, vol. 29, no. 7, pp. 1645–1660, 2013.

[3] Laurence Goasduff, "Gartner Says 5.8 Billion Enterprise and Automotive IoT Endpoints Will Be in Use in 2020." `https://www.gartner.com/en/newsroom/press-releases/2019-08-29-gartner-says-5-8-billion-enterprise-and-automotive-io`, Last accessed on 2020-02-25.

[4] Eclipse Foundation Inc., "IoT Developer Survey 2019 Results." `https://iot.eclipse.org/community/resources/iot-surveys/assets/iot-developer-survey-2019.pdf`, Last accessed on 2020-04-29.

[5] TIOBE - The software quality company, "The Java Programming Language," 2019. `https://www.tiobe.com/tiobe-index/java/`, Last accessed on 2019-06-17.

[6] D. Simon, C. Cifuentes, D. Cleal, J. Daniels, and D. White, "Java™ on the Bare Metal of Wireless Sensor Devices: the Squawk Java Virtual Machine," in *Proceedings of the 2nd international conference on Virtual execution environments*, pp. 78–88, 2006.

[7] N. Brouwers, K. Langendoen, and P. Corke, "Darjeeling, a Feature-Rich VM for the Resource Poor," in *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, SenSys '09, (New York, NY, USA), p. 169–182, Association for Computing Machinery, 2009.

[8] N. Reijers and C.-S. Shih, "CapeVM: A Safe and Fast Virtual Machine for Resource-Constrained Internet-of-Things Devices," in *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*, SenSys '18, (New York, NY, USA), p. 250–263, Association for Computing Machinery, 2018.

[9] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *The Java Virtual Machine Specification, Java SE 8 Edition*. Addison-Wesley Professional, 1st ed., 2014.

[10] J. Smith and R. Nair, *Virtual machines: versatile platforms for systems and processes*. Elsevier, 2005.

[11] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[12] J. Aycock, "A Brief History of Just-in-Time," *ACM Computing Surveys*, vol. 35, pp. 97–113, June 2003.

[13] L. P. Deutsch and A. M. Schiffman, "Efficient Implementation of the Smalltalk-80 System," in *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '84, (New York, NY, USA), p. 297–302, Association for Computing Machinery, 1984.

[14] R. Jones, A. Hosking, and E. Moss, *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman and Hall/CRC, 2016.

[15] M. Gaudet and M. Stoodley, "Rebuilding an airliner in flight: a retrospective on refactoring IBM testarossa production compiler for Eclipse OMR," pp. 24–27, 10 2016.

[16] F. Sogaro, E. Aubanel, K. B. Kent, V. Sundaresan, M. Pirvu, and P. Shipton, "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, (Markham, Ontario, Canada), pp. 140–150, IBM Corp., 2017.

[17] E. Coffin, S. Young, K. B. Kent, and M. Pirvu, "A Roadmap for Extending MicroJIT: A Lightweight Just-in-Time Compiler for Decreasing Startup Time," in *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*, CASCON '19, (Markham, Ontario, Canada), p. 293–298, IBM Corp., 2019.

[18] TIOBE - The software quality company, "The C# Programming Language," 2019. `https://www.tiobe.com/tiobe-index/csharp/`, Last accessed on 2019-06-17.

[19] Genevieve Warren, "Managed Code," 2020. `https://docs.microsoft.com/en-us/dotnet/standard/managed-code`, Last Accessed on 2020-04-08.

[20] J. Gosling, B. Joy, G. L. Steele, G. Bracha, and A. Buckley, *The Java Language Specification, Java SE 8 Edition*. Addison-Wesley Professional, 1st ed., 2014.

[21] ComputerWeekly.com, "Write once, run anywhere," 2002. `https://www.computerweekly.com/feature/Write-once-run-anywhere/`, Last accessed on 2019-06-17.

[22] C. Kelleher and R. Pausch, "Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers," *ACM Computing Surveys (CSUR)*, vol. 37, no. 2, pp. 83–137, 2005.

[23] Eclipse, "Eclipse OpenJ9 Github," 2020. `https://github.com/eclipse/openj9`, Last accessed on 2020-05-07.

[24] Oracle Corporation, "Java SE HotSpot at a Glance." https://www.oracle.com/technetwork/java/javase/tech/index-jsp-136373.html, Last Accessed on 2020-03-30.

[25] Azul Systems, Inc., "Zing runtime for Java." https://www.azul.com/products/zing/, Last Accessed on 2020-03-30.

[26] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger, "An overview of the scala programming language," tech. rep., 2004.

[27] Kotlin Foundation, "Kotlin Programming Language," 2020. =https://kotlinlang.org, Last accessed on 2020-07-03.

[28] Rich Hickey , "Clojure." https://clojure.org/,Last Accessed on 2020-07-11.

[29] I. H. Kazi, H. H. Chen, B. Stanley, and D. J. Lilja, "Techniques for Obtaining High Performance in Java Programs," *ACM Comput. Surv.*, vol. 32, pp. 213–240, Sept. 2000.

