Just-In-Time Compiler Performance Evaluation on the AArch64 Platform

by

Aaron G. Graham

Bachelor of Computer Science,
(Co-Operative Education Program),
University of New Brunswick, 2013

A thesis Submitted in Partial Fulfilment of
the Requirements for the Degree of

Master of Computer Science

In the Graduate Academic Unit of Computer Science

Supervisor: Kenneth B. Kent, Ph.D., Computer Science

Examining Board: Gerhard W. Dueck, Ph.D., Computer Science, Chair
Suprio Ray, Ph.D., Computer Science
Julian Cardenas-Barrera, Ph.D.,
Electrical and Computer Engineering

This thesis is accepted by the
Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

September, 2021

© Aaron G. Graham, 2021

Abstract

The embedded computing market, which includes Internet-of-Things (IoT) and mobile computing devices, is a non-traditional computing market where computation resources—including CPU, memory, power—are more limited. Due to these limitations, software is required to be more compact and efficient. Providing a managed runtime, such as Eclipse OpenJ9 built on top of Eclipse OMR, in this climate differs from a cloud/desktop-based environment. This thesis focuses on porting the OpenJ9+OMR technology, which has a heritage of running in resource constrained systems, to a new environment while continuing to provide a generic run-time environment. The low-power AArch64 (ARMv8-A) platform—compatible with commonly used electronic devices—is becoming the answer for resource constrained environments of embedded systems. This thesis explores the performance of the Just-In-Time (JIT) compiler in OpenJ9, a Java®[1] Virtual Machine (JVM), on an Instruction Set Architecture (ISA) appropriate for IoT and embedded devices. More specifically, we evaluate and validate the AArch64 implementation of OpenJ9's JIT against more mature architectures currently available. The evaluation reveals performance discrepancies and necessary improvements, beyond those that are already known, by comparing the AArch64 implementation to another ISA, x86-64. Our work is an effort to template new architectural support and allow others to follow our model. We provide a baseline for future research on OpenJ9, OMR and the JIT on the AArch64 platform and outline some improvements as future work.

[1] Java® is a registered trademark of Oracle and/or its affiliates.

Dedication

To my loving wife Crystal. Without your support this work and this thesis document would not have been possible. Thank you so very much, more than words can express!

"I sought perfection and found so much more!"—Paraphrased from Orson Scott Card, Ender's Game "Speaker" Series, Book 2 [1]

Acknowledgements

I would like to thank Jean-Philippe Legault for the continuous guidance and the help through-out the project. I would also like to thank Julie Brown as she served as an invaluable sounding-board. I'd like to acknowledge DeVerne Jones for the continuing support he provides that enables my research efforts. Next, I'd like to thank my Technical Project Manger at CAS-Atlantic Stephen MacKay for his very keen grammatical eye and always helpful research suggestions. From the Faculty of Computer Science, I'd like to thank my supervisor Dr. Kenneth B. Kent for his valuable advice on research direction. All of the above helped keep me sane through this thesis process and the COVID-19 pandemic! To the AArch64 team at IBM, Daryl Maier, Kazuhiro Konno and James Kingdon, many thanks for their initial and ongoing technical advice and guidance along the way. As alluded to before, this research was conducted within the Centre for Advanced Studies—Atlantic, Faculty of Computer Science, University of New Brunswick. The author is grateful for all the colleagues of CAS–Atlantic in supporting this research. Furthermore, the author would like to acknowledge the funding support provided by the Atlantic Canada Opportunities Agency (ACOA) through the Atlantic Innovation Fund (AIF) program. Finally, but not least, the author would also like to thank the New Brunswick Innovation Foundation for contributing to this research.

Table of Contents

Abstract ii

Dedication iii

Acknowledgments iv

Table of Contents v

List of Tables viii

List of Figures ix

Listings xi

List of Acronyms xii

1 Introduction 1

1.1 Contributions 3

2 Background 6

2.1 Virtual Machines (VMs) 6

2.1.1 Interpreters 7

2.1.2 The *Java* Virtual Machine (JVM) 8

2.2 Eclipse OMR and Eclipse OpenJ9 9

2.2.1 Eclipse OMR 9

2.2.2 The Eclipse OpenJ9 JVM 10
2.2.3 The IBM J9 JVM 10
2.3 Optimization 11
2.3.1 VM Just-In-Time (JIT) Compiler 11
2.3.2 The Eclipse OpenJ9 JVM Testarossa JIT (TRJIT) 12
2.4 Evaluation 15
2.4.1 Benchmarking 16
2.4.1.1 DaCapo Benchmarks 16
2.4.1.2 SciMark Benchmarks 16
2.4.1.3 SPECjvm®2008 Benchmarks 17
2.4.1.4 Renaissance Suite 17
2.4.2 Perf 18

3 Design of the AArch64 JVM and TRJIT 19

3.1 From Port to Port 19
3.2 Eclipse OMR TRJIT Design 20
3.2.1 OMR Intermediate Language (IL) 22
3.2.2 OMR Tree Evaluators 23
3.2.2.1 OpenJ9 Overridden Tree Evaluators 25
3.2.3 OMR Tril Tests 25
3.2.4 Instruction OpCode Mnemonics 26
3.2.5 Binary Encoding 27
3.2.6 Infinite Virtual Register Set and Register Assignment 28

4 Evaluation of the AArch64 JVM and TRJIT 30

4.1 Experimental Setup 30
4.1.1 The Devices 32
4.1.2 The Benchmarking and Testing Framework 33

4.2 Experimental Evaluation 34
4.2.1 Results 35
4.2.2 Bytecode Granularity Benchmarking 42
4.3 Templating New Architectural Support 46

5 Future Work **52**

6 Conclusion **54**

Bibliography **57**

Appendix A **74**

Appendix B **81**

Vita

List of Tables

4.1 Beelink BT3 Pro II x86-64 Mini PC specifications. 31

4.2 Pine64 Rock64 A53-based AArch64 embedded device specifications. . 31

4.3 Khadas VIM3 big.LITTLE A73/A53-based AArch64 embedded device specifications. 31

4.4 Raspberry Pi 4 B A72-based AArch64 embedded device specifications. 32

6.1 Branch bytecode test execution time in milliseconds for 1,000,000 iterations. 81

6.2 Conversion Branch bytecode test execution time in milliseconds for 1,000,000 iterations. 82

6.3 Double Branch bytecode test execution time in milliseconds for 1,000,000 iterations. 83

6.4 Float Branch bytecode test execution time in milliseconds for 1,000,000 iterations. 83

6.5 Int Branch bytecode test execution time in milliseconds for 1,000,000 iterations. 84

6.6 Invoke Branch bytecode test execution time in milliseconds for 1,000,000 iterations. 85

6.7 Long Branch bytecode test execution time in milliseconds for 1,000,000 iterations. 86

6.8 Stack bytecode test execution time in milliseconds for 1,000,000 iterations. 87

List of Figures

1.1 *Java* source files (`.java`) transpiled to Java *bytecode* files (`.class`) via the *javac* executable. 2

1.2 *Java* class files executed by a JVM. 3

2.1 A JVM saves the JIT-ed code to the Shared Class Cache (SCC) for later Ahead-of-Time (AoT) loading. 8

2.2 Ahead-of-Time (AoT) load of JIT-ed code, originating in a previous JVM execution, from the SCC into a subsequent JVM run. 9

2.3 The JVM's tight execution loop for interpretation and execution of JIT-compiled code. 12

2.4 Tiered Compilation. 13

2.5 The JVM's recompilation loop for interpretation and execution of JIT-compiled code. 14

3.1 Overview of TRJIT Compiler Phases. 21

3.2 Graphical representation of the IL tree for the arithmetic expression: $a = \frac{a-b}{c}$. 23

4.1 DaCapo elapsed time relative to x86-64 device comparison for OpenJ9 0.23 (Log10, negative is better). 36

4.2 SPECjvm®2008 elapsed time relative to x86-64 device comparison for OpenJ9 0.23 (Log10, negative is better). 37

4.3 SPECjvm®2008 startup elapsed time relative to x86-64 device comparison for OpenJ9 0.23 (Log10, negative is better). 38
4.4 SciMark elapsed time relative to x86-64 device comparison for OpenJ9 0.23 (Log10, negative is better). 39
4.5 Renaissance Suite elapsed time relative to x86-64 device comparison for OpenJ9 0.23 (Log10, negative is better). 40
4.6 Bytecode Branch Tests (1,000,000 iterations). 43
4.7 Bytecode Conversion Tests (1,000,000 iterations). 44
4.8 Bytecode Double Tests (1,000,000 iterations). 45
4.9 Bytecode Float Tests (1,000,000 iterations). 46
4.10 Bytecode Int Tests (1,000,000 iterations). 47
4.11 Bytecode Invoke Tests (1,000,000 iterations). 48
4.12 Bytecode Long Tests (1,000,000 iterations). 49
4.13 Bytecode Stack Tests (1,000,000 iterations). 50

Listings

3.1 Virtual instruction representation of arithmetic expression: $a = \frac{a-b}{c}$. . 22

3.2 ARMv8-A instruction mnemonics for 32-bit arithmetic add and 64-bit arithmetic add operations. 27

3.3 OMR instruction mnemonics for 32-bit arithmetic add and 64-bit arithmetic add operations. 27

3.4 ARMV8-A binary encoding in OMR for 32-bit arithmetic add and 64-bit arithmetic add operations. 28

4.1 Java command-line options used during OpenJ9 runs. 30

List of Acronyms

AArch64	ARMv8-A / ARM 64-Bit (ARM64)
ISA	Instruction Set Architecture
HLL	High-Level Language
VM	Virtual Machine
JVM	Java Virtual Machine
JCL	Java Class Library
JIT	Just-In-Time
JIT Compiler	Just-In-Time Compiler
TRJIT	Testarossa JIT
API	Application Programming Interface
ABI	Application Binary Interface
IL	Intermediate Language
JDK	Java Development Kit
SCC	Shared Class Cache
OS	Operating System
KVM	Kernel-based Virtual Machine
Codegen	Code Generator
GPR	General-Purpose Register
FPR	Floating-Point Register
VPR	Vector Register

Chapter 1

Introduction

The AArch64, or ARMv8-A (ARM 64-Bit), Instruction Set Architecture (ISA) is increasingly a more common platform in the technology ecosystem; this is especially true for the mobile, embedded and IoT computing spaces [2, 3]. AArch64 is a RISC-based (Reduced Instruction Set Computer) ISA [4] in contrast to x86-64, a CISC-based (Complex Instruction Set Computer) ISA. A RISC ISA provides ease of use from a reduced, and a more general instruction-set, requiring fewer CPU cycles to execute. A CISC ISA, on the other hand, provides more specialized and complex instructions, which may take more CPU clock cycles to execute.

The AArch64 architecture is designed with integration into low-power System on Chips (SoCs) in mind [4]. That being said, because of the low-powered design per-SoC, AArch64 also lends itself to being packed into larger packages with many cores for use in the high-performance server space. This multifaceted nature makes the AArch64 an interesting architecture for research investigation.

Java is a high-level language (HLL) that enables programmers to write platform, architecture and hardware agnostic code; "Write once, run anywhere" [5, 6]. The language semantics, usage and the *Java* Class Library (JCL) overall layout is defined in the *Java* Language Specification [6]. *Java*'s popularity has remained strong over

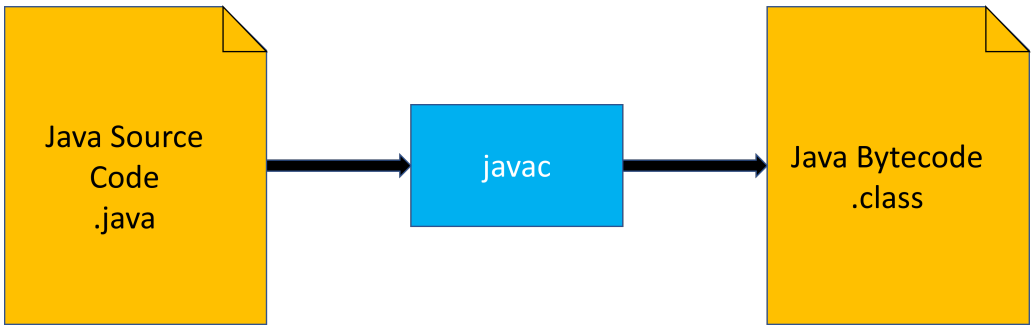

Figure 1.1: *Java* source files (`.java`) transpiled to Java *bytecode* files (`.class`) via the *javac* executable.

the years; in 2019, Stack Overflow listed *Java* as the 5th most used language [7]. This popularity makes *Java* an interesting language, with many large-scale enterprise applications, to target research towards [8]. The *Java* language framework, using the *javac* program, compiles the source into an intermediate bytecode format; see Figure 1.1.

The Java Virtual Machine (JVM) is the application that executes the end-user *Java* programs on the underlying platform, architecture and hardware. A platform-specific JVM executes *Java* bytecode on its native architecture. JVMs are implemented against the JVM Specification [9–11]. There are many implementations of the JVM (J9, OpenJ9, GraalVM, Hotspot, etc.) [12–17], but we focus on the Eclipse OpenJ9 JVM. *Java* source files are compiled into *Java* bytecode, which then can be interpreted or just-in-time (JIT) compiled by the JVM.

Interpreting the individual bytecode in a simple fetch, decode, emulate pattern, allows the JVM to have a very low overhead at the cost of performance [18]. Just-in-time compilation is a form of compilation at runtime that can achieve higher levels of optimization than interpretation, albeit with more runtime overhead. JIT compilation is discussed in Subsection 2.3.1.

The remainder of the thesis is organized as follows: background and related work are in Chapter 2; design and implementation of the OpenJ9 and OMR JIT on AArch64 are presented in Chapter 3; the evaluation and results are discussed in in Chapter 4;

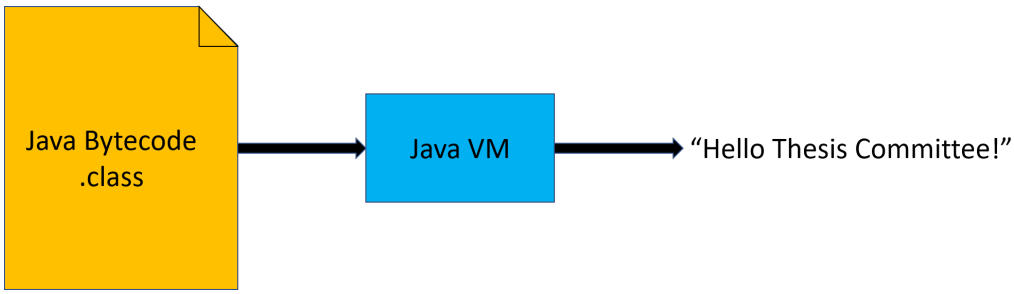

Figure 1.2: *Java* class files executed by a JVM.

with and future work in Chapter 5 and conclusions in Chapter 6.

1.1 Contributions

In this thesis, we focus on bringing Eclipse OpenJ9 and Eclipse OMR's Just-in-Time (JIT) compiler to the AArch64 platform. More specifically, we bring AArch64-specific design and implementation for the following to OpenJ9 and OMR:

- Build (Make and CMake) infrastructure [A1–A3].
- AArch64 JVM platform support (Port Library) [A4].
- Code generation:
  - Binary encoding [A5–A7].
  - Opcode mnemonics [A5, A7–A9].
  - AArch64 GCMaps [A10].
  - AArch64 Processor Information [A11].
  - Tree Evaluator implementations [A12–A40].
  - Testarossa JIT (TRJIT) Code generator helper functions [A41, A42].
  - Tril Tests [A43–A47].
  - Trampoline support for AArch64 [A48, A49].

  - Store/load/full/allocationFence for AArch64 [A50].
  - Unsafe_compareAndSwapInt_jlObjectJII_Z [A51].

- Expansion (Renaissance Suite) and refinement of the benchmarking framework (Subsection 4.1.2) [19–21].

- Evaluation and validation of the AArch64 implementation of the JIT against a more mature architecture (Subsection 4.2.1).
  - The evaluation reveals performance discrepancies and optimization opportunities by comparing the AArch64 implementation to the x86-64 implementation.

- We designed and implemented the AArch64 JIT from scratch using the other platform's compilers as templates.
  - This required us to make many decisions on how to architect and implement the AArch64 solution.
  - These decisions are explored in Section 3.1.

- Our work is a template for new architectural support and allows others to follow our model for new architectures.
  - This is explored in Section 4.3.

- We provide a baseline for future research on OMR and OpenJ9 on the AArch64 platform.

In terms of novelty of this research, we bring AArch64 support to the Eclipse OpenJ9 JVM, Eclipse OMR and the TRJIT (Testarossa JIT). This allows for an enterprise-grade AArch64 JVM for ARMv8-A devices, desktops and servers. In the server space, an example of this platform would be the Cavium-based ThunderX family of

servers [22]. In the device space, some examples of this platform would be the the Pine64 Rock64, the Khadas VIM3 and the Raspberry Pi 4 B (more on these devices in Subsection 4.1.1). In the desktop space, some examples of this platform would be the the Apple Silicon M1 chip in the MacBook Pro laptop and Mac Mini desktop [23]. Not only do we bring this support to the OpenJ9 JVM and TRJIT, we provide a baseline set of results for this implementation to compare back to and we identify bottlenecks for future improvement. We also provide a template for implementing architectural support on new and different platforms.

Chapter 2

Background

This chapter discusses a number of concepts necessary to provide a background for the later chapters. These concepts include: virtual machines (VMs) (Section 2.1), Eclipse OMR and Eclipse OpenJ9 (Section 2.2), VM optimization (Section 2.3) and JVM evaluation (Section 2.4) in terms of standard benchmarks and Linux performance tools (Subsection 2.4.2).

2.1 Virtual Machines (VMs)

In this section we discuss VMs (Section 2.1), Interpreters (Subsection 2.1.1) and the Java Virtual Machine (JVM) (Subsection 2.1.2). Over the last 30-plus years language Virtual Machines (VMs), also known as process virtual machines or managed runtimes, have become a very popular mechanism for executing user applications; in contrast to static compilation (compiled before execution e.g., C, C++, etc.) [18]. Language VMs should not be conflated with hypervisor virtual machines—either level-1 bare-metal hypervisors (e.g., VMware ESXi, XEN or Proxmox) or level-2 hosted on an operating system (OS) hypervisors (e.g., Hyper-V, VirtualBox or Kernel-based Virtual Machine—KVM)—also known as system virtual machines, in this thesis [18, 24–34].

While many of the mechanisms that enable both types of VMs to function are similar or the same, for our purposes we focus on high-level language (HLL) Virtual Machines (VMs) [18]. An HLL VM is designed to execute a platform-agnostic application inside an operating system on platform-specific hardware [18]. The end goal of using a VM is to have the generic application operate within that system the same as if it were natively compiled for that platform [18].

There are two main implementations of VMs: stack-based or register-based [18, 35, 36]. Stack-based VMs push and pop all operands and return values on the stack during runtime [18, 35, 36]. Stack-based implementations may push and pop the same operands multiple times to successfully fulfill the user's application [35, 36]. Register-based implementations, on the other hand, allocate all values into corresponding registers in the CPU; the VM operates on the registers' values. Stack-based VMs are easier to implement but, due to the issue with multiple pushes and pops on the same operands, have increased overhead. Register-based VMs, while much harder to implement, can take advantage of memory and CPU usage optimizations.

2.1.1 Interpreters

The main mechanism for many HLL VMs is the interpretation of source instructions, or virtual instructions, to provide the first line of support for executing an end-user's application [18, 37]. VM interpreters provide reliable execution of program code. They interpret in a standard pattern of *fetch, decode and emulate* the execution of the individual virtual instructions [18]. VM interpreters have very low overhead, in terms of startup, compared to other methods of execution. However, interpreted code is executed slower than native hardware instructions.

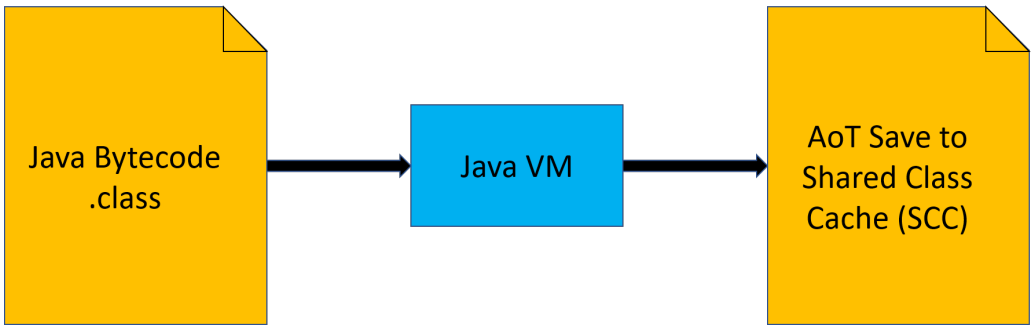

Figure 2.1: A JVM saves the JIT-ed code to the Shared Class Cache (SCC) for later Ahead-of-Time (AoT) loading.

2.1.2 The *Java* Virtual Machine (JVM)

The JVM is the application that interprets, or runs, the agnostic end-user *Java* programs on the underlying platform, architecture and hardware. A platform-specific JVM executes *Java* bytecode on its native architecture. JVMs are implemented against the JVM Specification [9–11]. There are many implementations of the JVM (J9, OpenJ9, GraalVM, Hotspot, etc.) [12–17], but we focus on one implementation, Eclipse OpenJ9, as discussed in Subsection 2.2.2. As mentioned in Chapter 1, we see in Figure 1.2 *Java* source files compiled into *Java* bytecode, which then can be interpreted or just-in-time (JIT) compiled by the JVM. Just-in-time compilation is a form of compilation at runtime that can achieve higher levels of optimization than interpretation, albeit with an increased one-time runtime overhead. This additional overhead is required to perform the optimization itself, however, this one-time cost is considered minor and it is amortized over the entire runtime of the user application in the JVM. JIT compilation is touched on more in Subsection 2.3.1.

We see in Figure 2.1 some JVMs take the just-in-time compiled (JIT-ed/JIT compiled) code from an application's run and save it to a data store, sometimes called a Shared Class Cache (SCC), for loading into a subsequent JVM run. This subsequent load can be seen in Figure 2.2. This later loading of previously JIT-ed code is called Ahead-of-Time (AoT) compilation.

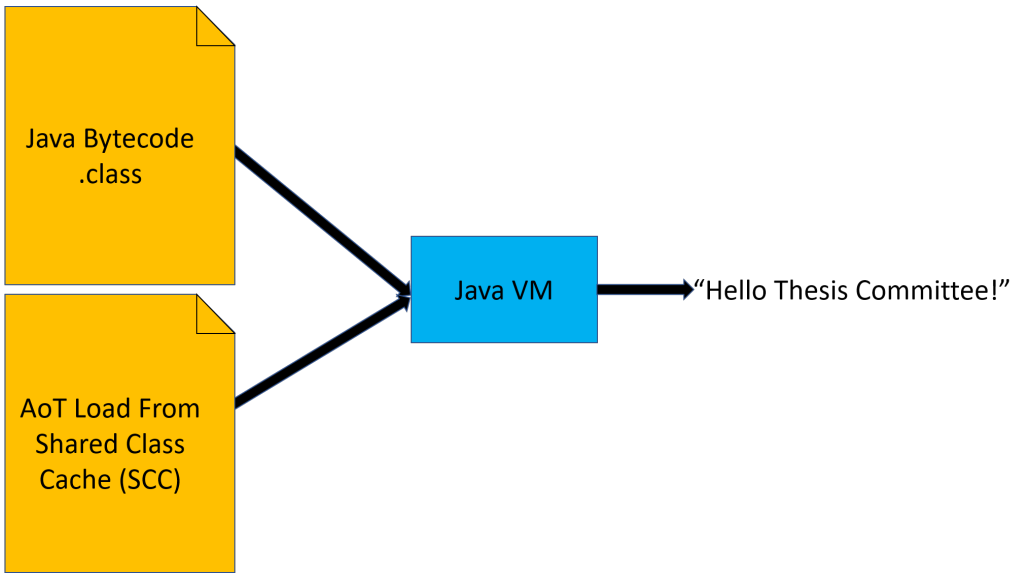

Figure 2.2: Ahead-of-Time (AoT) load of JIT-ed code, originating in a previous JVM execution, from the SCC into a subsequent JVM run.

2.2 Eclipse OMR and Eclipse OpenJ9

In this section, we discuss Eclipse OMR (Subsection 2.2.1), the Eclipse OpenJ9 JVM (Subsection 2.2.2) and the IBM J9 JVM (Subsection 2.2.3).

2.2.1 Eclipse OMR

Eclipse OMR is an open-source toolkit of language-agnostic, C and C++, components that can be used to build robust language runtimes to support many different hardware and operating system platforms [38–45]. These components include a garbage collector, a just-in-time compiler, the JitBuilder API, a platform support library, a threading library and a signal handling library. Of note here is JitBuilder, a simplified way to incorporate OMR's TRJIT technology into a front-end language. However, OpenJ9 discussed in Subsection 2.2.2, hooks directly into TRJIT in OMR, bypassing the JitBuilder interface entirely. Eclipse OMR originated as the language-agnostic VM core of the IBM *closed-source* J9 JVM. Due to the many enterprise-grade optimizations made to Eclipse OMR and its predecessors, it makes an interesting

platform for the exploration of new architectures. In this case, basic support for AArch64, primarily the framework necessary for the interpreter to work, exists in OMR. Adding a JIT compiler is investigated and implemented as a part of this research work.

2.2.2 The Eclipse OpenJ9 JVM

Eclipse OpenJ9 is a *Java* Virtual Machine for OpenJDK that consumes the Eclipse OMR toolkit and libraries for its language-agnostic components [12, 13, 46]. Eclipse OpenJ9, built on top of Eclipse OMR, already has AArch64 support for running end-user *Java* applications via interpretation. As of the time of writing, early releases of an AArch64 OpenJ9 OpenJDK11 are hosted by AdoptOpenJDK; going forward, IBM will host the binaries themselves as the "IBM Semeru Runtimes" [47, 48]. These releases contain previous work and ongoing work on the AArch64 JVM and JIT [49, 50]. Eclipse OpenJ9 and Eclipse OMR are consumed as a part of IBM's enterprise J9 JVM, discussed in Subsection 2.2.3. Many components for the AArch64 JIT reside in Eclipse OMR, but several components live in Eclipse OpenJ9 as well. So given we cannot do a practical test of the AArch64 JIT with just OMR alone and that there are some interesting *Java*-specific JIT components in OpenJ9, we look at both Eclipse OpenJ9 and Eclipse OMR together for this work.

2.2.3 The IBM J9 JVM

The J9 JVM is IBM's enterprise-grade implementation of the *Java* Language and Virtual Machine Specifications [6, 10, 11, 51]. J9 contains decades of IBM developer-years of optimized implementation [52]. The current blueprint of J9 consumes Eclipse OMR and Eclipse OpenJ9 with some extra IBM "secret sauce" (e.g., special cryptographic libraries, etc.) on top; this is what separates IBM J9 from Eclipse OpenJ9. This consumption means that all retained, GitHub contributed, development

efforts and optimizations eventually make it into the IBM's enterprise J9 JVM.

2.3 Optimization

In this section, we will discuss optimizations from VM JIT compilation (Subsection 2.3.1) and specifically the Eclipse OpenJ9 JVM Testarossa JIT compiler (Subsection 2.3.2).

2.3.1 VM Just-In-Time (JIT) Compiler

Just-in-time compilation, or dynamic compilation, is a mechanism that VMs can use to improve performance [14–16, 18]. The VM uses profiling data, usually run-time profiling data, to determine the temperature of code blocks. A temperature gradient, or *hotness*, is used to describe the frequency at which a function is used: cold, warm, hot, very-hot or scorching; where a hotter function or method is where more time is being spent (i.e., code blocks infrequently executed are called *cold*, code executed occasionally is referred to as *warm* and blocks frequently executed are considered *hot*, etc.). Profiling provides the information required to decide which discrete blocks of code are considered ideal for optimizing [53]. Once this *hotness* threshold is reached, JITs compile the equivalents to the source instructions on the native platform; this natively compiled code is also known as generated code. Once generated, the VM then executes the native code instead of interpreting the bytecode on subsequent executions. The VM only falls back to the much slower interpreted version if JIT-generated code is not available in the code cache [37]. JITs enable VMs to attain increased performance over the long term of that VM's execution. Because the JIT takes CPU execution time away from the end-user's program, JITs increase startup time overhead [54].

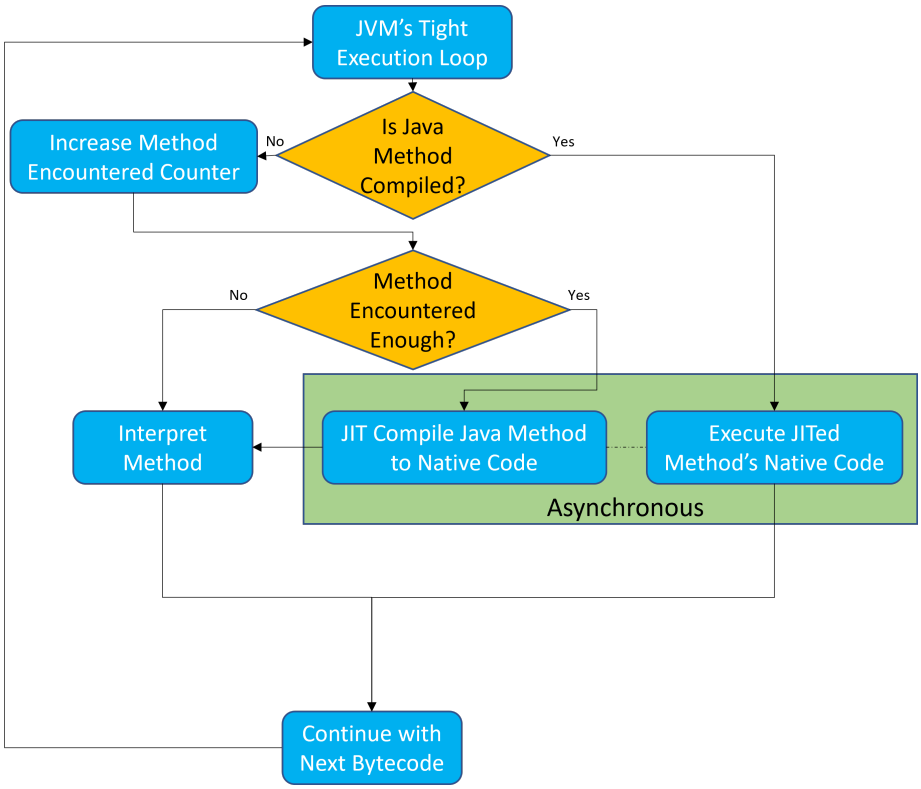

Figure 2.3: The JVM's tight execution loop for interpretation and execution of JIT-compiled code.

2.3.2 The Eclipse OpenJ9 JVM Testarossa JIT (TRJIT)

The Testarossa JIT, or TRJIT, is the multi-pass optimizer and JIT compiler component of Eclipse OMR, and by extension, Eclipse OpenJ9 [53, 55]. TRJIT is the primary component that we focus upon in this research. TRJIT has been heavily optimized to generate extremely performant native code and also offers tiered levels of compilation for the purposes of optimization. Generated code, if sufficiently *hot*, can be re-compiled at a higher level of optimization, albeit with added CPU and memory overhead on each re-compilation. Because of this, TRJIT can sometimes be a heavy-weight option, with a significant CPU overhead and consuming much memory. This thesis investigates and implements TRJIT on the AArch64 platform and focuses on possible TRJIT optimizations for resource-constrained or embedded environments.

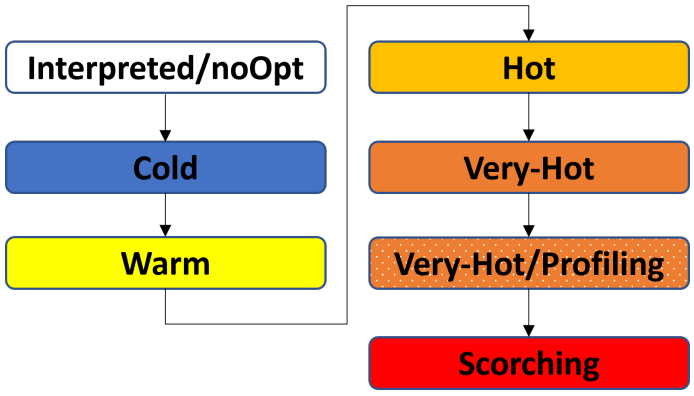

Figure 2.4: Tiered Compilation.

In Figure 2.3, we see the JVM's tight execution loop, or bytecode loop, for interpretation and JIT compilation. The bytecode loop is a continuance engine; it processes one bytecode from the incoming stream after the next until the user's program finishes execution. This is an implementation of the classic *fetch-decode-execute* paradigm. For each encounter of a method signature, the JVM checks whether this method has been JIT-compiled already; if so, it executes the natively compiled code. If the method has not been JIT-ed yet, it increments a counter representing the number of times this method has been encountered in the runtime of the user's application. The JVM then checks to see if this counter has passed a threshold. If not, the bytecode for a method falls-back to being interpreted by the JVM's C-based interpreter. Otherwise, the JVM starts JIT compilation of this method, and once the method has been JIT-ed, the JVM puts a pointer in the method's class structure to the native code [18]. This method is no longer interpreted during future encounters and executes the optimized version instead. This JIT compilation of the method happens asynchronously in a separate JIT compilation thread to avoid hanging the user application's mutator thread [56, 57]. After initiating the JIT compilation, the bytecode loop continues invoking the JVM's interpreter and proceeds with the next bytecode. Once the JIT compilation thread is finished JIT-ing the method, subsequent encounters execute the JIT-ed code instead of interpreting.

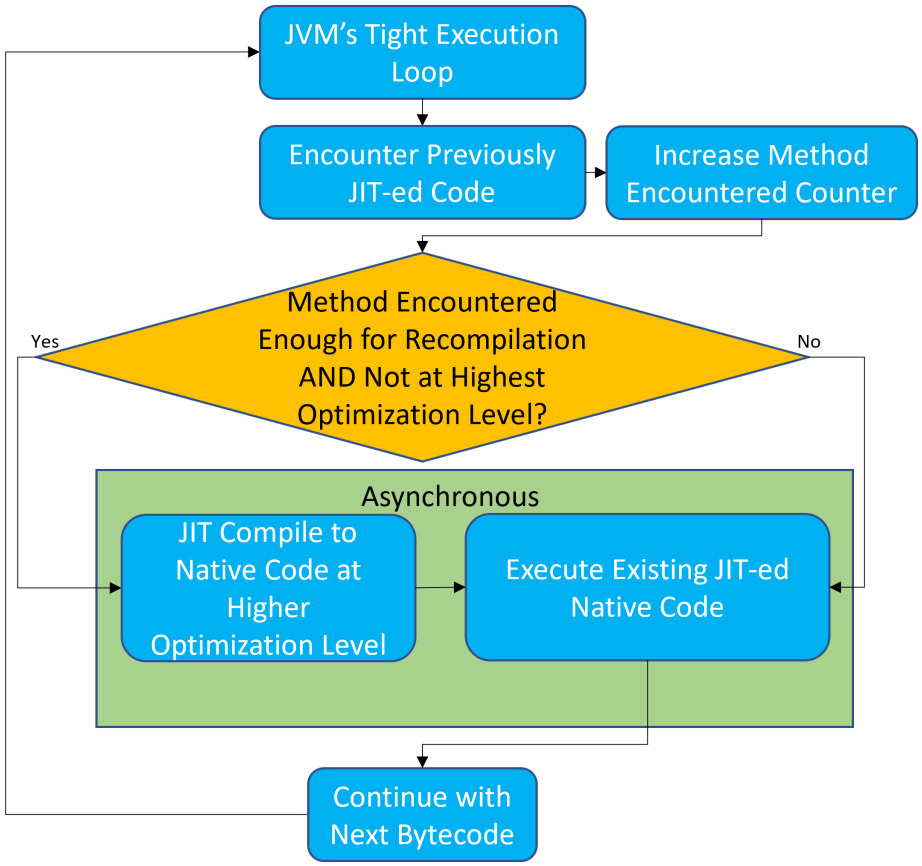

Figure 2.5: The JVM's recompilation loop for interpretation and execution of JIT-compiled code.