[30] B. R. Rau, "Levels of Representation of Programs and the Architecture of Universal Host Machines," *SIGMICRO Newsl.*, vol. 9, p. 67–79, Nov. 1978.

[31] M. A. Ertl and D. Gregg, "The structure and performance of efficient interpreters," *Journal of Instruction-Level Parallelism*, vol. 5, pp. 1–25, 2003.

[32] P. A. Kulkarni, "JIT compilation policy for modern machines," in *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, pp. 773–788, 2011.

[33] G. J. Hansen, "Adaptive systems for the dynamic run-time optimization of programs.," 1974.

[34] K. Hazelwood and M. D. Smith, "Code cache management schemes for dynamic optimizers," in *Proceedings Sixth Annual Workshop on Interaction between Compilers and Computer Architectures*, pp. 102–102, Citeseer, 2002.

[35] G. Manjunath and V. Krishnan, "A small hybrid JIT for embedded systems," *ACM SIGPLAN Notices*, vol. 35, no. 4, pp. 44–50, 2000.

[36] H. Inoue, H. Hayashizaki, P. Wu, and T. Nakatani, "Adaptive Multi-Level Compilation in a Trace-Based Java JIT Compiler," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, (New York, NY, USA), p. 179–194, Association for Computing Machinery, 2012.

[37] A. Gal, C. W. Probst, and M. Franz, "HotpathVM: an effective JIT compiler for resource-constrained devices," in *Proceedings of the 2nd international conference on Virtual execution environments*, pp. 144–153, 2006.

[38] U. Hölzle and D. Ungar, "A third-generation SELF implementation: reconciling responsiveness with performance," in *Proceedings of the ninth annual conference on Object-oriented programming systems, language, and applications*, pp. 229–243, 1994.

[39] S. J. Fink and F. Qian, "Design, implementation and evaluation of adaptive recompilation with on-stack replacement," in *International Symposium on Code Generation and Optimization, 2003. CGO 2003.*, pp. 241–252, IEEE, 2003.

[40] J. Brock, C. Ding, X. Xu, and Y. Zhang, "PAYJIT: space-optimal JIT compilation and its practical implementation," in *Proceedings of the 27th International Conference on Compiler Construction*, pp. 71–81, 2018.

[41] U. Hölzle and D. Ungar, "Reconciling Responsiveness with Performance in Pure Object-Oriented Languages," *ACM Trans. Program. Lang. Syst.*, vol. 18, no. 4, p. 355–400, 1996.

[42] K. Kennedy and J. R. Allen, *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., 2001.

[43] K. Cooper and L. Torczon, *Engineering a Compiler*. Elsevier, 2011.

[44] W. M. McKeeman, "Peephole optimization," *Communications of the ACM*, vol. 8, no. 7, pp. 443–444, 1965.

[45] C. Chambers and D. Ungar, "Making pure object-oriented languages practical," in *Conference proceedings on Object-oriented programming systems, languages, and applications*, pp. 1–15, 1991.

[46] U. Hölzle, C. Chambers, and D. Ungar, "Optimizing dynamically-typed object-oriented languages with polymorphic inline caches," in *European Conference on Object-Oriented Programming*, pp. 21–38, Springer, 1991.

[47] P. Guide, "Intel® 64 and ia-32 architectures software developer's manual," *Volume 3B: System programming Guide, Part*, vol. 2, p. 11, 2011.

[48] D. A. Patterson and C. H. Sequin, "RISC I: A Reduced Instruction Set VLSI Computer," in *Proceedings of the 8th Annual Symposium on Computer Architecture*, ISCA '81, (Washington, DC, USA), p. 443–457, IEEE Computer Society Press, 1981.

[49] AdoptOpenJDK, "Prebuilt OpenJDK Binaries for Free!," 2020. `https://adoptopenjdk.net/`, Last accessed on 2020-02-27.

[50] D. Maier and X. Liang, "Supercharge a Language Runtime!," in *Proceedings of the 27th Annual International Conference on Computer Science and Software*

*Engineering*, CASCON '17, (Markham, Ontario, Canada), p. 314, IBM Corp., 2017.

[51] 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*, vol. 39, no. 1, pp. 175–193, 2000.

[52] R. N. Sanchez, J. N. Amaral, D. Szafron, M. Pirvu, and M. Stoodley, "Using machines to learn method-specific compilation strategies," in *International Symposium on Code Generation and Optimization (CGO 2011)*, pp. 257–266, IEEE, 2011.

[53] Eclipse OpenJ9, "The JIT Compiler." `https://www.eclipse.org/openj9/docs/jit/`, Last Accessed on 2020-04-20 .

[54] Eclipse OMR, "OMR Command-line Options." `https://github.com/eclipse/omr/blob/master/compiler/control/OMROptions.cpp`, Last Accessed on 2020-04-19.

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

[56] CISCO, "IOT Gateways." `https://www.cisco.com/c/en_ca/solutions/internet-of-things/iot-gateways.html`, Last Accessed on 2020-04-24 .

[57] D. Detlefs and O. Agesen, "The case for multiple compilers," in *OOPSLA*, vol. 99, pp. 180–194, 1999.

[58] Standard Performance Evaluation Corporation, "SPEC JVM98 Benchmarks." `https://www.spec.org/jvm98/`, Last accessed on 2019-06-19.

[59] WebAssembly, "WebAssembly High-Level Goals," 2020. https://webassembly.org/, Last Accessed on 2002-06-20.

[60] Clemens Hammacher, "Liftoff: a new baseline compiler for WebAssembly in V8," 2018. https://v8.dev/blog/liftoff, Last accessed on 2020-06-10.

[61] Ben Corrie, Hang Shao, "Class sharing in Eclipse OpenJ9," 2018. https://developer.ibm.com/tutorials/j-class-sharing-openj9/, Last accessed on 2019-06-19.

[62] Marius Pirvu, "Optimize JVM start-up with Eclipse OpenJ9," 2018. (https://developer.ibm.com/articles/optimize-jvm-startup-with-eclipse-openjj9/, Last accessed on 2019-06-19.

[63] M. Thom, G. W. Dueck, K. Kent, and D. Maier, "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, (Markham, Ontario, Canada), pp. 275–281, IBM Corp., 2018.

[64] Docker Inc., "Docker Image Build." https://docs.docker.com/engine/reference/commandline/image_build/, Last Accessed on 2020-06-22.

[65] Oracle, "GraalVM Native Image." https://www.graalvm.org/docs/reference-manual/native-image/, Last Accessed on 2020-06-22 .

[66] C. Wimmer, C. Stancu, P. Hofer, V. Jovanovic, P. Wögerer, P. B. Kessler, O. Pliss, and T. Würthinger, "Initialize Once, Start Fast: Application Initialization at Build Time," *Proc. ACM Program. Lang.*, vol. 3, Oct. 2019.

[67] Eclipse OpenJ9 Interpreter Actions, "BytecodeInterpreter.hpp," 2020. https://github.com/eclipse/openj9/blob/master/runtime/vm/BytecodeInterpreter.hpp, Last accessed on 2020-05-07.

[68] Kent, Kenneth B., and Serra, Micaela , "Context Switching in a Hardware/-Software Co-Design of the Java Virtual Machine," *Forum of Design Automation & Test in Europe (DATE)*, pp. 81–86, mar 2002.

[69] K. B. Kent, "Branch Sensitive Context Switching between Partitions in a Hardware/Software Co-Design of the Java Virtual Machine," in *2003 IEEE Pacific Rim Conference on Communications Computers and Signal Processing (PACRIM 2003) (Cat. No.03CH37490)*, vol. 2, pp. 642–645 vol.2, 2003.

[70] E. Gamma, R. Johnson, J. Vlissides, and R. Helm, "Design patterns: elements of reusable object-oriented software," 1995.

[71] Eric Auchard, Technology News, Reuters, "Rivals ARM and Intel make peace to secure Internet of Things." `https://www.spec.org/jvm98/`, Last accessed on 2019-06-19.

[72] Sean Michael Kerner, "Server Market Trends 2019," 2019. `https://www.serverwatch.com/server-reviews/server-market-trends-2019.html`, Last accessed on 2020-07-06.

[73] The NASM development team, "NASM - The Netwide Assembler." `https://www.nasm.us/`, Last Accessed on 2020-07-06.

[74] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The DaCapo Benchmarks: Java Benchmarking Development and Analysis," *SIGPLAN Not.*, vol. 41, pp. 169–190, Oct. 2006.

[75] The JUnit Team, "JUnit," 2020. `https://junit.org`,Last accessed on 2020-05-12.

[76] Intel, "Intel Atom® x5-Z8350 Processor." https://ark.intel.com/content/www/us/en/ark/products/93361/intel-atom-x5-z8350-processor-2m-cache-up-to-1-92-ghz.html, Last Access on 2020-04-25.

[77] Software in the Public Interest, "Debian, The Universal Operating System." https://www.debian.org/, Last Accessed on 2020-05-25.

[78] Python Software Foundation, "Statistics — Mathematical Statistics Functions," 2020. https://docs.python.org/3/library/statistics.html, Last accessed on 2020-06-10.

[79] E. Blem, J. Menon, and K. Sankaralingam, "A detailed analysis of contemporary ARM and x86 architectures," *UW-Madison Technical Report*, 2013.

# Appendix A

# Bytecode Implementation Status

The following listings show our implementation progress as of June 15, 2020. Note that bytecodes marked as Pending will be implemented in the future, while those marked as Unsupported will not be implemented.