If a method is encountered enough it is re-compiled to a higher tier of compilation. Figure 2.4 shows us the levels of tiered compilation in the JVM's JIT compiler. The tiers start with *Interpreted/noOpt*; this results in the method either being directly interpreted or being JIT-compiled at a level of no-Optimization (noOpt). This *noOpt* level of JIT compilation often uses a simple template compilation mechanism, where the form of the method is identified and the equivalent non-optimized code is substituted [58, 59]. The next tiers of compilation are *cold*, *warm*, *hot* and *very-hot*. At each of these levels, the JIT re-compiles the previously JIT-ed code at levels of increasingly aggressive optimization. After the *very-hot* tier, a special *very-hot profiling* level of compilation is employed. This profiling phase inserts profiling code into the JIT-ed native code of the *very-hot* tier, which provides meta-data on subsequent executions of that method. Once sufficient profiling metadata is recorded, a final tailored *scorching* tier is JIT compiled based on the data. This form of tiered compilation, and re-compilation, is an important process for selectively minimizing CPU and memory overhead. This is the classic time/space vs. optimization compromise. The JVM uses these counters/thresholds for initiating the specific levels of optimized compilation. Figure 2.5 shows us the workflow for deciding if re-compilation to a higher level of optimization is warranted in the JVM's JIT compiler. The JVM assumes that if a method has been encountered n times, it is worth spending the time to compile and using the space to store a more optimized version of the method, as the method is likely to be encountered many more times.

2.4 Evaluation

In this last section, we discuss some of the benchmarks chosen (Subsection 2.4.1)—DaCapo (Subsection 2.4.1.1), SciMark (Subsection 2.4.1.2), SPECjvm®2008[1] (Subsection 2.4.1.3)

[1]SPECjvm® and SPECjbb® are registered trademarks of Standard Performance Evaluation Corporation (SPEC®)

and the Renaissance Suite (Subsection 2.4.1.4). We will also look at Linux Performance "Perf" Tools (Subsection 2.4.2).

2.4.1 Benchmarking

In this sub-section, we discuss the benchmarks and why they were chosen.

2.4.1.1 DaCapo Benchmarks

The DaCapo benchmark is a very popular set of workloads used in evaluating implementations and incremental improvements of a JVM [60, 61]. A key goal of DaCapo is its focus on "real-world" applications over synthetic benchmarks that some other suites use. DaCapo also prioritizes the low barrier to entry for use and for gathering measurements. The selection of included applications is meant to provide coverage of a large number of real-world applications [60, 62]. The workloads include avrora (sensor network simulator that performs accurate event simulation in a parallel fashion), fop (print formatter that is output-independent), h2 (an in-memory *Java* database), jython (*Java*-based Python interpreter), luindex (text indexer), lusearch (text searcher), pmd (*Java* source code analysis), sunflow (ray tracing graphical renderer) and xalan (XSLT processor to aid in transforming XML documents) [60, 62].

2.4.1.2 SciMark Benchmarks

The SciMark suite is composed of a number of mathematical micro-benchmarks [63–65]. The SciMark mathematic micro-benchmarks consist of the following mathematical algorithms: Fast Fourier Transform (FFT), Jacobi Successive Over-Relaxation (SOR), Monte Carlo, Sparse matrix multiply and dense LU matrix factorization [63–65]. SciMark comes in two readily available implementations: a *Java* implementation and a native C implementation. This provides a unique opportunity, where the C

statically-compiled version can be taken as a baseline. Then the *Java* version executed by a JVM, and its JIT, can then compare a dynamically compiled implementation back to it. This way, we can analyze incremental performance improvements in a JVM and a JIT to the native version.

2.4.1.3 SPECjvm®2008 Benchmarks

SPECjvm®2008[2] is an industry standard benchmark in the JVM field [61, 66, 67]. SPECjvm®2008 and its fore-bearers have been used as the standard measuring stick on incremental JVM optimizations and improvements. It consists of synthetic benchmarks that allow developers to evaluate the application profile (e.g., compute-intensive, database heavy, etc.) of their program. This allows the developers finer control of the "shape" of their benchmarking applications. Nevertheless, it has the drawback that, in some cases, these synthetic benchmarks are not representative of real-world scenarios.

2.4.1.4 Renaissance Suite

The Renaissance benchmarking suite is a modern open-source benchmarking suite with a wide range of benchmarks that have a particular focus on modern applications, i.e., applications that target specific Java paradigms (e.g., streams, lambdas, futures, etc.), or common Java workload types [68–71]. Many of the workloads in the Renaissance Suite are parts of common frameworks and familiar systems, i.e., Apache Spark, Java Util Concurrent (JUC), databases (Java in-memory and Neo4J), Scala (compiler, collections, ScalaSTM framework), Twitter Finagle, etc. [72–77].

[2]SPECjvm® and SPECjbb® are registered trademarks of Standard Performance Evaluation Corporation (SPEC®)

2.4.2 Perf

Perf, or Perf tools, is a set of Linux performance tools [78]. Perf gives access to hardware and software performance counters at the Kernel level [79]. These counters provide insight into running software's: memory usage, cache misses, CPU cycles, trace-points, and more. These capabilities lend Perf to being utilized for light-weight performance profiling of Linux software systems [80]. Perf is a standard tool used for evaluating JVM performance and identifying potential performance bottlenecks in a managed runtime [81, 82].

Chapter 3

Design of the AArch64 JVM and TRJIT

This chapter discusses the various relevant components within the Eclipse OpenJ9 and Eclipse OMR repositories and the necessary design and implementation steps to enable AArch64 platform support in OpenJ9 and OMR. OMR and the OpenJ9 JVM already have runtime support for AArch64 in the form of interpretation, however, we focus on improving this performance by implementing the JIT compiler. In Section 3.1, we discuss the many challenges and opportunities that lie in porting the TRJIT compiler in OMR and OpenJ9 to the AArch64 platform from the other existing TRJIT compiler platforms. In Section 3.2, we discuss an overview of the different phases of the TRJIT and its code generation.

3.1 From Port to Port

OMR's, and by extension OpenJ9's, TRJIT supports many underlying hardware and operating system platforms. The x86 (64-Bit and 32-Bit) TRJIT compilers are in the same source tree in OMR and are known as the X compiler(s) [83]. The Power PC

(Big Endian—ppc64—and Little Endian—ppc64le)[1] TRJIT compilers collectively are known as the P compiler(s) in OMR [84, 85]. The System-Z/s390(x) TRJIT compiler is known as the Z compiler in OMR [86].

There is also an ARM (32-Bit) TRJIT compiler that was ported from IBM SDK 8 to OpenJ9 OpenJDK11 [87]. Our new 64-Bit ARMv8-A TRJIT compiler is known as the AArch64 compiler in OpenJ9 and OMR [88].

A challenge we had is that there was no design and architecture up-front for bringing AArch64 platform support to OMR, OpenJ9 and TRJIT. We created the design ourselves in parallel to the implementation work; as the implementation progressed. This provided the freedom for creativity and the capacity to emulate the best constituent parts of each of the X, P, Z and 32-Bit ARM TRJIT compiler back-ends. This also had the challenge that in some cases we had to choose between differing, but equally appropriate and mutually exclusive, designs in the other (X, P, Z, etc.) compilers for particular components. Some of these choices caused later component designs to be eliminated. Often this resulted in an earlier design decision dictating the choice for further design decisions. Some of these decisions in the design and implementation were also dictated by hardware, software and Application Binary Interface (ABI) in the AArch64 architecture. For these reasons differing designs were not implemented to be compared amongst themselves. The different TRJIT compilers and code generators (codegens) were written by different groups of people, and so have very different coding styles [89, 90].

3.2 Eclipse OMR TRJIT Design

Figure 3.1 illustrates a high-level view of the phases of TRJIT. In the following we provide a more detailed view of the different components and phases of TRJIT. The

[1]Endianness represents the ordering of bytes in computer memory and CPU registers [84]. Ordering in the form of most significant bytes in the smaller address to least significant is Big Endian and the reverse is Little Endian.

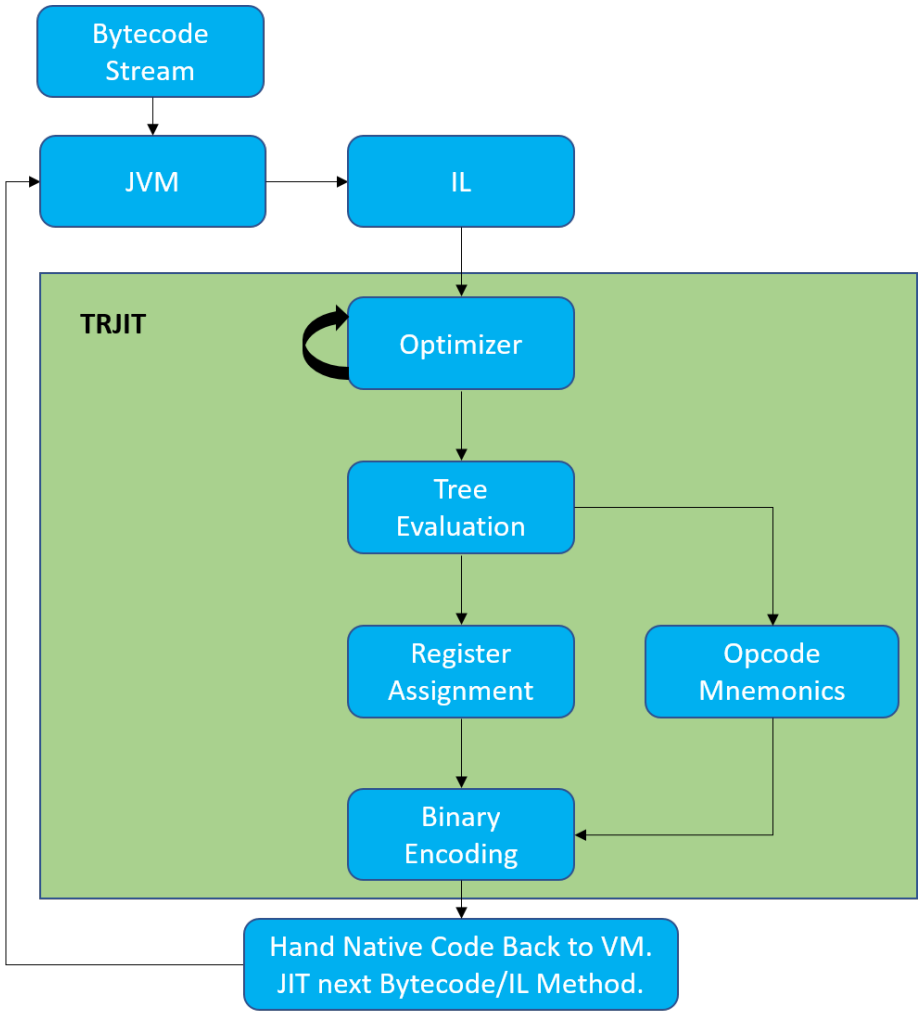

Figure 3.1: Overview of TRJIT Compiler Phases.

JVM takes in a bytecode and converts it to Intermediate Language (IL), which gets passed into the TRJIT compiler (see Subsection 3.2.1 for more on the IL).

The first phase of TRJIT is the optimizer. Once the IL is optimized then it is passed to the tree evaluators (see Subsection 3.2.2 for more on tree evaluation). Tree evaluation evaluates the IL in concert with swapping out IL for opcode mnemonics (see Subsection 3.2.4 for more on opcode mnemonics). Next the IL is handed off to register assignment (see Subsection 3.2.6 for more on the infinite virtual register set and register assignment). Finally, the opcodes in the IL are converted to platform-native binary encoding to be handed off to the JVM for execution (see Subsection 3.2.5

for more on binary encoding).

3.2.1 OMR Intermediate Language (IL)

There is a requirement for a common language to communicate the application's virtual instructions, from the user-facing language (e.g., *Java*) through to OMR; and vice versa. For this purpose, OMR and TRJIT use an Intermediate Language (IL). Methods or functions in the front-end language are converted into trees of OMR IL [91]. Listing 3.1 shows the virtual instructions for the arithmetic expression $a = \frac{a-b}{c}$ [91] and Figure 3.2 illustrates the graphical representation for the corresponding IL tree.

```
1    n1  istore <a>
2    n2    idiv
3    n3      isub
4    n4        iload <a>
5    n5        iload <b>
6    n6      iload <c>
```

Listing 3.1: Virtual instruction representation of arithmetic expression: $a = \frac{a-b}{c}$.

Each sub-operation (i.e., method) is grouped into a sub-tree, with operands being the leaf nodes, and the root of the tree being called a treetop. OMR receives IL as a stream of treetops that are processed sequentially. IL is usually generated by the ILgen tool, either manually from the front-end language or via the JitBuilder API. Each VM interacts differently with OMR in how it passes IL. Everything in a sub-tree is, for the most part, self-contained. So in a limited manner, sub-trees represented by treetops are handled flexibly. OMR optimizes the compilation of sub-trees by controlling which treetops are processed first. In a controlled way, OMR reorders the treetops without affecting the functional correctness of the user's application. A basic block is a code sequence that is self-contained with no branches except for the entry and exit points [18]. This structure of OMR IL treetops is still related to basic

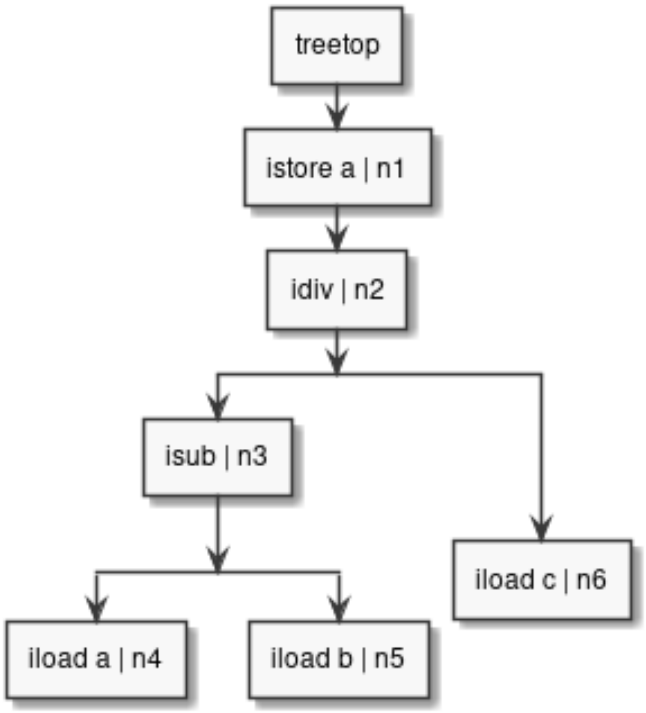

Figure 3.2: Graphical representation of the IL tree for the arithmetic expression: $a = \frac{a-b}{c}$.

blocks in VM/JIT nomenclature [18, 91]. The OMR IL basic block is extended in that multiple nodes can be referenced multiple times [91].

3.2.2 OMR Tree Evaluators

The first main phase of the TRJIT compilation process is tree evaluation. The tree evaluators operate on the stream of IL treetops. In this phase, the IL trees are evaluated and converted into their virtual instruction forms (see Subsection 3.2.4). This is a depth-first process starting at the leaf nodes and evaluating back up the tree to the treetops. Each tree evaluator corresponds to some part of an OMR IL operation. The deepest operations, or sub-operations are handled first as the values of their arguments are already in place; as constants or as the results of operations in previous treetops.

As there are many evaluators, for readability they are broken up into separate source

files by operation type. For example, operations that take one argument (e.g., logical operations) reside in the UnaryEvaluator file, operations that take two arguments reside in the BinaryEvaluator file, operations that handle the control-flow of the program source reside in the ControlFlowEvaluator file and operations that process Floating-Point numbers reside in the FloatingPointEvaluators file, etc. [92–95]. Each evaluator determines what kind of, and how many, virtual registers are required for the operation. If existing virtual registers from previous/sub-operations can be re-used, they are re-used to reduce overhead. However, if preexisting virtual registers are not available, then a new virtual register is allocated at this time (see Subsection 3.2.6 for more information in register allocation and re-use).

Once the virtual registers have been processed, the next step is to determine which operation needs to be performed and which AArch64 virtual instruction/mnemonic has to be assigned (see Subsection 3.2.4 for more information on opcode mnemonics). Depending on the sizes of the arguments and the desired capacity of the return type, different instructions are chosen. Once the appropriate instruction is chosen, the tree evaluator applies this information, also known as code generation, to the current/parent node in the IL tree on which this evaluator is operating.

Before returning, it is a convention for an evaluator (the parent node) to decrement the "in-use" reference count of its child nodes. If a child node's counter goes to *0*, then that child IL tree node is collected by TRJIT. It is against the convention for an evaluator to decrement itself; the parent always decrements its children instead.

These tree evaluators are one of the main locations for optimizations: choice of virtual instructions, number and types of virtual registers, how to generate the code, etc. all have significant side-effects in terms of performance and overhead of the application run. This makes evaluator implementation a key part of creating a performant AArch64 TRJIT.

This tree evaluation process happens continuously for each operation/sub-operation

passing the results up the tree from a parent node as a child for the calling evaluator. Once the call chain returns to the treetop all of the necessary code to perform, this sub-tree has been generated and the evaluation process commences again on the subsequent treetop until the user's program finishes execution.

We implemented a number of arithmetic, logical, control-flow and helper evaluators [A12–A14, A16–A40].

3.2.2.1 OpenJ9 Overridden Tree Evaluators

The tree evaluators in TRJIT are front-end language (e.g., *Java*) agnostic. This makes implementing a front-end language much easier in that one does not have to re-implement the majority of evaluators for common operations. However, there may be scenarios where knowing front-end language-specific semantics, one could develop a particularly optimized tree evaluator.

For this purpose Eclipse OpenJ9 contains some of these overridden tree evaluators for *Java* and hosts them in the J9TreeEvaluators file [96]. Another sub-set of evaluators that do not live in OMR are those that are specific to the *Java* language [96]. Some operations may be language specific and it does not make sense, or may not be possible, to have a generic/language-agnostic version live in OMR. Some *Java*-specific examples of evaluators that are overridden in OpenJ9 are synchronization, read/write barriers and division checks (e.g., divide by *0* and *Java* exception handling in that case) [96].

3.2.3 OMR Tril Tests

Tril tests (*compilertriltest*) is a testing framework within the OMR project that is built on top of the *GoogleTest* unit testing framework [97–99]. Tril tests in OMR became an important tool for our team to verify our implementation work. Tril tests are front-end, language agnostic e.g., *Java*. They target the OMR IL directly, which

enables us to directly test our AArch64 compiler's Tree Evaluator implementation (more on Tree Evaluators in Subsection 3.2.2) [100].