## A.1 Constant Bytecodes

| Opcode | Mnemonic | Status |
|---|---|---|
| 00 (0x00) | nop | Pending |
| 01 (0x01) | aconst_null | Pending |
| 02 (0x02) | iconst_m1 | Complete |
| 03 (0x03) | iconst_0 | Complete |
| 04 (0x04) | iconst_1 | Complete |
| 05 (0x05) | iconst_2 | Complete |
| 06 (0x06) | iconst_3 | Complete |
| 07 (0x07) | iconst_4 | Complete |
| 08 (0x08) | iconst_5 | Complete |
| 09 (0x09) | lconst_0 | Complete |
| 10 (0x0a) | lconst_1 | Complete |
| 11 (0x0b) | fconst_0 | Complete |
| 12 (0x0c) | fconst_1 | Complete |
| 13 (0x0d) | fconst_2 | Complete |
| 14 (0x0e) | dconst_0 | Complete |
| 15 (0x0f) | dconst_1 | Complete |
| 16 (0x10) | bipush | Complete |

| | | |
|---|---|---|
| 17 (0x11) | sipush | Pending |
| 18 (0x12) | ldc | Pending |
| 19 (0x13) | ldc_w | Pending |
| 20 (0x14) | ldc2_w | Pending |

Table A.1: Implementation Status for Constant bytecodes.

## A.2 Load Bytecodes

| **Opcode** | **Mnemonic** | **Status** |
|---|---|---|
| 21 (0x15) | iload | Complete |
| 22 (0x16) | lload | Complete |
| 23 (0x17) | fload | Complete |
| 24 (0x18) | dload | Complete |
| 25 (0x19) | aload | Complete |
| 26 (0x1a) | iload_0 | Complete |
| 27 (0x1b) | iload_1 | Complete |
| 28 (0x1c) | iload_2 | Complete |
| 29 (0x1d) | iload_3 | Complete |
| 30 (0x1e) | lload_0 | Complete |
| 31 (0x1f) | lload_1 | Complete |
| 32 (0x20) | lload_2 | Complete |
| 33 (0x21) | lload_3 | Complete |
| 34 (0x22) | fload_0 | Complete |
| 35 (0x23) | fload_1 | Complete |
| 36 (0x24) | fload_2 | Complete |
| 37 (0x25) | fload_3 | Complete |
| 38 (0x26) | dload_0 | Complete |
| 39 (0x27) | dload_1 | Complete |
| 40 (0x28) | dload_2 | Complete |
| 41 (0x29) | dload_3 | Complete |
| 42 (0x2a) | aload_0 | Complete |
| 43 (0x2b) | aload_1 | Complete |
| 44 (0x2c) | aload_2 | Complete |
| 45 (0x2d) | aload_3 | Complete |
| 46 (0x2e) | iaload | Pending |
| 47 (0x2f) | laload | Pending |
| 48 (0x30) | faload | Pending |
| 49 (0x31) | daload | Pending |
| 50 (0x32) | aaload | Pending |
| 51 (0x33) | baload | Pending |
| 52 (0x34) | caload | Pending |

| | | |
|---|---|---|
| 53 (0x35) | saload | Pending |

Table A.2: Implementation Status for Load bytecodes.

## A.3 Store Bytecodes

| Opcode | Mnemonic | Status |
|---|---|---|
| 54 (0x36) | istore | Complete |
| 55 (0x37) | lstore | Complete |
| 56 (0x38) | fstore | Complete |
| 57 (0x39) | dstore | Complete |
| 58 (0x3a) | astore | Complete |
| 59 (0x3b) | istore_0 | Complete |
| 60 (0x3c) | istore_1 | Complete |
| 61 (0x3d) | istore_2 | Complete |
| 62 (0x3e) | istore_3 | Complete |
| 63 (0x3f) | lstore_0 | Complete |
| 64 (0x40) | lstore_1 | Complete |
| 65 (0x41) | lstore_2 | Complete |
| 66 (0x42) | lstore_3 | Complete |
| 67 (0x43) | fstore_0 | Complete |
| 68 (0x44) | fstore_1 | Complete |
| 69 (0x45) | fstore_2 | Complete |
| 70 (0x46) | fstore_3 | Complete |
| 71 (0x47) | dstore_0 | Complete |
| 72 (0x48) | dstore_1 | Complete |
| 73 (0x49) | dstore_2 | Complete |
| 74 (0x4a) | dstore_3 | Complete |
| 75 (0x4b) | astore_0 | Complete |
| 76 (0x4c) | astore_1 | Complete |
| 77 (0x4d) | astore_2 | Complete |
| 78 (0x4e) | astore_3 | Complete |
| 79 (0x4f) | iastore | Pending |
| 80 (0x50) | lastore | Pending |
| 81 (0x51) | fastore | Pending |
| 82 (0x52) | dastore | Pending |
| 83 (0x53) | aastore | Pending |
| 84 (0x54) | bastore | Pending |
| 85 (0x55) | castore | Pending |
| 86 (0x56) | sastore | Pending |

Table A.3: Implementation Status for Store bytecodes.

## A.4 Stack Bytecodes

| Opcode | Mnemonic | Status |
|---|---|---|
| 87 (0x57) | pop | Pending |
| 88 (0x58) | pop2 | Pending |
| 89 (0x59) | dup | Complete |
| 90 (0x5a) | dup_x1 | Complete |
| 91 (0x5b) | dup_x2 | Complete |
| 92 (0x5c) | dup2 | Complete |
| 93 (0x5d) | dup2_x1 | Complete |
| 94 (0x5e) | dup2_x2 | Complete |
| 95 (0x5f) | swap | Pending |

Table A.4: Implementation Status for Stack operation bytecodes.

## A.5 Math Bytecodes

| Opcode | Mnemonic | Status |
|---|---|---|
| 96 (0x60) | iadd | Complete |
| 97 (0x61) | ladd | Complete |
| 98 (0x62) | fadd | Pending |
| 99 (0x63) | dadd | Pending |
| 100 (0x64) | isub | Complete |
| 101 (0x65) | lsub | Complete |
| 102 (0x66) | fsub | Pending |
| 103 (0x67) | dsub | Pending |
| 104 (0x68) | imul | Complete |
| 105 (0x69) | lmul | Complete |
| 106 (0x6a) | fmul | Pending |
| 107 (0x6b) | dmul | Pending |
| 108 (0x6c) | idiv | Complete |
| 109 (0x6d) | ldiv | Complete |
| 110 (0x6e) | fdiv | Pending |
| 111 (0x6f) | ddiv | Pending |
| 112 (0x70) | irem | Pending |
| 113 (0x71) | lrem | Pending |
| 114 (0x72) | frem | Pending |
| 115 (0x73) | drem | Pending |
| 116 (0x74) | ineg | Complete |
| 117 (0x75) | lneg | Complete |
| 118 (0x76) | fneg | Pending |

| | | |
|---|---|---|
| 119 (0x77) | dneg | Pending |
| 120 (0x78) | ishl | Pending |
| 121 (0x79) | lshl | Pending |
| 122 (0x7a) | ishr | Pending |
| 123 (0x7b) | lshr | Pending |
| 124 (0x7c) | iushr | Pending |
| 125 (0x7d) | lushr | Pending |
| 126 (0x7e) | iand | Pending |
| 127 (0x7f) | land | Pending |
| 128 (0x80) | ior | Complete |
| 129 (0x81) | lor | Complete |
| 130 (0x82) | ixor | Complete |
| 131 (0x83) | lxor | Complete |
| 132 (0x84) | iinc | Complete |

Table A.5: Implementation Status for Math bytecodes.

## A.6 Type Conversion Bytecodes

| **Opcode** | **Mnemonic** | **Status** |
|---|---|---|
| 133 (0x85) | i2l | Pending |
| 134 (0x86) | i2f | Pending |
| 135 (0x87) | i2d | Pending |
| 136 (0x88) | l2i | Pending |
| 137 (0x89) | l2f | Pending |
| 138 (0x8a) | l2d | Pending |
| 139 (0x8b) | f2i | Pending |
| 140 (0x8c) | f2l | Pending |
| 141 (0x8d) | f2d | Pending |
| 142 (0x8e) | d2i | Pending |
| 143 (0x8f) | d2l | Pending |
| 144 (0x90) | d2f | Pending |
| 145 (0x91) | i2b | Pending |
| 146 (0x92) | i2c | Pending |
| 147 (0x93) | i2s | Pending |

Table A.6: Implementation Status for Type Conversion bytecodes.

## A.7 Comparison Bytecodes

| Opcode | Mnemonic | Status |
|---|---|---|
| 148 (0x94) | lcmp | Pending |
| 149 (0x95) | fcmpl | Pending |
| 150 (0x96) | fcmpg | Pending |
| 151 (0x97) | dcmpl | Pending |
| 152 (0x98) | dcmpg | Pending |
| 153 (0x99) | ifeq | Pending |
| 154 (0x9a) | ifne | Complete |
| 155 (0x9b) | iflt | Pending |
| 156 (0x9c) | ifge | Pending |
| 157 (0x9d) | ifgt | Pending |
| 158 (0x9e) | ifle | Pending |
| 159 (0x9f) | if_icmpeq | Pending |
| 160 (0xa0) | if_icmpne | Pending |
| 161 (0xa1) | if_icmplt | Pending |
| 162 (0xa2) | if_icmpge | Complete |
| 163 (0xa3) | if_icmpgt | Complete |
| 164 (0xa4) | if_icmple | Pending |
| 165 (0xa5) | if_acmpeq | Pending |
| 166 (0xa6) | if_acmpne | Pending |

Table A.7: Implementation Status for Comparison / Conditional bytecodes.

## A.8 Control Bytecodes

| Opcode | Mnemonic | Status |
|---|---|---|
| 167 (0xa7) | goto | Complete |
| 168 (0xa8) | jsr | Pending |
| 169 (0xa9) | ret | Pending |
| 170 (0xaa) | tableswitch | Pending |
| 171 (0xab) | lookupswitch | Pending |
| 172 (0xac) | ireturn | Complete |
| 173 (0xad) | lreturn | Complete |
| 174 (0xae) | freturn | Pending |
| 175 (0xaf) | dreturn | Pending |
| 176 (0xb0) | areturn | Pending |
| 177 (0xb1) | return | Complete |

Table A.8: Implementation Status for Control flow bytecodes.