OMR also contains underlying platform-specific functional verification tests (*fvtests*), which enable custom targeting and extreme specificity to the underlying compiler's platform (e.g., X, P, etc.) [101]. The downside of the *fvtests* framework is that a new compiler would have to implement the totality of included tests specifically for that platform (i.e., AArch64).

In contrast, tril tests enable us to use all of the existing platform-agnostic tests as an existing functional metric for our AArch64 TRJIT implementation. Tril hooks into an entry point, specific to the data types of the arguments and return type, in the OMR test compiler used by the *fvtests* framework [101]. Tril generates tests in the form of an IL tree and inserts the provided arguments. The Tril tests contain a series of small *C* functions that are functionally equivalent to the TRJIT's implementation for the IL opcodes being tested. The arguments for the test are passed into both the TRJIT entry point and the local *C* function. The return values are then compared against each other for validity using a *GoogleTest* built-in equivalency function. If the two values are equivalent, this Tril test is considered passed.

3.2.4 Instruction OpCode Mnemonics

For developer convenience, OMR and OpenJ9 provide human-readable mnemonics, labels or virtual instructions, instead of using the raw AArch64 instructions or binary encoding in the TRJIT compiler source. This makes the TRJIT code, especially in the Tree Evaluators (see Subsection 3.2.2), much easier to read and write. The ARMv8-A, AArch64, instruction set architecture (ISA) has its own mnemonics as defined in the reference manual [102]. ARMv8-A overloads the same label many times but differentiates the operation to perform by the types/sizes of the argument(s) passed in.

Some sample ARMv8-A instructions are shown in Listing 3.2 [102]. Here we see the usage of 'w's and 'x's to indicate whether the operation is 32-bit or 64-bit, the first for a 32-bit arithmetic add operation and the second for a 64-bit arithmetic add operation [102, 103]. The two OMR opcode mnemonics seen in Listing 3.3 correspond to the ARMv8-A instructions in Listing 3.2 [103].

```
1    ADD W0, W1, W2 // Add 32-bit registers
2    ADD X0, X1, X2 // Add 64-bit registers
```

Listing 3.2: ARMv8-A instruction mnemonics for 32-bit arithmetic add and 64-bit arithmetic add operations.

```
1    addimmw // Add 32-bit integers
2    addimmx // Add 64-bit integers
```

Listing 3.3: OMR instruction mnemonics for 32-bit arithmetic add and 64-bit arithmetic add operations.

We implemented the initial set of opcode mnemonics for the JVM [104, 105].

3.2.5 Binary Encoding

As discussed in Subsection 3.2.4, the AArch64 ISA and OMR uses mnemonics for readability and writability during development. However, the ARMv8 CPU does not execute these mnemonics directly; instead the CPU operates on binary encoding [102]. Binary encoding is the raw version of an instruction to be executed by the CPU [102]. In Listing 3.4 we see the binary encoding (in hexadecimal) for the same two 32-bit arithmetic add and 64-bit arithmetic add operations seen in Listing 3.2 and Listing 3.3. During the binary encoding processing phase of the TRJIT compiler, any necessary registers containing the arguments for the corresponding operation, as well as the register to store the return data and any options (e.g., how to handle any carrys in the case of an arithmetic operation), or any relevant condition codes that should be checked are masked (logical AND/OR-ed) into the *000000* (as seen in Listing 3.4)

section of the binary encoding. This masking occurs before the instruction is sent to the CPU for execution.

```
1    0x11000000,    /* ADD    addimmw    */
2    0x91000000,    /* ADD    addimmx    */
```

Listing 3.4: ARMV8-A binary encoding in OMR for 32-bit arithmetic add and 64-bit arithmetic add operations.

We implemented the initial set of ARMv8-A binary encodings for the JVM [104, 105].

3.2.6 Infinite Virtual Register Set and Register Assignment

As mentioned in Subsection 3.2.2, during TRJIT's tree evaluation phase, virtual registers are allocated to hold the values of arguments and return values for an operation. During this tree evaluation phase, care is taken to maximize reuse of previously allocated registers. While evaluating the IL trees, TRJIT does not concern itself with the number of available physical registers in the CPU of the underlying platform. Instead, TRJIT utilizes the concept of an *infinite virtual register set*. TRJIT allocates a pseudo-infinite number of virtual registers for the purpose of tracking and holding arguments and return data.

However, the underlying architecture and platform contain a finite number of physical registers. Usually this small set of physical registers is even further broken down into even smaller sub-sets of general purpose registers (GPRs), floating-point registers (FPRs) and more specialized registers like vector purpose registers (VPRs). After the tree evaluation phase, OMR's TRJIT performs the register assignment phase. TRJIT takes these virtual registers and assigns them to physical registers on the system. If TRJIT encounters a situation where there are no available physical registers of an appropriate type (e.g., GPR, FPR, etc.), it spills the contents of an existing, but not immediately required, register to the stack for storage to free up a register

for the current operation. After the current operation has been completed, TRJIT re-loads the value stored on the stack back into the register it came from to preserve the previous state for future operations. We implemented the initial version of the register assignment for AArch64 for the JVM [104, 105].

Chapter 4

Evaluation of the AArch64 JVM and TRJIT

In this chapter, we discuss the evaluation of the JVM on AArch64. In the sub-section about the experimental setup (Section 4.1), we discuss the devices used and the framework used for testing and benchmarking. In the final section, we discuss the results of the evaluation of the JVM (Section 4.2).

4.1 Experimental Setup

We tested and benchmarked our work with AdoptOpenJDK's OpenJDK11 OpenJ9 Early Access release (EA) 0.23, that contains our design and implementation [106, 107]. We performed benchmarking runs on OpenJ9's AArch64 and x86-64 implementations, for comparison. The JVM command-line options used when running OpenJ9 are shown in Listing 4.1.

```
1 java -Xms2G -Xmx2G
```

Listing 4.1: Java command-line options used during OpenJ9 runs.

Table 4.1: Beelink BT3 Pro II x86-64 Mini PC specifications.

Graph Label	**x86-64**
Manufacturer	Beelink
Name	BT3 Pro II Mini PC
Architecture	x86-64
Core	4-core Intel X5-Z8350
Max Speed	1.92GHz
Locked Speed	1.44GHz
Cache	L1-32KB L2-2MB
Memory	4GB DDR3@1600MHz

Table 4.2: Pine64 Rock64 A53-based AArch64 embedded device specifications.

Graph Label	**A53**
Manufacturer	Pine64
Name	Rock64
Architecture	ARMv8-AArch64
Core	4-core A53
Max Speed	1.5GHz
Locked Speed	1.5GHz
Cache	L1-32KB L2-256KB
Memory	4GB DDR3@1600MHz

As previously mentioned, for industry standard benchmarks we are using DaCapo (Subsection 2.4.1.1), SciMark (Subsection 2.4.1.2), SPECjvm®2008 (Subsection 2.4.1.3) and Renaissance Suite (Subsection 2.4.1.4) on both AArch64 and x86-64.

Table 4.3: Khadas VIM3 big.LITTLE A73/A53-based AArch64 embedded device specifications.

Graph Label	**A73**
Manufacturer	Khadas
Name	VIM3
Architecture	ARMv8-AArch64
Core	4x A73-2x A53
Max Speed	A73-2.2GHz A53-1.8GHz
Locked Speed	1.48GHz
Cache	L1-32KB L2-1MB (shared)
Memory	4GB DDR4@3200MHz

Table 4.4: Raspberry Pi 4 B A72-based AArch64 embedded device specifications.

Graph Label	**A72**
Manufacturer	Raspberry Pi
Name	4 B
Architecture	ARMV8-AArch64
Core	4x-core A72
Max Speed	1.5GHz
Locked Speed	1.5GHz
Cache	L1-48KB L2-1MB (shared)
Memory	8GB DDR4@3200MHz

4.1.1 The Devices

The devices in Tables 4.1 to 4.4 are used for the development, testing, debugging and benchmarking in this thesis.

- The *Beelink BT3 Pro II Mini PC* is an x86-64-based device running at 1.92GHz on the Debian OS with Kernel version 5.7.10-1. The reason an x86-64 mini PC is included in the devices and results is to provide a comparison to a more mature Interpreter and JIT implementation in the OpenJ9 JVM.

- The *Pine64 Rock64* is an A53-based device running at 1.5GHz on the Arch Linux ARM OS with Kernel version 5.3.8-2-ARCH.

- The *Khadas VIM3* is a device using 4x A73 cores and 2x A53 cores in a big.LITTLE configuration running at 2.2 GHz (A73) and 1.8 GHz (A53) on the Debian OS with Kernel version 5.5.0-rc2 [108–110]. Of note, the SoC in the VIM3 is equivalent to the ODROID-N2 board with 4GB of RAM.

- The *Raspberry Pi 4 B* is an A72-based device running at 1.5GHz on the Ubuntu 20.04.1 LTS OS with Kernel version 5.4.0-1025-raspi.

Running our experiments on a variety of devices gives us greater insight into our results. We are looking at an x86-64 board, which has comparable specifications to the AArch64 boards, to be able to see how the AArch64 JIT in its current

state performs against the more mature x86-64 JIT. Having the Rock64, VIM3 and Raspberry Pi 4 B gives us a wide spectrum of examples of different AArch64 architecture implementations. The Rock64 gives us a good example of a low power (A5x) device. The Raspberry Pi 4 B gives us a good example of a high performance (A7x) device. The VIM3 gives us a good example of a big.LITTLE heterogenous core implementation.

Sometimes, depending on the device, manufacturers or integrators undervolt, or underclock, the CPUs and sometimes they overvolt, or overclock, the CPUs. This makes it hard to compare one board against another in an "apples to apples" manner. However, by locking the speeds (see Subsection 4.1.2), we reduce the variance in CPU clock speeds for comparable results.

The devices we use for benchmarking have the same capacities of RAM at 4GB (except for the Raspberry Pi 4 B, which has 8GB), but differing RAM speeds. This difference in the total size of memory could affect the benchmarking results if we fail to control the amount of RAM that the Java processes are able to use. Therefore, to mitigate this issue we have locked the JVM to only be able to use 2GB of heap via the *-Xms* and *-Xmx Java* command-line arguments (see Listing 4.1). The Beelink and the Rock64 have DDR3 RAM at 1600MHz. The VIM3 and the Raspberry Pi 4 B have DDR4 RAM at 3200MHz. This certainly could affect a direct comparison, but is interesting to give a comparison between low-power vs. high-performance devices.

4.1.2 The Benchmarking and Testing Framework

We created our own benchmarking and testing framework for ease of maintainability, reproducibility, finer granularity of control of individual benchmark options and to allow for quick benchmarking setup across disparate devices [19–21]. This framework handles gathering of the individual benchmarks (DaCapo, SciMark, SPECjvm®2008 and the Renaissance Suite). It also handles the acquisition of a number of dependencies

to run and test the OpenJ9 JIT on the aforementioned platforms.

Another important responsibility this framework handles, for comparability, is locking each different device's CPU frequency to a value that affords comparison. The framework has a main driver script that allows the execution of each benchmark serially. Once benchmarks have run, the framework then handles parsing and aggregating results in a layout that enables ease of analysis and graphing [111, 112].

4.2 Experimental Evaluation

As previously mentioned, in this thesis we look at benchmarking results from the Beelink (Table 4.1), Rock64 (Table 4.2), VIM3 (Table 4.3) and Raspberry Pi 4 B (Table 4.4) devices. We examine results from the DaCapo (Figure 4.1), SPECjvm®2008 (Figure 4.2), SPECjvm®2008 Startup (Figure 4.3), SPECjvm®2008 SciMark (Figure 4.4) and Renaissance Suite (Figure 4.5) benchmarks that were gathered using the aforementioned benchmarking and testing framework (see Subsection 4.1.2).

All of the results and graphs in Subsection 4.2.1, Subsection 4.2.2 and Appendix B are using the OpenJ9 0.23 release JVM. The devices labled A53, A72 and A73 are running OpenJ9 with our AArch64 TRJIT design and implementation; whilst the device labled x86-64 is running the existing X platform's TRJIT implementation.

The AArch64 JVM and TRJIT being evaluated in this thesis is an early access release that is rapidly progressing toward a production release, and as such only performs some code optimizations. There are a number of more advanced optimizations and performance enhancements for the JVM that are yet to be implemented, which the x86-64 JVM already contains. Some of these outstanding optimizations and performance features to be implemented include inlined array copies, improved SIMD (Single Instruction Multiple Data) support and the usage of a dynamic polymorphic inline cache (PIC) for interface dispatches. Chapter 5 lays out some of these items as

a part of future work.

4.2.1 Results

The overall theme for DaCapo, SPECjvm®2008 and SciMark has the A73 outperforming everything else because it is the stronger core. In the DaCapo benchmark from Figure 4.1, the performance of the A73 based machine often matched or performed better than its x86-64 counterpart. DaCapo benchmarks are memory bound, leading to much better performance when the memory frequency is higher, as in the VIM3 case [68]. The same behaviour with memory bound benchmarks is displayed in the SPECjvm®2008 SciMark large subset; the small subset exercises cache and CPU speed more specifically (Figure 4.2).

The A7x cores perform better in all benchmarks (Figures 4.1 to 4.5) compared to their A53 counterparts. This hierarchy can be attributed to the better class of A7x processors. The A73 core outperforms all other boards in the DaCapo benchmark, aside from h2 where the x86-64 device takes the lead (Figure 4.1). This reversal can be attributed to more advanced hardware overall support in the x86-64 JVM.

There are outliers in the compression and crypto workloads as OpenJ9 on AArch64 is currently using a software implementation rather than hardware support built into the architecture for crypto opcodes. This coupled with differing implementations of the architecture in each core could explain these results.

The mpegaudio workload is also an outlier. This could be attributed to the heavy usage of floating-point up codes and the fact that vectorization is not implemented in the VM.

Xml.validation is also an outlier in that the x86-64 takes the lead over all of the Axx cores. This can be attributed to the fact that this workload relies on moving large amounts of data around and so not having inline arrayCopy support implemented is having a large impact here.

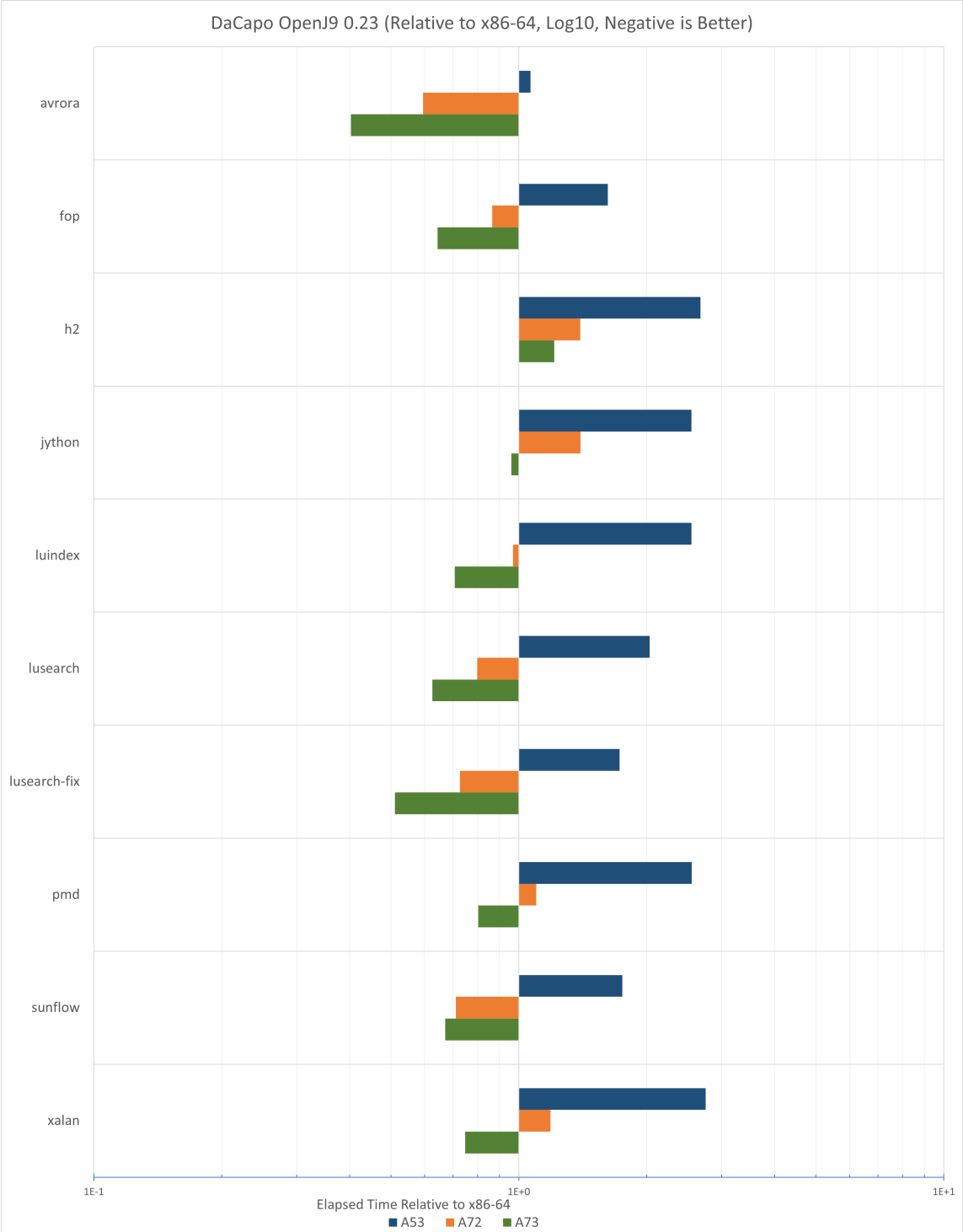

Figure 4.1: DaCapo elapsed time relative to x86-64 device comparison for OpenJ9 0.23 (Log10, negative is better).