## A.9 Reference Bytecodes

| Opcode | Mnemonic | Status |
|---|---|---|
| 178 (0xb2) | getstatic | Completed |
| 179 (0xb3) | putstatic | Completed |
| 180 (0xb4) | getfield SY | Pending |
| 181 (0xb5) | putfield SY | Pending |
| 182 (0xb6) | invokevirtual | Pending |
| 183 (0xb7) | invokespecial | Pending |
| 184 (0xb8) | invokestatic | Pending |
| 185 (0xb9) | invokeinterface | Pending |
| 186 (0xba) | invokedynamic | Pending |
| 187 (0xbb) | new | Pending |
| 188 (0xbc) | newarray | Pending |
| 189 (0xbd) | anewarray | Pending |
| 190 (0xbe) | arraylength | Pending |
| 191 (0xbf) | athrow | Pending |
| 192 (0xc0) | checkcast | Pending |
| 193 (0xc1) | instanceof | Pending |
| 194 (0xc2) | monitorenter | Pending |
| 195 (0xc3) | monitorexit | Pending |

Table A.9: Implementation Status for Reference bytecodes.

## A.10 Extended Bytecodes

| Opcode | Mnemonic | Status |
|---|---|---|
| 196 (0xc4) | wide | Pending |
| 197 (0xc5) | multianewarray | Pending |
| 198 (0xc6) | ifnull | Pending |
| 199 (0xc7) | ifnonnull | Pending |
| 200 (0xc8) | goto_w | Completed |
| 201 (0xc9) | jsr_w | Pending |

Table A.10: Implementation Status for Extended bytecodes.

## A.11 Reserved Bytecodes

| Opcode | Mnemonic | Status |
|---|---|---|
| 202 (0xca) | breakpoint | Pending |
| 254 (0xfe) | impdep1 | Unsupported |
| 255 (0xff) | impdep2 | Unsupported |

Table A.11: Implementation Status for Reserved bytecodes.

# Appendix B

# Unsupported Bytecode Frequency

The following listings show the bytecode frequency of the DaCapo Benchmark Suite version 9.12 [74] as of June 15, 2020. The invocation threshold, `mjitCount`, was set to 20.

## B.1 Avrora

| **Bytecode** | **Count** | **% of Unsupported** | **% of Total** |
|---|---|---|---|
| JBinvokevirtual | 2518 | 18.535 | 05.259 |
| JBputfield | 1647 | 12.124 | 03.440 |
| JBinvokespecial | 1530 | 11.262 | 03.196 |
| JBifeq | 773 | 05.690 | 01.615 |
| JBnewdup | 632 | 04.652 | 01.320 |
| JBldc | 575 | 04.233 | 01.201 |
| JBaconstnull | 366 | 02.694 | 00.764 |
| JBathrow | 351 | 02.584 | 00.733 |
| JBifnull | 338 | 02.488 | 00.706 |
| JBnop | 267 | 01.965 | 00.558 |
| **Total Bytecodes** | 47,877 | | |
| **Supported Bytecodes** | 34,292 | | |
| **Unsupported Bytecodes** | 13,585 | | |
| **Supported Methods** | 205 | | |
| **Unsupported Methods** | 1557 | | |

Table B.1: The 10 most used unsupported bytecodes for `avrora`.

## B.2 Eclipse

| Bytecode | Count | % of Unsupported | % of Total |
|---|---|---|---|
| JBinvokevirtual | 36,160 | 21.175 | 06.715 |
| JBputfield | 14,574 | 08.534 | 02.706 |
| JBinvokespecial | 11,699 | 06.851 | 02.172 |
| JBifeq | 10,062 | 05.892 | 01.868 |
| JBnop | 6150 | 03.601 | 01.142 |
| JBinvokeinterface | 6089 | 03.566 | 01.131 |
| JBinvokeinterface2 | 6089 | 03.566 | 01.131 |
| JBnewdup | 5747 | 03.365 | 01.067 |
| JBifnull | 4791 | 02.806 | 00.890 |
| JBaaload | 4456 | 02.609 | 00.827 |
| JBaconstnull | 4260 | 02.495 | 00.791 |
| JBcheckcast | 4043 | 02.368 | 00.751 |
| **Total Bytecodes** | 538,526 | | |
| **Supported Bytecodes** | 367,759 | | |
| **Unsupported Bytecodes** | 170,767 | | |
| **Supported Methods** | 1223 | | |
| **Unsupported Methods** | 10,289 | | |

Table B.2: The 10 most used unsupported bytecodes for `eclipse`.