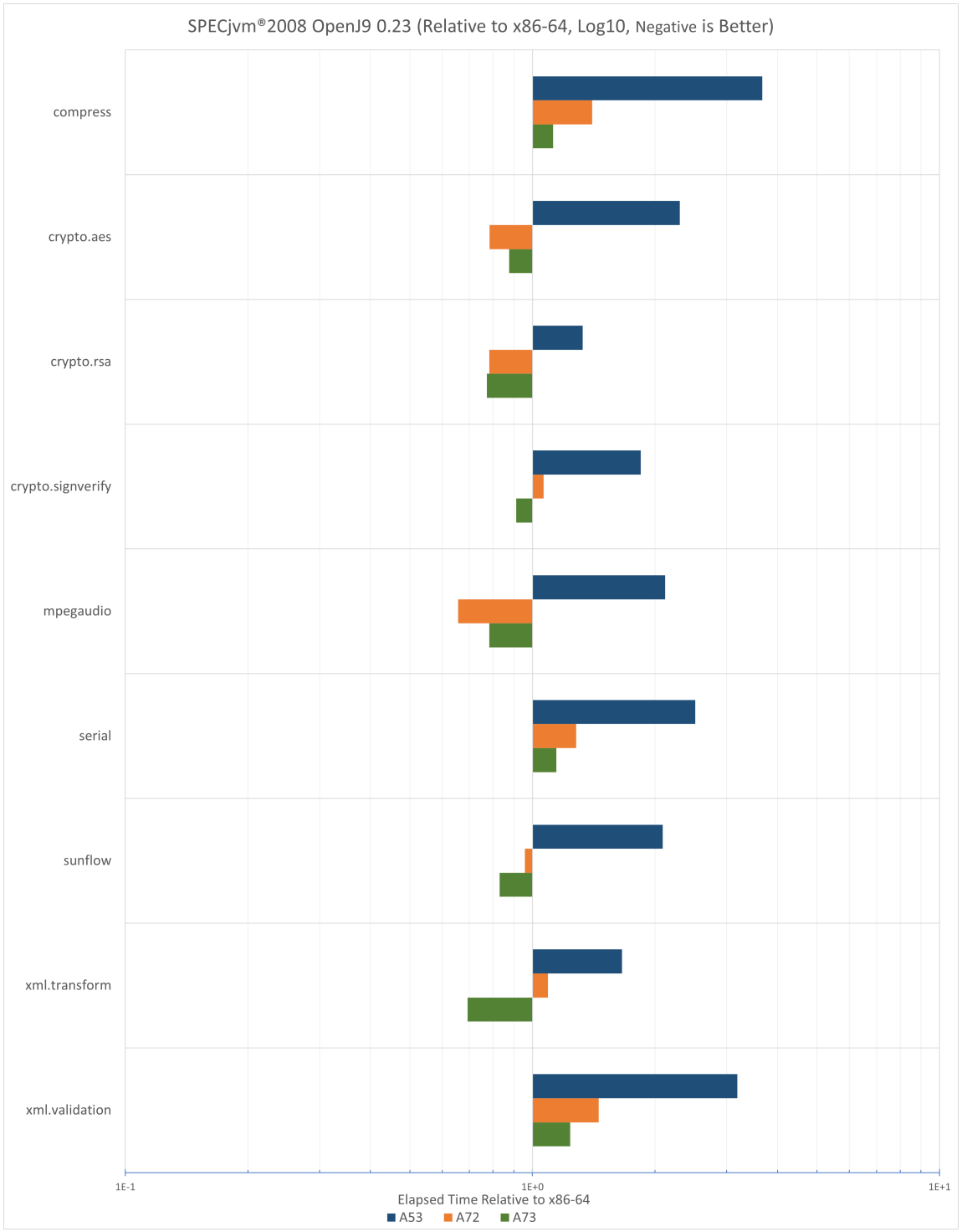

Figure 4.2: SPECjvm®2008 elapsed time relative to x86-64 device comparison for OpenJ9 0.23 (Log10, negative is better).

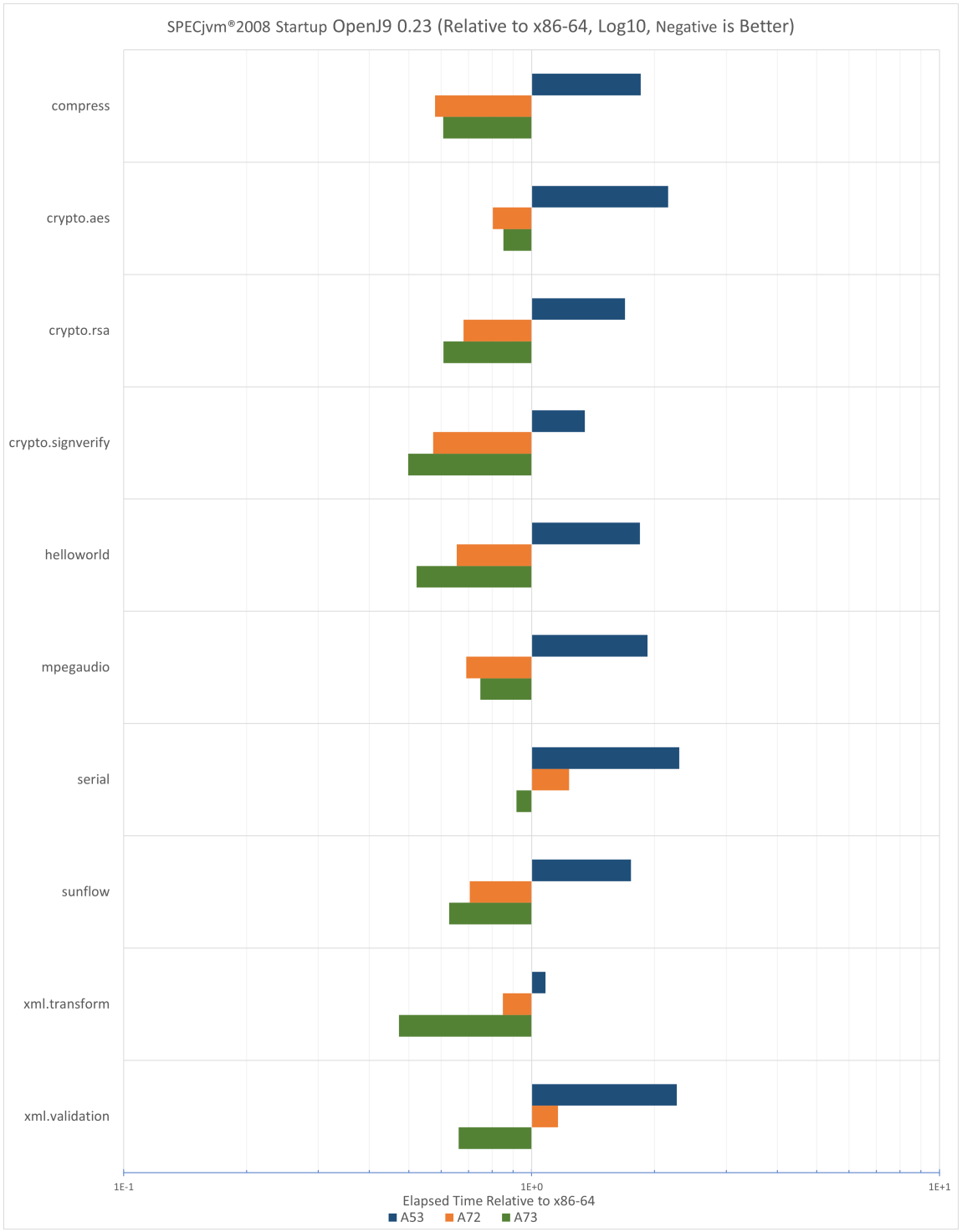

Figure 4.3: SPECjvm®2008 startup elapsed time relative to x86-64 device comparison for OpenJ9 0.23 (Log10, negative is better).

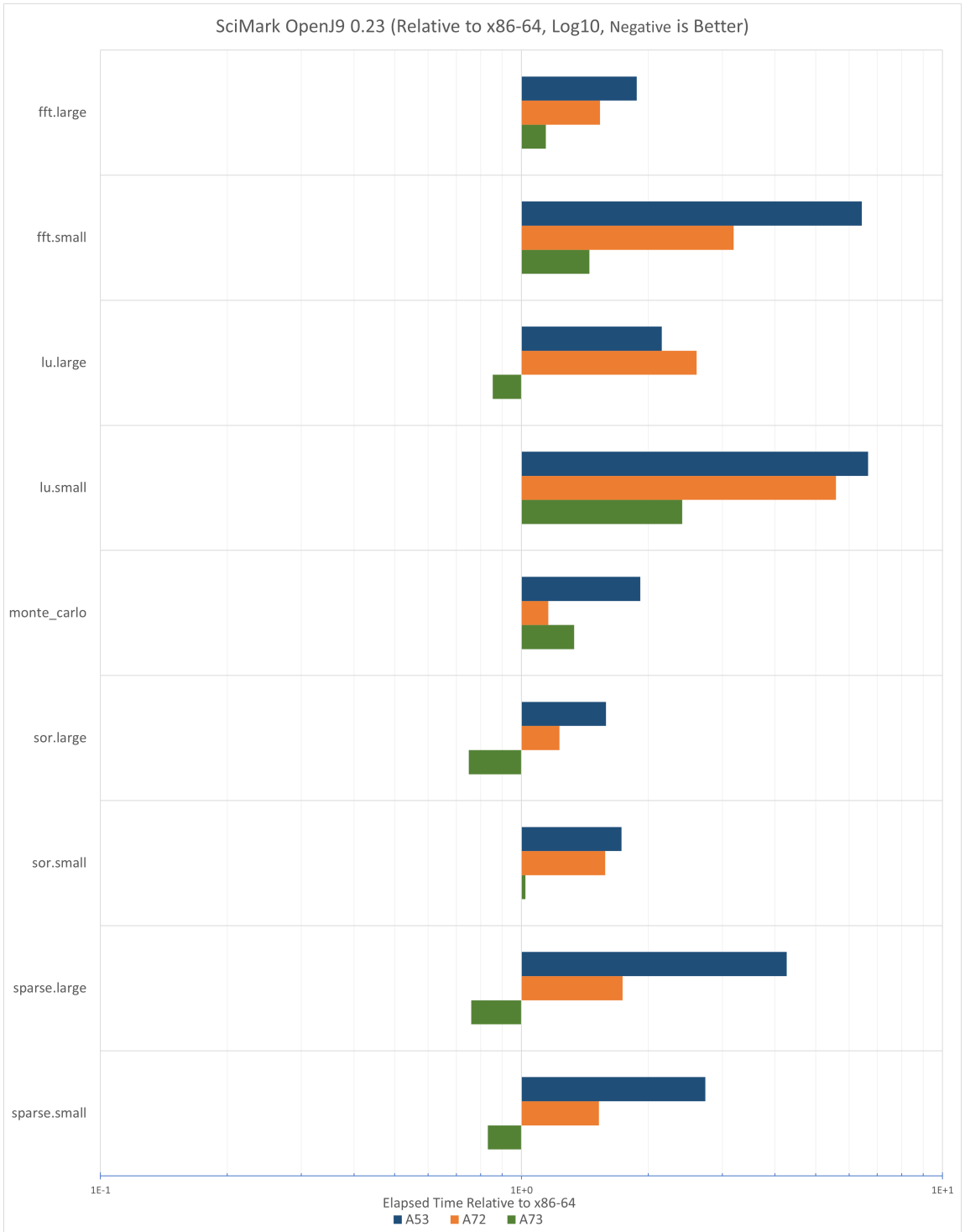

Figure 4.4: SciMark elapsed time relative to x86-64 device comparison for OpenJ9 0.23 (Log10, negative is better).

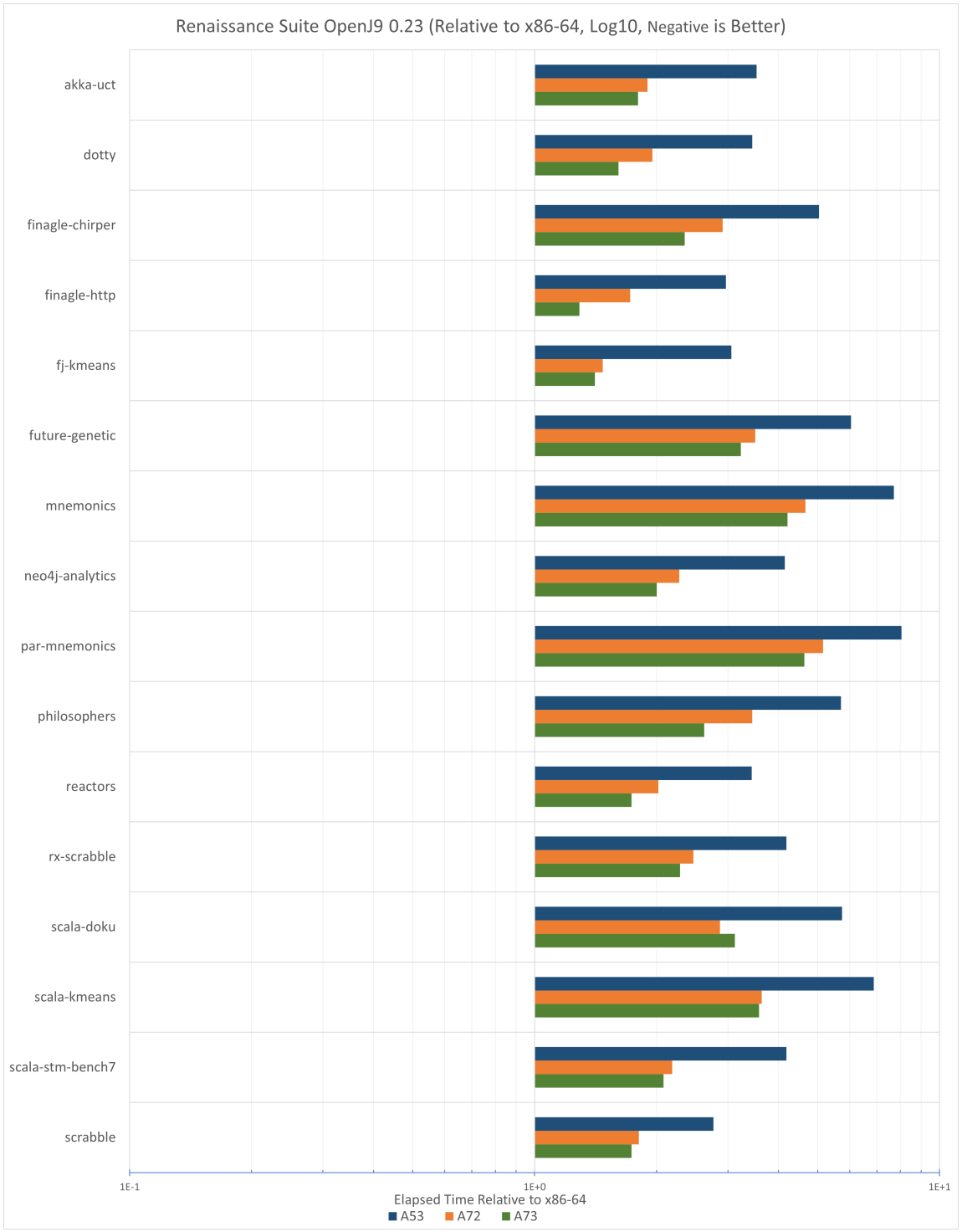

Figure 4.5: Renaissance Suite elapsed time relative to x86-64 device comparison for OpenJ9 0.23 (Log10, negative is better).

The VIM3 and Raspberry Pi 4 B boards outperform all other platforms in most benchmarks in startup (Figure 4.3). This can be accounted for by faster memory speeds (DDR4) and faster disk speeds (150MB/s). Here we see some of the same outliers as in the full-run SPECjvm®2008 workloads with the reversal of the compress workload. This can be attributed to the aforementioned faster disk speeds that cause faster access to the file to compress and the output of the compressed file during the startup phase, compared to the disk speeds being throttled on a full run.

Monte-Carlo, in Figure 4.4, still proved problematic with the A73 board and is to be explored further in future research (see Chapter 5). However, one speculative theory on why we see the results we do for Monte-Carlo could be that the performance limits of the existing floating-point support in the AArch64 JVM are being reached. It could be that this benchmark could benefit from implementing vectorization (SIMD) support in the VM.

In contrast to DaCapo, SPECjvm®2008 and SciMark, the more real-world Renaissance Suite of benchmarks shows the same x86-64 device beating out the A7x cores in all benchmarks (Figure 4.5). As the other benchmarks are more synthetic, and the Renaissance Suite is intended to show production application performance, the more mature x86-64 JVM shows better real-world application performance. As these are more modern workloads, this difference can also be attributed to support for the more modern Java language constructs in the AArch64 JVM being less mature than that of the x86-64. This also could be reflected in that many of the Renaissance workloads are focused on cloud applications. The scala-doku workload has a slight outlier in that the A72 and A73 trend is reversed here. Speculating, this could be caused by the lack of floating-point/vectorization support combined with a lack of inline arrayCopy support in the AArch64 JVM (see Chapter 5).

To reiterate all of the results and graphs in this subsection running on an Axx device are using the OpenJ9 0.23 release JVM with our AArch64 TRJIT design

and implementation. The device labeled x86-64 is running the existing X platform's TRJIT implementation. To summarize, the x86-64 beats all other cores 46% of the time. This is a large portion because the x86-64 JVM is a much more mature implementation. The remaining 53% of the time our AArch64 JVM beats out x86-64 on an Axx core in the majority of workloads. Of that, 83% of the time the A73 core beats all the other Axx cores. The remaining 17% is the A72 core. At an astounding 0%, the A53 fails to take any top result across the totality of the sub benchmarks. This stark contrast in A7x vs. A5x can be attributed to the A7x line of cores being more advanced and powerful across the board.

In Subsection 4.2.2 we will take a more granular look at the AArch64 JVM to see if we can identify bytecodes that may be the root of some of the outliers identified in this section.

4.2.2 Bytecode Granularity Benchmarking

The benchmarks in Subsection 4.2.1 provided much insight into the workings of the JVM and TRJIT and suggest some of the potential performance deficiencies. However, these benchmarks operate at a higher level of granularity. To acquire more insight at a finer-level of granularity, we use a retrofitted version of the *microjit-tests* framework, developed by our colleagues at CAS-Atlantic on the MicroJIT team [56, 58, 113].