## B.3 FOP

| Bytecode | Count | % of Unsupported | % of Total |
|---|---|---|---|
| JBinvokevirtual | 7598 | 22.674 | 07.565 |
| JBputfield | 3000 | 08.953 | 02.987 |
| JBinvokespecial | 2993 | 08.932 | 02.980 |
| JBldc | 1727 | 05.154 | 01.720 |
| JBnewdup | 1644 | 04.906 | 01.637 |
| JBifeq | 1631 | 04.867 | 01.624 |
| JBnop | 1075 | 03.208 | 01.070 |
| JBinvokeinterface | 1043 | 03.113 | 01.038 |
| JBinvokeinterface2 | 1043 | 03.113 | 01.038 |
| JBaconstnull | 866 | 02.584 | 00.862 |
| JBifnull | 833 | 02.486 | 00.829 |
| JBificmpne | 707 | 02.110 | 00.704 |
| **Total Bytecodes** | 100,435 | | |
| **Supported Bytecodes** | 66,925 | | |
| **Unsupported Bytecodes** | 33,510 | | |

| | |
|---|---|
| **Supported Methods** | 566 |
| **Unsupported Methods** | 2695 |

Table B.3: The 10 most used unsupported bytecodes for `fop`.

## B.4 H2

| Bytecode | Count | % of Unsupported | % of Total |
|---|---|---|---|
| JBinvokevirtual | 6590 | 21.592 | 06.967 |
| JBinvokespecial | 3324 | 10.891 | 03.514 |
| JBputfield | 1973 | 06.465 | 02.086 |
| JBifeq | 1695 | 05.554 | 01.792 |
| JBldc | 1659 | 05.436 | 01.754 |
| JBnewdup | 1162 | 03.807 | 01.228 |
| JBnop | 834 | 02.733 | 00.882 |
| JBinvokeinterface | 802 | 02.628 | 00.848 |
| JBinvokeinterface2 | 802 | 02.628 | 00.848 |
| JBathrow | 727 | 02.382 | 00.769 |
| JBifnull | 683 | 02.238 | 00.722 |
| JBaconstnull | 678 | 02.221 | 00.717 |
| **Total Bytecodes** | 94,587 | | |
| **Supported Bytecodes** | 64,067 | | |
| **Unsupported Bytecodes** | 30,520 | | |
| **Supported Methods** | 340 | | |
| **Unsupported Methods** | 2307 | | |

Table B.4: The 10 most frequently used unsupported bytecodes for `h2`.

## B.5 Jython

| Bytecode | Count | % of Unsupported | % of Total |
|---|---|---|---|
| JBinvokevirtual | 29,480 | 31.736 | 11.672 |
| JBaconstnull | 7009 | 07.545 | 02.775 |
| JBsipush | 5116 | 05.508 | 02.026 |
| JBinvokespecial | 5027 | 05.412 | 01.990 |
| JBputfield | 4236 | 04.560 | 01.677 |
| JBnop | 4030 | 04.338 | 01.596 |
| JBinvokeinterface | 3330 | 03.585 | 01.318 |
| JBinvokeinterface2 | 3330 | 03.585 | 01.318 |

| | | | |
|---|---|---|---|
| JBldc | 3307 | 03.560 | 01.309 |
| JBpop | 2833 | 03.050 | 01.122 |
| JBifeq | 2803 | 03.018 | 01.110 |
| JBnewdup | 2572 | 02.769 | 01.018 |
| **Total Bytecodes** | 252,573 | | |
| **Supported Bytecodes** | 159,682 | | |
| **Unsupported Bytecodes** | 92,891 | | |
| **Supported Methods** | 539 | | |
| **Unsupported Methods** | 4042 | | |

Table B.5: The 10 most used unsupported bytecodes for `jython`.

## B.6 Luindex

| **Bytecode** | **Count** | **% of Unsupported** | **% of Total** |
|---|---|---|---|
| JBinvokevirtual | 3226 | 18.962 | 05.692 |
| JBinvokespecial | 1678 | 09.863 | 02.961 |
| JBputfield | 1510 | 08.876 | 02.664 |
| JBnewdup | 873 | 05.131 | 01.540 |
| JBifeq | 791 | 04.649 | 01.396 |
| JBldc | 758 | 04.455 | 01.337 |
| JBathrow | 612 | 03.597 | 01.080 |
| JBaconstnull | 414 | 02.433 | 00.730 |
| JBifnull | 413 | 02.428 | 00.729 |
| JBificmpne | 348 | 02.045 | 00.614 |
| JBifnonnull | 329 | 01.934 | 00.580 |
| JBaaload | 320 | 01.881 | 00.565 |
| **Total Bytecodes** | 56,676 | | |
| **Supported Bytecodes** | 39,663 | | |
| **Unsupported Bytecodes** | 17,013 | | |
| **Supported Methods** | 184 | | |
| **Unsupported Methods** | 1414 | | |

Table B.6: The 10 most used unsupported bytecodes for `luindex`.