The MicroJIT is a lightweight, template-based, JIT compiler designed to bridge the gap during JVM startup when TRJIT has not yet started compiling methods or gathering profiling data [56, 58, 113]. This framework is utilized for the MicroJIT project to perform testing on how individual *Java* bytecodes are compiled by MicroJIT. Their goal is to validate and verify the functional correctness of the MicroJIT-compiled native code. Our goal is not the same as we have other ways of performing functional verification (e.g., tril tests and fvtests Subsection 3.2.3) that are more appropriate

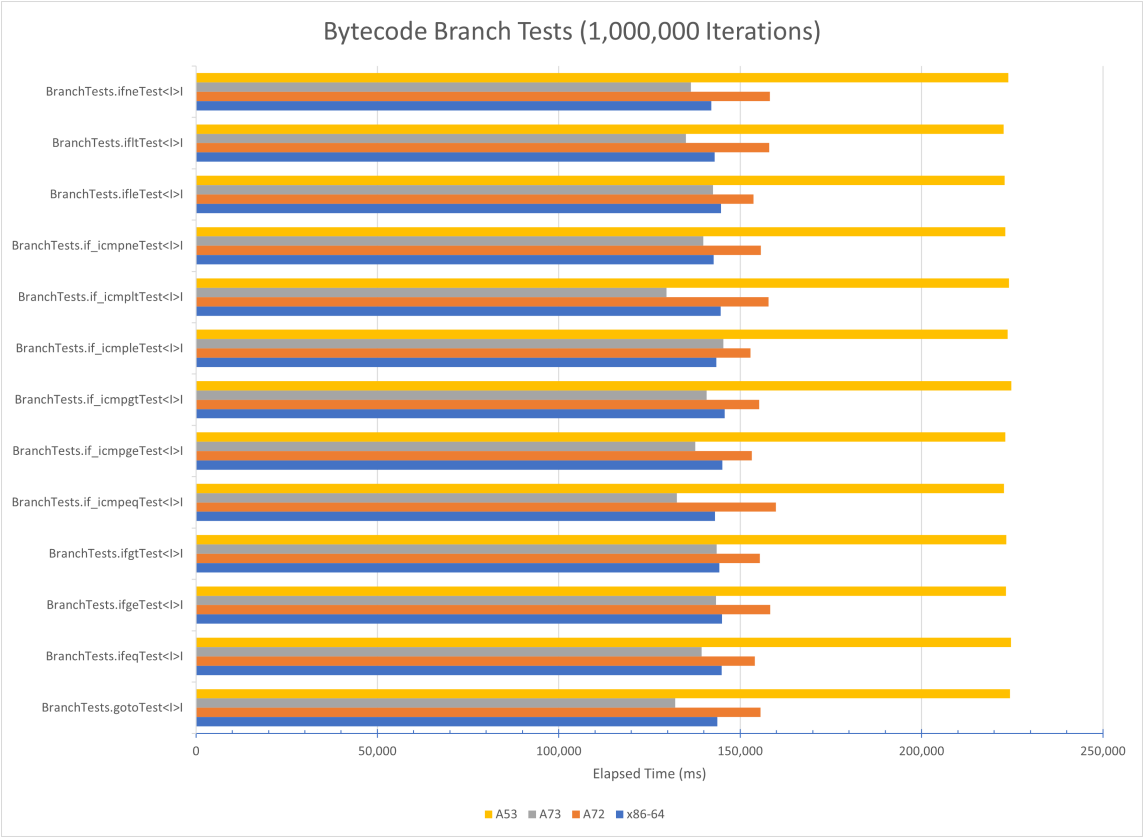

Figure 4.6: Bytecode Branch Tests (1,000,000 iterations).

for the TRJIT. We opt to use the *microjit-tests* framework to aid in a finer-grain performance investigation, hence we retrofitted it. We modified the *microjit-tests* framework to drive increased application load on a bytecode-by-bytecode basis. The goal of this method is to more directly demonstrate which bytecodes, and therefore which OpenJ9 and OMR IL opcodes and evaluators, warrant investigation for further optimization.

The overall trend is that bytecodes on one platform perform the same relative amount on the others, with few exceptions, as can be seen in Figures 4.6 to 4.13.

In Figure 4.12 we see the long tests have outliers in the division, and shift bytecodes. The division outlier can be attributed to the way we are handling the special cases (e.g., divide by 0 exception handling) of division in the division evaluator. The shift outliers are interesting as there is no obvious reason for the evaluator to be slower

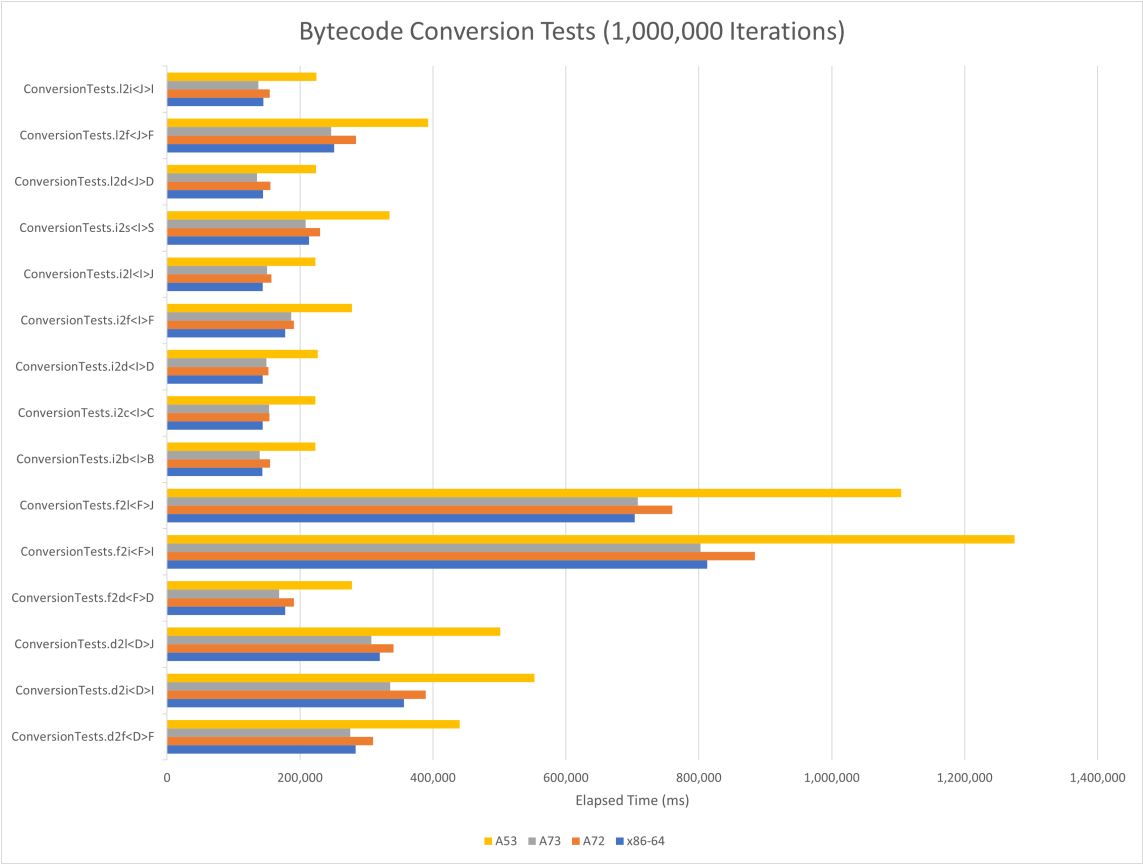

Figure 4.7: Bytecode Conversion Tests (1,000,000 iterations).

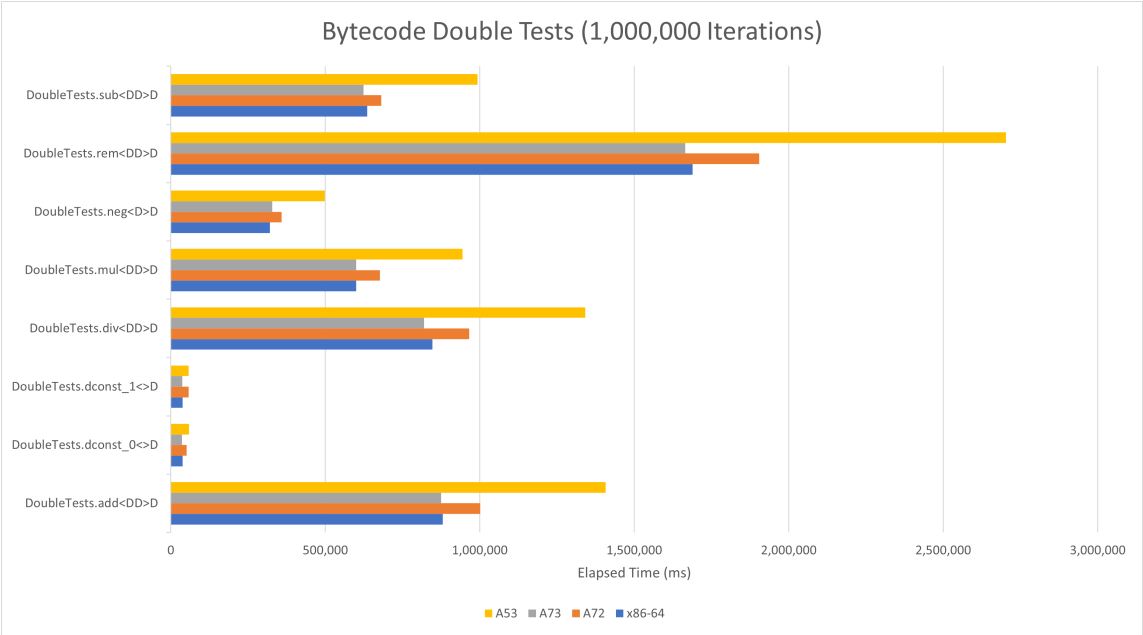

Figure 4.8: Bytecode Double Tests (1,000,000 iterations).

on AArch64 aside from not implementing hardware support. This not the case in the integer tests seen in Figure 4.10, further investigation is required. However, one speculative theory is that our long shift evaluator could be a good candidate for improvement if it is not as efficient as the integer shift evaluator.

As anticipated, in Figures 4.6 to 4.11 and 4.13 the relative performance between boards and between AArch64 and x86-64 results are comparable.

These small number of outliers identified in these bytecode granularity tests will be investigated further and cross-referenced with results from Linux Perf tools (see Chapter 5) to identify their source. However, some speculative theories on possible causes for these outliers have been put forward inline above where they have come up, and future work (Chapter 5) will investigate and confirm these speculations. The raw data corresponding to Figures 4.6 to 4.13 can be seen in Tables 6.1 to 6.8 from Appendix B.

Overall, this early access release of the OpenJDK11 OpenJ9 JVM shows great promise while it is progressing towards a full release. We are currently working on identifying

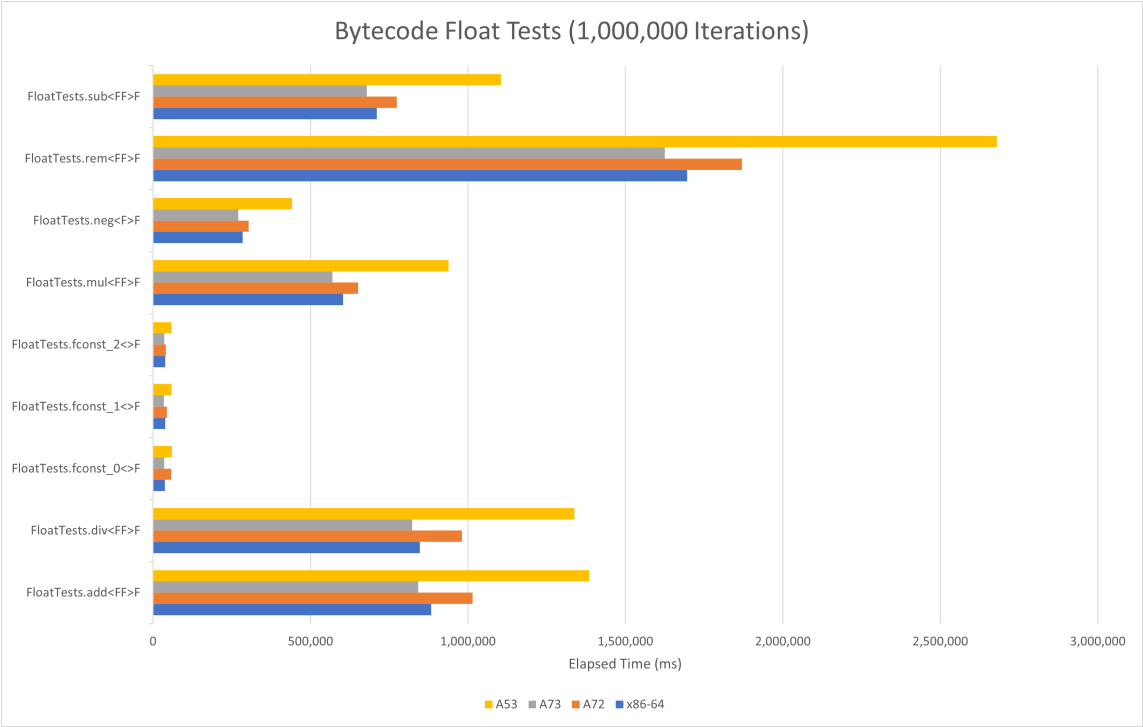

Figure 4.9: Bytecode Float Tests (1,000,000 iterations).

and improving upon further performance deficiencies. By the time of the full release, the performance of the AArch64 JVM will only be improved over these baseline results.

4.3 Templating New Architectural Support

Aside from implementing the AArch64 JVM and TRJIT and putting in place a baseline set of benchmarking results, another goal of our work is to put forward a template that allows others to follow the AArch64 model for new architectures. Depending on the similarities in the ISAs (Instruction Set Architectures) between AArch64 and the target, a large amount of the code generator, and particularly the tree evaluators, could be used as a very close template for a new architecture. All of the previously existing Tril tests and those that we implemented can be used as an excellent metric for when a new architecture's tree evaluators are functioning properly.

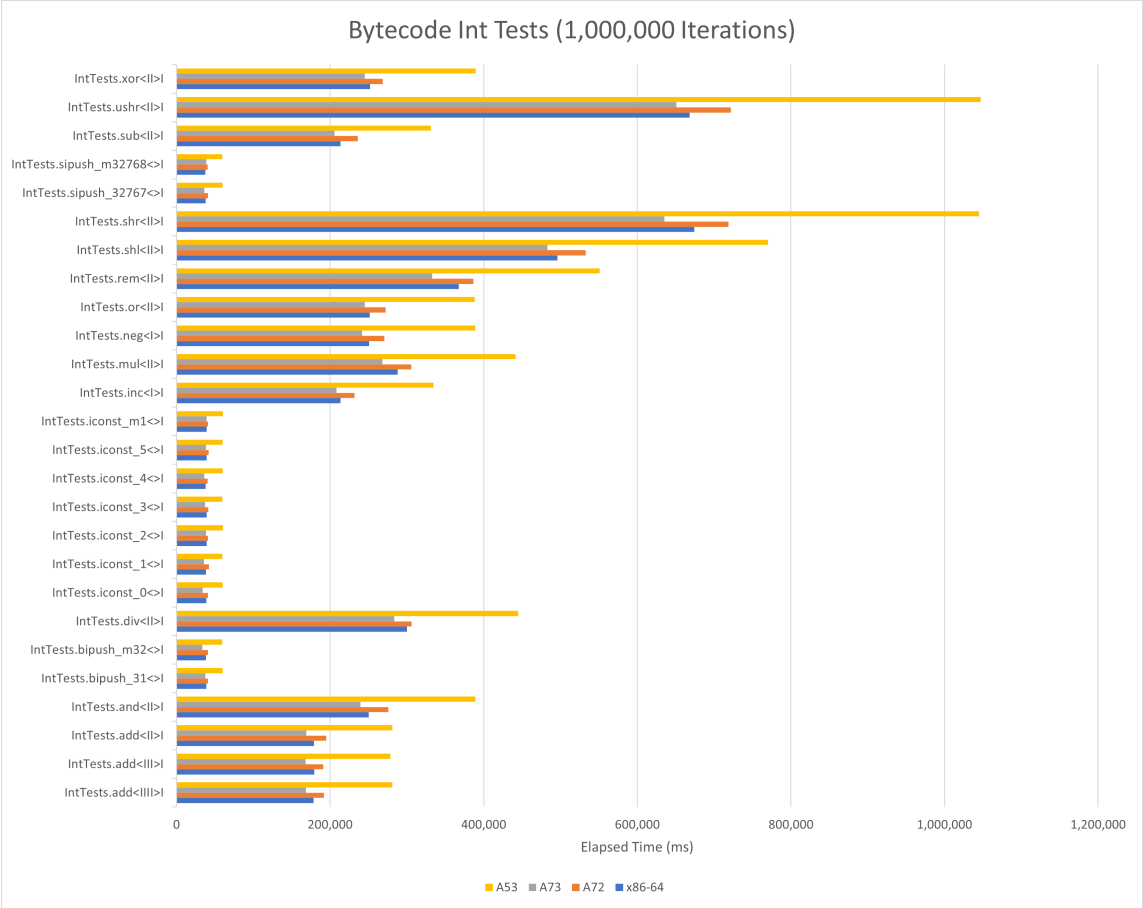

Figure 4.10: Bytecode Int Tests (1,000,000 iterations).