## B.7 Lusearch

| Bytecode | Count | % of Unsupported | % of Total |
|---|---|---|---|
| JBinvokevirtual | 3270 | 19.247 | 06.020 |
| JBinvokespecial | 2033 | 11.966 | 03.742 |
| JBputfield | 1675 | 09.859 | 03.083 |
| JBnewdup | 913 | 05.374 | 01.681 |
| JBldc | 801 | 04.715 | 01.475 |
| JBifeq | 746 | 04.391 | 01.373 |
| JBathrow | 540 | 03.178 | 00.994 |
| JBaconstnull | 474 | 02.790 | 00.873 |
| JBifnull | 416 | 02.448 | 00.766 |
| JBificmpne | 390 | 02.295 | 00.718 |
| JBifnonnull | 306 | 01.801 | 00.563 |
| JBreturnZ | 284 | 01.672 | 00.523 |
| **Total Bytecodes** | 54,323 | | |
| **Supported Bytecodes** | 37,333 | | |
| **Unsupported Bytecodes** | 16,990 | | |
| **Supported Methods** | 193 | | |
| **Unsupported Methods** | 1477 | | |

Table B.7: The 10 most used unsupported bytecodes for `lusearch`.

## B.8 PMD

| Bytecode | Count | % of Unsupported | % of Total |
|---|---|---|---|
| JBinvokevirtual | 5640 | 16.255 | 05.457 |
| JBinvokespecial | 4671 | 13.462 | 04.519 |
| JBputfield | 2619 | 07.548 | 02.534 |
| JBifeq | 2515 | 07.248 | 02.433 |
| JBreturnZ | 1722 | 04.963 | 01.666 |
| JBnewdup | 1300 | 03.747 | 01.258 |
| JBnop | 1262 | 03.637 | 01.221 |
| JBinvokeinterface | 1225 | 03.530 | 01.185 |
| JBinvokeinterface2 | 1225 | 03.530 | 01.185 |
| JBldc | 1222 | 03.522 | 01.182 |
| JBathrow | 863 | 02.487 | 00.835 |
| JBsipush | 851 | 02.453 | 00.823 |
| **Total Bytecodes** | 103,354 | | |
| **Supported Bytecodes** | 68,656 | | |
| **Unsupported Bytecodes** | 34,698 | | |

| | |
|---|---|
| **Supported Methods** | 280 |
| **Unsupported Methods** | 2859 |

Table B.8: The 10 most used unsupported bytecodes for `pmd`.

## B.9 Sunflow

| Bytecode | Count | % of Unsupported | % of Total |
|---|---|---|---|
| JBinvokevirtual | 2955 | 15.942 | 04.713 |
| JBinvokespecial | 1606 | 08.664 | 02.562 |
| JBputfield | 1460 | 07.877 | 02.329 |
| JBnewdup | 807 | 04.354 | 01.287 |
| JBldc | 778 | 04.197 | 01.241 |
| JBifeq | 711 | 03.836 | 01.134 |
| JBfmul | 581 | 03.134 | 00.927 |
| JBathrow | 450 | 02.428 | 00.718 |
| JBaconstnull | 431 | 02.325 | 00.687 |
| JBreturnZ | 377 | 02.034 | 00.601 |
| JBifnull | 359 | 01.937 | 00.573 |
| JBificmpne | 324 | 01.748 | 00.517 |
| **Total Bytecodes** | 62,697 | | |
| **Supported Bytecodes** | 44,161 | | |
| **Unsupported Bytecodes** | 18,536 | | |
| **Supported Methods** | 263 | | |
| **Unsupported Methods** | 1482 | | |

Table B.9: The 10 most used unsupported bytecodes for `sunflow`.

## B.10 Xalan

| Bytecode | Count | % of Unsupported | % of Total |
|---|---|---|---|
| JBinvokevirtual | 5933 | 23.483 | 07.312 |
| JBinvokespecial | 2347 | 09.290 | 02.892 |
| JBputfield | 2222 | 08.795 | 02.738 |
| JBldc | 1368 | 05.415 | 01.686 |
| JBifeq | 1344 | 05.320 | 01.656 |
| JBaconstnull | 1102 | 04.362 | 01.358 |
| JBnewdup | 1071 | 04.239 | 01.320 |
| JBificmpne | 626 | 02.478 | 00.771 |

| | | | |
|---|---|---|---|
| JBathrow | 596 | 02.359 | 00.735 |
| JBifnull | 581 | 02.300 | 00.716 |
| JBnop | 504 | 01.995 | 00.621 |
| JBreturnZ | 475 | 01.880 | 00.585 |
| **Total Bytecodes** | 81,143 | | |
| **Supported Bytecodes** | 55,878 | | |
| **Unsupported Bytecodes** | 25,265 | | |
| **Supported Methods** | 304 | | |
| **Unsupported Methods** | 2192 | | |

Table B.10: The 10 most used unsupported bytecodes for `xalan`.

# Vita

Candidate's full name:
Eric Douglas Coffin

University attended (with dates and degrees obtained):

- University of New Brunswick, BCS, 2003
- University of New Brunswick, MCS, 2020

Publications:

- Papers:
  - 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 the 29th Annual International Conference on Computer Science and Software Engineering (CASCON '19). IBM Corp., USA, 293–298.
- Refereed Posters:
  - 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. 29th Annual International Conference on Computer Science and Software Engineering (CASCON '19), Markham, Canada, Nov 4-6, 2019.

Conference Presentations:

- 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. 29th Annual International Conference on Computer Science and Software Engineering (CASCON '19), Markham, Canada, Nov 4-6, 2019.