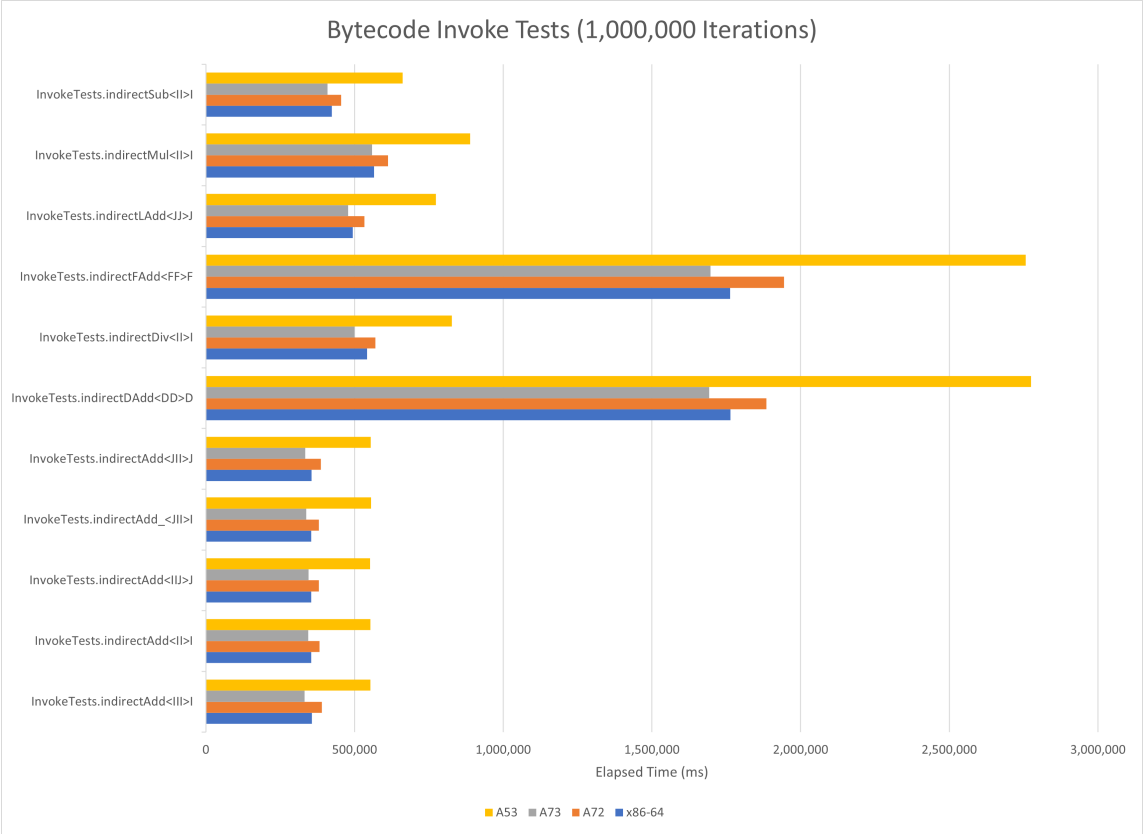

Figure 4.11: Bytecode Invoke Tests (1,000,000 iterations).

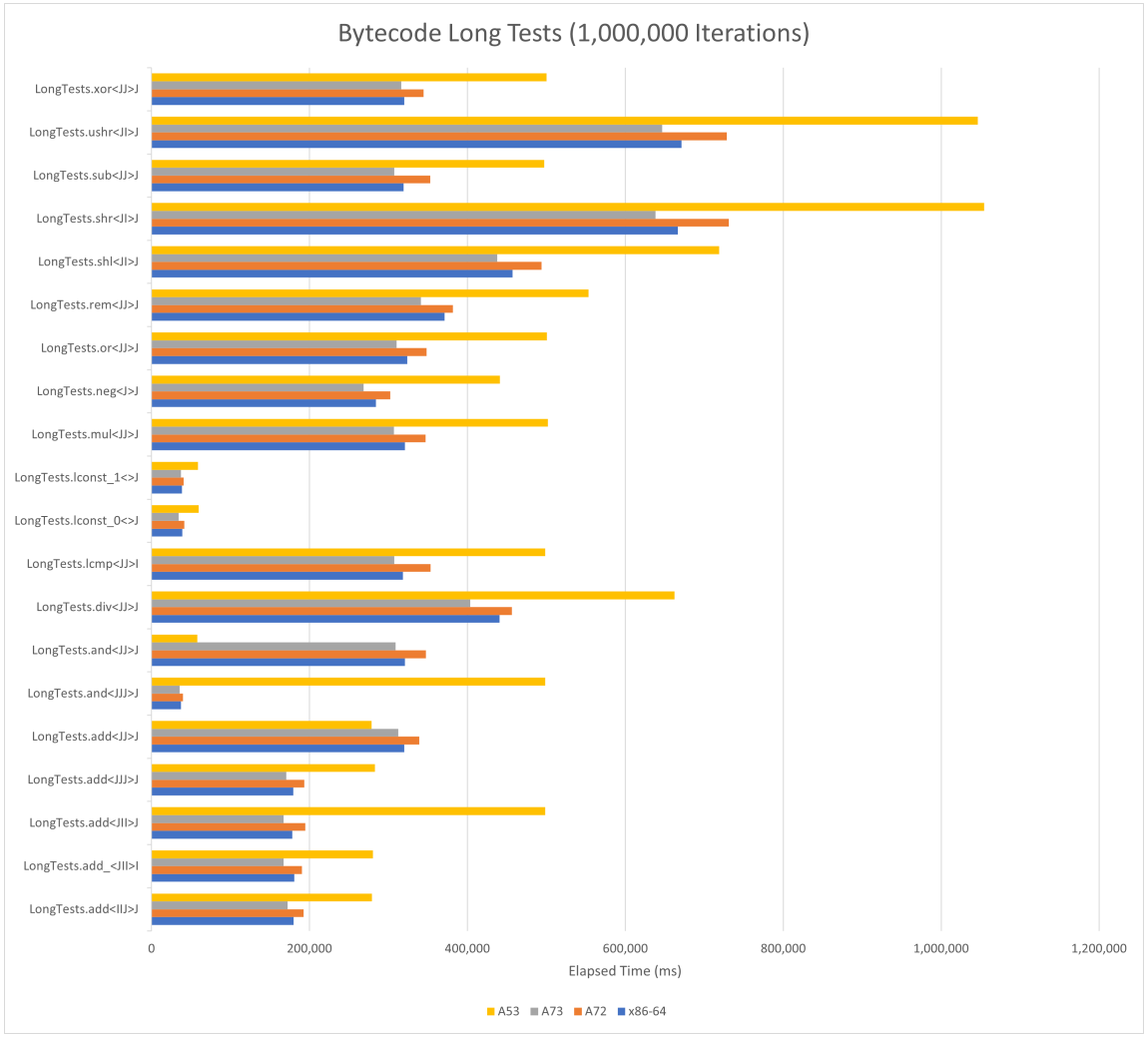

Figure 4.12: Bytecode Long Tests (1,000,000 iterations).

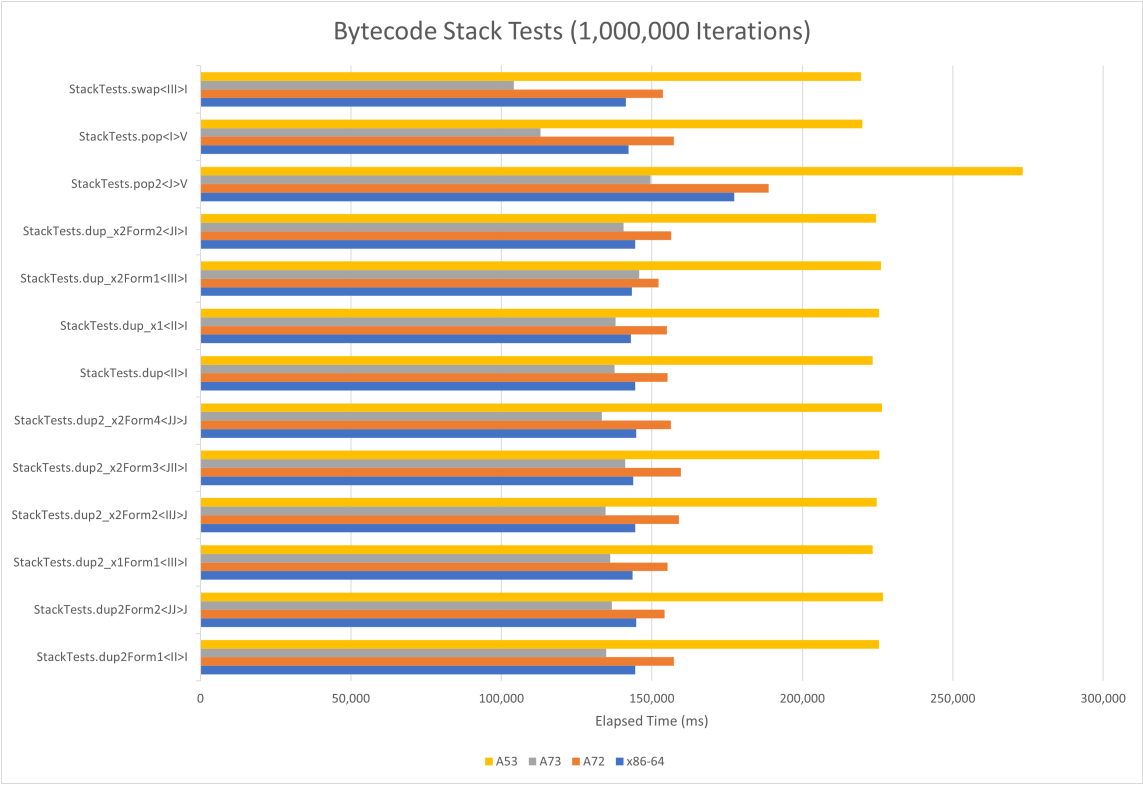

Figure 4.13: Bytecode Stack Tests (1,000,000 iterations).

A new architecture's JVM implementation can use our added Tril tests as a template for any new tests the target architecture requires. Keeping in mind the design of the OpenJ9 JVM and OMR itself requires a number of disparate components to be implemented and connected before any functional and performance testing is possible. This implies there will be a long initial period of implementation where any concrete manner of verification is limited. It is recommended following the implementation of an existing architecture closely to minimize debugging and rework once all the components are connected.

A strong recommendation would be to gain access to a reliable and performant hardware platform as early as possible, in the target architecture, which can support native building, testing and benchmarking of OpenJ9 and OMR. This was a bottleneck for our efforts as we started with no hardware and required cross-compilation from x86-64 and then ARMv8-A emulation with QEMU on x86-64. In these cases, build

times were 3-4× longer than those we have currently attained and test runs were 2-3× longer as well. Finally, we were able to acquire physical hardware to test on, the Rock64 (Table 4.2) however, the device was not powerful enough to support building so we still had to cross-compile or emulate the environment for building and then side-load the binaries and libraries onto the device. The Rock64 did enable us to perform initial testing, which brought down that overhead to 2× of what we now have. It wasn't until we acquired the VIM3 (Table 4.3), and subsequently the Raspberry Pi 4 B (Table 4.4), devices that we significantly brought our build times down and further reduced our testing and benchmarking times as can be seen in Subsections 4.2.1 and 4.2.2.

Making use of our benchmarking and testing framework (Subsection 4.1.2) for ease of generating reproducible and reliable results would be beneficial in speeding up the benchmarking process. This strategy also offers the ability to more easily see incremental improvements between implementation tasks or optimizations. Using the AArch64 baseline results provided here as a metric of a newer and younger JVM implementation for first comparisons would be beneficial. The X, P and Z JVM implementations are more mature and so are more highly optimized with hundreds of developer hours of performance optimizations. This makes results from initial benchmarking runs on a new architecture harder to compare against the X, P and Z JVMs. However, the AArch64 implementation is still in progress so would provide for a more reasonable initial comparison.

Chapter 5

Future Work

All of the benchmarks in Subsection 4.2.1, while extremely useful in providing a direct comparison, are at a higher level of granularity, in that they suggest potential causes of performance bottlenecks. These high-level benchmarks can identify likely causes, and general areas, of performance issues; however, they cannot directly pinpoint causes of performance deficiencies alone. There are two methods we foresee that are promising, aside from that described in Subsection 4.2.2, to perform a deeper dig into the causes of the aforementioned bottlenecks:

- Using Linux Perf tools (Subsection 2.4.2) a deeper-dive into JVM performance and further identification of potential bottlenecks and optimizations would be possible. Perf gathers sampling data of JVM functions and pieces of JIT-ed code where the JVM spends the bulk of its execution time, thus providing information on which areas of the code should be targeted for further improvement.

- Previously in Subsection 3.2.3 we presented the Tril tests. Related to the previous Perf suggestion, another idea is to hook into the tril tests in OMR and insert Linux Perf profiling. This setup would enable us to do a deep dive into the OMR IL (Intermediate Language) opcode evaluators in the JIT. The downside of this method is, as the tril tests are only in OMR and exclude OpenJ9, we

would only see the performance of the language-generic opcodes in this case; not any of the overridden opcodes and evaluators that are *Java*-specific. No doubt this would be an interesting data point, nevertheless, as it would provide more insight into the OMR half of the JIT's performance.

Another item for future work is ARM advanced SIMD (Single Instruction Multiple Data) capabilities. This SIMD support is currently unexploited in the AArch64 JVM and vectorized/scalar support in TRJIT will greatly improve overall performance. Exploiting vectorized loads and stores could vastly improve data movement speeds. By increasing these speeds, we can improve the overall performance of end user's applications.

Furthermore, the AArch64 ISA offers cryptographic support, which can improve the performance of the VM when used in security-sensitive applications. OpenJ9 makes use of software-side cryptographic libraries (e.g., OpenSSL). By leveraging ARMv8-A hardware support directly in TRJIT, this in-software implementation's performance could be enhanced even more.

Exploiting big.LITTLE architectures such as the VIM3 can further improve power usage and the overall performance of the JVM. Having the JVM aware of the heterogeneous cores available will be invaluable as it would allow the background tasks—garbage collector threads, busy-wait, etc.—to be handled by the *little* cores. By satisfying the background tasks with the *little* cores, the JVM can prioritize an end user's application threads to the *big* cores.

Finally, implementing native/inline arraycopy support is currently unsupported in the AArch64 JVM. Inline arraycopy support improves the movement of large *Java* arrays. Providing this capability to the JVM will increase performance in end-user applications with many large arrays.

Chapter 6

Conclusion

Overall, we have shown that the AArch64 platform lends itself as an ideal environment for the OpenJ9 JVM. To reiterate our contributions from Section 1.1, in this thesis, we focused on bringing Eclipse OpenJ9 and Eclipse OMR's Just-in-Time (JIT) compiler to the AArch64 platform. More specifically, we brought an AArch64-specific design and implementation to OpenJ9 and OMR.

We implemented a build (Make and CMake) infrastructure [A1–A3]. We added AArch64 platform support in the Port Library to the JVM [A4]. We implemented the base state of the code generator; including: AArch64 binary encoding [A5–A7], ARMV8-A opcode mnemonics [A5, A7–A9], AArch64 GCMaps [A10], AArch64 processor information [A11], ARM64 tree evaluators [A12–A40], TRJIT code generator helper functions for AArch64 [A41, A42], expanded the set of Tril tests [A43–A47], trampoline support for AArch64 [A48, A49], store/load/full/allocationFence for AArch64 [A50] and Unsafe_compareAndSwapInt_jlObjectJII_Z for AArch64 [A51]. We expanded (Renaissance Suite) and refined the benchmarking and testing framework framework (discussed in Subsection 4.1.2) [19–21]. We provided an evaluation and validation of the AArch64 implementation of the JIT against a more mature architecture (shown in Subsection 4.2.1). This evaluation revealed performance dis-

crepancies and optimization opportunities by comparing the AArch64 implementation to the x86-64 implementation. We designed and implemented the AArch64 JIT from scratch using the other platforms' compilers as templates. This required us to make many decisions on how to architect and implement the AArch64 solution. These decisions were explored in Section 3.1. Our work was an effort to provide a template for new architectural support to allow others to follow our model when targeting new architectures. This was explored in Section 4.3.

To summarize the novelty of this research, we brought AArch64 support to the Eclipse OpenJ9 JVM, Eclipse OMR and the TRJIT. This allows for an enterprise-grade AArch64 JVM for ARMv8-A devices, desktops and servers. A specific example of an excellent use for our contributions in this research is the ever increasingly popular Apple Silicon M1 chip in the MacBook Pro laptop and Mac Mini desktop [23]. We began our research and development of the AArch64 JVM and TRJIT implementation well before the M1 chip was launched in 2020 [23]. We had the foresight to see that this platform and architecture would be extremely important going forward and the market is confirming this.

We have shown that in most cases the A7x cores display comparable, or better performance than their x86-64 counterpart. While the A53 core provides lower performance, these results provide insight into the possible performance provided by entry level CPUs coupled with a highly performant runtime. The A72 core's performance lies in between the A73 and A53 cores. The ARM offerings display great flexibility, and the ability to utilize these resources, improving the infrastructure.

We endeavoured in Subsection 4.2.2 to take a more granular look at the bytecode level to more directly identify which bytecodes should be optimized. While we identified a few outlier cases that will be investigated further with Linux Perf tools, overall this deeper dive did not have the desired effect of directly identifying performance deficiencies observed in Subsection 4.2.1.

We laid out a number of future work items in Chapter 5 to delve deeper into which parts of the TRJIT can be further optimized.

In conclusion, this work lays out a baseline for the comparison, evaluation and analysis of future implementations and optimizations in the JVM on the AArch64 platform.

Bibliography

[1] Orson Scott Card. *Xenocide: Volume Three of the Ender Quintet*. Vol. 3. Tor Books, 2009.

[2] ARM. *ARMv8 Architecture, The whys & wherefores of AArch64*. Last accessed on 2020-04-19. 2013. URL: https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/armv8-architecture-the-whys-wherefores-of-aarch64---64-bit-applications/.

[3] ARM. *ARM*. Last accessed on 2020-04-19. 2020. URL: https://www.arm.com/.

[4] Richard Grisenthwaite. "Armv8 technology preview". In: *IEEE Conference*. 2011.

[5] Matt Curtin. "Write once, run anywhere: Why it matters". In: *Technical Article. http://java.sun.com/features/1998/01/wo* (1998).

[6] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java language specification*. Addison-Wesley Professional, 2000.

[7] Stack Overflow. *Stack Overflow Developer Survey Results 2019*. Last accessed on 2020-04-19. 2019. URL: https://insights.stackoverflow.com/survey/2019/.

[8] Inderjeet Singh, Mark Johnson, and Beth Stearns. *Designing enterprise applications with the J2EE platform*. Addison-Wesley Professional, 2002.

[9] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification, Java SE 7 Edition: Java Virt Mach Spec Java_3*. Addison-Wesley, 2013.

[10] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java virtual machine specification*. Pearson Education, 2014.

[11] Frank Yellin and Tim Lindholm. *The java virtual machine specification*. 1996.

[12] GitHub. *Eclipse OpenJ9*. Last accessed on 2020-04-19. 2020. URL: https://github.com/eclipse/openj9/.

[13] Eclipse. *Eclipse OpenJ9*. Last accessed on 2020-04-19. 2020. URL: https://www.eclipse.org/openj9/.

[14] Michal Cierniak, Marsha Eng, Neal Glew, Brian Lewis, and James Stichnoth. "The Open Runtime Platform: a flexible high-performance managed runtime environment". In: *Concurrency and Computation: Practice and Experience* 17.5-6 (May 2005), pp. 617–637. ISSN: 1532-0634. DOI: 10.1002/cpe.852. URL: http:https://doi.org/10.1002/cpe.852/.

[15] Nicolas Geoffray, Gaël Thomas, Julia Lawall, Gilles Muller, and Bertil Folliot. "VMKit: A Substrate for Managed Runtime Environments". In: *Proceedings of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE '10. Pittsburgh, Pennsylvania, USA: ACM, 2010, pp. 51–62. ISBN: 978-1-60558-910-7. DOI: 10.1145/1735997.1736006. URL: http://doi.acm.org/10.1145/1735997.1736006/.

[16] D. Mulchandani. "Java for embedded systems". In: *IEEE Internet Computing* 2.3 (May 1998), pp. 30–39. ISSN: 1089-7801. DOI: 10.1109/4236.683797.

[17] Doug Simon, Cristina Cifuentes, Dave Cleal, John Daniels, and Derek 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. VEE '06. Ottawa, Ontario, Canada: ACM, 2006, pp. 78–88. ISBN: 1-59593-332-8. DOI: 10.1145/1134760.1134773. URL: http://doi.acm.org/10.1145/1134760.1134773/.

[18] Jim Smith and Ravi Nair. *Virtual machines: versatile platforms for systems and processes*. Elsevier, 2005.

[19] Jean-Philippe Legault and Aaron G. Graham. *aarch64_omr_benchmark*. Last accessed on 2021-02-15. 2020. URL: https://github.com/cas-atlantic/aarch64_omr_benchmark.

[20] Jean-Philippe Legault and Aaron G. Graham. *scimarkC*. Last accessed on 2021-02-15. 2021. URL: https://github.com/cas-atlantic/scimarkC.

[21] Jean-Philippe Legault and Aaron G. Graham. *scimarkJava*. Last accessed on 2021-02-15. 2021. URL: https://github.com/cas-atlantic/scimarkJava.

[22] WikiChip. *ThunderX - Cavium*. Last accessed on 2021-08-25. 2020. URL: https://en.wikichip.org/wiki/cavium/thunderx.

[23] WikiChip. *M1 - Apple*. Last accessed on 2021-08-25. 2020. URL: https://en.wikichip.org/wiki/apple/mx/m1.

[24] Attila Kovari and P Dukan. "KVM & OpenVZ virtualization based IaaS open source cloud virtualization platforms: OpenNode, Proxmox VE". In: *2012 IEEE 10th Jubilee International Symposium on Intelligent Systems and Informatics*. IEEE. 2012, pp. 335–339.

[25] Arief Arfriandi. "Perancangan, Implementasi, Dan Analisis Kinerja Virtualisasi Server Menggunakan Proxmox, Vmware Esx, Dan Openstack". In: *Jurnal Teknologi* 5.2 (2012), pp. 182–191.

[26] Sultan Abdullah Algarni, Mohammad Rafi Ikbal, Roobaea Alroobaea, Ahmed S Ghiduk, and Farrukh Nadeem. "Performance evaluation of Xen, KVM, and proxmox hypervisors". In: *International Journal of Open Source Software and Processes (IJOSSP)* 9.2 (2018), pp. 39–54.

[27] Rik Goldman. *Learning Proxmox VE*. Packt Publishing Ltd, 2016.

[28] Dirk Leinenbach and Thomas Santen. "Verifying the Microsoft Hyper-V hypervisor with VCC". In: *International Symposium on Formal Methods*. Springer. 2009, pp. 806–809.

[29] Hasan Fayyad-Kazan, Luc Perneel, and Martin Timmerman. "Benchmarking the performance of Microsoft Hyper-V server, VMware ESXi and Xen hypervisors". In: *Journal of Emerging Trends in Computing and Information Sciences* 4.12 (2013), pp. 922–933.

[30] Anthony Velte and Toby Velte. *Microsoft virtualization with Hyper-V*. McGraw-Hill, Inc., 2009.

[31] Jon Watson. "Virtualbox: bits and bytes masquerading as machines". In: *Linux Journal* 2008.166 (2008), p. 1.

[32] Peng Li. "Selecting and using virtualization solutions: our experiences with VMware and VirtualBox". In: *Journal of Computing Sciences in Colleges* 25.3 (2010), pp. 11–17.

[33] Dave Mishchenko. *VMware ESXi: Planning, implementation, and security*. Nelson Education, 2010.

[34] John Paul Walters, Andrew J Younge, Dong In Kang, Ke Thia Yao, Mikyung Kang, Stephen P Crago, and Geoffrey C Fox. "GPU passthrough performance: A comparison of KVM, Xen, VMWare ESXi, and LXC for CUDA and OpenCL applications". In: *2014 IEEE 7th international conference on cloud computing*. IEEE. 2014, pp. 636–643.

[35] Parrot Foundation. *Parrot Overview*. Last accessed on 2021-02-16. 2021. URL: http://docs.parrot.org/parrot/latest/html/docs/intro.pod.html.

[36] Dan Sugalski. *Squaks of the Parrot*. Last accessed on 2021-02-16. 2021. URL: https://www.sidhe.org/oldblog/archives/000189.html.

[37] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. "From interpreter to compiler and virtual machine: a functional derivation". In: *BRICS Report Series* 10.14 (2003).

[38] GitHub. *Eclipse OMR*. 2020. URL: https://github.com/eclipse/omr/.

[39] Eclipse. *Eclipse OMR*. Last accessed on 2020-04-19. 2020. URL: https://www.eclipse.org/omr/.

[40] Georgiy Krylov, Gerhard W Dueck, Kenneth B Kent, Daryl Maier, and Irwin D'Souza. "Ahead-of-time compilation in OMR: overview and first steps". In: *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*. 2019, pp. 299–304.

[41] Samer Al Masri, Nazim Uddin Bhuiyan, Sarah Nadi, and Matthew Gaudet. "Software variability through C++ static polymorphism: a case study of challenges and open problems in eclipse OMR". In: *Proceedings of the 27th Annual International Conference on Computer Science and Software Engineering*. IBM Corp. 2017, pp. 285–291.

[42] OMR Eclipse. *Building language runtimes for the cloud*.

[43] Daryl Maier and Xiaoli Liang. "Supercharge a language runtime!" In: *Proceedings of the 27th Annual International Conference on Computer Science and Software Engineering*. IBM Corp. 2017, pp. 314–314.

[44] Daryl Maier and Kenneth Kent. "Advances in open runtime technologies for the cloud". In: *Proceedings of the 27th Annual International Conference on Computer Science and Software Engineering*. IBM Corp. 2017, pp. 303–303.

[45] Daryl Maier, Kenneth B. Kent, Ben Thomas, Vijay Sundaresan, Shelley Lambert, Leonardo Banderali, Younes Manton, Aaron G. Graham, Jean-Philippe Legault, Mark Thom, et al. "2nd workshop on advances in open runtime technology for cloud environments". In: *Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering*. IBM Corp. 2018, pp. 364–368.

[46] Mark Thom, Gerhard W Dueck, Kenneth Kent, and Daryl 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*. IBM Corp. 2018, pp. 275–281.

[47] IBM Developer. *Introducing the IBM Semeru Runtimes*. Last accessed on 2021-09-13. 2021. URL: https://developer.ibm.com/blogs/introducing-the-ibm-semeru-runtimes/.

[48] IBM Developer. *IBM Semeru Runtimes Releases*. Last accessed on 2021-09-13. 2021. URL: https://developer.ibm.com/languages/java/semeru-runtimes/.

[49] AdoptOpenJDK. *AdoptOpenJDK JDK11 OpenJ9 AArch64 Early Release 0.20.0*. Last accessed on 2020-04-19. 2020. URL: https://github.com/AdoptOpenJDK/openjdk11-binaries/releases/tag/jdk-11.0.7%2B10_openj9-0.20.0/.

[50] OpenJ9. *OpenJ9 AArch64 Early Release 0.20.0*. Last accessed on 2020-04-24. 2020. URL: https://blog.openj9.org/2020/04/24/aarch64-early-access-in-openj9-0-20-0/.

[51] Yang Wang, Kenneth B Kent, and Graeme Johnson. "Improving J9 virtual machine with LTTng for efficient and effective tracing". In: *Software: Practice and Experience* 45.7 (2015), pp. 973–987.

[52] Dev Bhattacharya, Kenneth B Kent, Eric Aubanel, Daniel Heidinga, Peter Shipton, and Aleksandar Micic. “Improving the performance of JVM startup using the shared class cache”. In: *2017 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*. IEEE. 2017, pp. 1–6.

[53] Toshio Suganuma, Takeshi Ogasawara, Mikio Takeuchi, Toshiaki Yasue, Motohiro Kawahito, Kazuaki Ishizaki, Hideaki Komatsu, and Toshio Nakatani. “Overview of the IBM Java just-in-time compiler”. In: *IBM systems Journal* 39.1 (2000), pp. 175–193.

[54] Toshio Suganuma, Toshiaki Yasue, Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani. “A dynamic optimization framework for a Java just-in-time compiler”. In: *ACM SIGPLAN Notices* 36.11 (2001), pp. 180–195.

[55] Matthew Gaudet and Mark Stoodley. “Rebuilding an airliner in flight: A retrospective on refactoring IBM Testarossa production compiler for Eclipse OMR”. In: *Proceedings of the 8th International Workshop on Virtual Machines and Intermediate Languages*. 2016, pp. 24–27.

[56] Eric Coffin, Scott Young, Kenneth B Kent, and Marius 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*. 2019, pp. 293–298.

[57] Lukasz Wycislik and Lukasz Ogorek. “Issues on Performance of Reactive Programming in the Java Ecosystem with Persistent Data Sources”. In: *International Conference on Man–Machine Interactions*. Springer. 2019, pp. 249–258.

[58] Eric Coffin, Scott Young, Harpreet Kaur, Julie Brown, Marius Pirvu, and Kenneth B Kent. “MicroJIT: a case for templated just-in-time compilation in

constrained environments". In: *Proceedings of the 30th Annual International Conference on Computer Science and Software Engineering*. 2020, pp. 179–188.

[59] Alex Iliasov. "Templates-based portable just-in-time compiler". In: *ACM SIGPLAN Notices* 38.8 (2003), pp. 37–43.

[60] Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khang, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, et al. "The DaCapo benchmarks: Java benchmarking development and analysis". In: *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*. 2006, pp. 169–190.

[61] Philipp Lengauer, Verena Bitto, Hanspeter Mössenböck, and Markus Weninger. "A comprehensive java benchmark study on memory and garbage collection behavior of dacapo, dacapo scala, and specjvm2008". In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*. 2017, pp. 3–14.

[62] DaCapo. *DaCapo Benchmark Descriptions*. Last accessed on 2021-02-22. 2021. URL: `http://dacapobench.sourceforge.net/benchmarks.html`.

[63] Roldan Pozo and Bruce Miller. *scimark2*. Last accessed on 2021-02-15. 2021. URL: `https://math.nist.gov/scimark2`.

[64] Roldan Pozo and Bruce Miller. *scimark2 About*. Last accessed on 2021-02-15. 2021. URL: `https://math.nist.gov/scimark2/about.html`.

[65] Roldan Pozo. "SciMark 2.0". In: *http://math. nist. gov/scimark2/* (2000).

[66] SPEC. *SPECjvm2008*. Last accessed on 2020-06-12. 2020. URL: `https://www.spec.org/jvm2008`.

[67] Kumar Shiv, Kingsum Chow, Yanping Wang, and Dmitry Petrochenko. "The SPECjvm2008 performance characterization". In: *SPEC Benchmark Workshop*. Springer. 2009, pp. 17–35.

[68] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, et al. "Renaissance: benchmarking suite for parallel applications on the JVM". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2019, pp. 31–47.

[69] Renaissance Suite. *Renaissance Suite Website*. Last accessed on 2020-10-23. 2020. URL: `https://renaissance.dev/`.

[70] Davood Mazinanian, Ameya Ketkar, Nikolaos Tsantalis, and Danny Dig. "Understanding the use of lambda expressions in Java". In: *Proceedings of the ACM on Programming Languages* 1.OOPSLA (2017), pp. 1–31.

[71] Adam Welc, Suresh Jagannathan, and Antony Hosking. "Safe futures for Java". In: *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 2005, pp. 439–453.

[72] Twitter Finagle. *Twitter Finagle Website*. Last accessed on 2020-10-23. 2020. URL: `https://github.com/twitter/finagle/`.

[73] Abdullah Talha Kabakus and Resul Kara. "A performance evaluation of in-memory databases". In: *Journal of King Saud University-Computer and Information Sciences* 29.4 (2017), pp. 520–525.

[74] Raffi Khatchadourian, Yiming Tang, Mehdi Bagherzadeh, and Syed Ahmed. "Safe automated refactoring for intelligent parallelization of Java 8 streams". In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE. 2019, pp. 619–630.

[75] Justin J Miller. “Graph database applications and concepts with Neo4j”. In: *Proceedings of the Southern Association for Information Systems Conference, Atlanta, GA, USA*. Vol. 2324. 2013.

[76] Panagiotis Patros, Eric Aubanel, David Bremner, and Michael Dawson. “A Java util concurrent park contention tool”. In: *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores*. 2015, pp. 106–111.

[77] Apache Spark. *Apache Spark Website*. Last accessed on 2020-10-23. 2020. URL: `https://spark.apache.org/`.

[78] Arnaldo Carvalho De Melo. “The new linux’perf’tools”. In: *Slides from Linux Kongress*. Vol. 18. 2010, pp. 1–42.

[79] perf-tools. *Perf Tools*. Last accessed on 2020-06-12. 2020. URL: `https://github.com/brendangregg/perf-tools`.

[80] Vincent M Weaver. “Linux perf_event features and overhead”. In: *The 2nd International Workshop on Performance Analysis of Workload Optimized Systems, FastPath*. Vol. 13. 2013.

[81] Swapnil Gaikwad, Andy Nisbet, and Mikel Luján. “Performance analysis for languages hosted on the truffle framework”. In: *Proceedings of the 15th International Conference on Managed Languages & Runtimes*. 2018, pp. 1–12.

[82] Sasha Goldshtein. “Profiling JVM Applications in Production”. In: *SREcon18 Europe/Middle East/Africa (SREcon18 Europe)* (2018).

[83] Eclipse OMR. *Eclipse OMR X TRJIT Compiler Source Tree*. Last accessed on 2021-03-03. 2021. URL: `https://github.com/eclipse/omr/tree/master/compiler/x`.

[84] Kevin Bradley. “Defining digital sustainability”. In: *Library Trends* 56.1 (2007), pp. 148–163.

[85] Eclipse OMR. *Eclipse OMR P TRJIT Compiler Source Tree*. Last accessed on 2021-03-03. 2021. URL: https://github.com/eclipse/omr/tree/master/compiler/p.

[86] Eclipse OMR. *Eclipse OMR Z TRJIT Compiler Source Tree*. Last accessed on 2021-03-03. 2021. URL: https://github.com/eclipse/omr/tree/master/compiler/z.

[87] Eclipse OMR. *Eclipse OMR ARM TRJIT Compiler Source Tree*. Last accessed on 2021-03-03. 2021. URL: https://github.com/eclipse/omr/tree/master/compiler/arm.

[88] Eclipse OMR. *Eclipse OMR AArch64 TRJIT Compiler Source Tree*. Last accessed on 2021-03-03. 2021. URL: https://github.com/eclipse/omr/tree/master/compiler/aarch64.

[89] Eclipse OpenJ9. *Eclipse OpenJ9 Contribution Guidelines*. Last accessed on 2021-03-03. 2021. URL: https://github.com/eclipse/openj9/blob/master/CONTRIBUTING.md.

[90] Eclipse OMR. *Eclipse OMR Coding Standard*. Last accessed on 2021-03-03. 2021. URL: https://github.com/eclipse/omr/blob/master/doc/CodingStandard.md.

[91] Eclipse OMR. *OMR/Testarossa Intermediate Language: An Intro to Trees*. Last accessed on 2021-03-03. 2021. URL: https://github.com/eclipse/omr/blob/master/doc/compiler/il/IntroToTrees.md.

[92] Eclipse OMR. *Eclipse OMR Binary Evaluators*. Last accessed on 2021-03-28. 2021. URL: https://github.com/eclipse/omr/blob/master/compiler/aarch64/codegen/BinaryEvaluator.cpp.

[93] Eclipse OMR. *Eclipse OMR Control Flow Evaluators*. Last accessed on 2021-03-28. 2021. URL: https://github.com/eclipse/omr/blob/master/compiler/aarch64/codegen/ControlFlowEvaluator.cpp.

[94] Eclipse OMR. *Eclipse OMR Floating-Point Evaluators*. Last accessed on 2021-03-28. 2021. URL: https://github.com/eclipse/omr/blob/master/compiler/aarch64/codegen/FPTreeEvaluator.cpp.

[95] Eclipse OMR. *Eclipse OMR Unary Evaluators*. Last accessed on 2021-03-28. 2021. URL: https://github.com/eclipse/omr/blob/master/compiler/aarch64/codegen/UnaryEvaluator.cpp.

[96] Eclipse OpenJ9. *Eclipse OpenJ9 Evaluators*. Last accessed on 2021-03-28. 2021. URL: https://github.com/eclipse-openj9/openj9/blob/master/runtime/compiler/aarch64/codegen/J9TreeEvaluator.cpp.

[97] Eclipse OMR. *OMR Tril Tests*. Last accessed on 2021-03-25. 2021. URL: https://github.com/eclipse/omr/blob/master/fvtest/tril/README.md.

[98] Eclipse OMR. *GoogleTest*. Last accessed on 2021-03-25. 2021. URL: https://github.com/google/googletest/tree/master/googletest.

[99] IBM. *Google Testing Framework*. Last accessed on 2021-03-25. 2021. URL: https://developer.ibm.com/technologies/systems/articles/au-googletestingframework/.

[100] Eclipse OMR. *OMR Tril Language Reference*. Last accessed on 2021-04-24. 2021. URL: https://github.com/eclipse/omr/blob/master/doc/compiler/tril/TrilLanguageReference.md.

[101] Eclipse OMR. *OMR fvtests*. Last accessed on 2021-03-25. 2021. URL: https://github.com/eclipse/omr/blob/master/fvtest/compilertest/README.md.

[102] ARM. *ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile*. Last accessed on 2021-03-28. 2021. URL: https://documentation-service.arm.com/static/60119835773bb020e3de6fee?token=.

[103] Eclipse OMR. *OMR AArch64 OpCode Mnemonics*. Last accessed on 2021-03-25. 2021. URL: https://github.com/eclipse/omr/blob/master/compiler/aarch64/codegen/OMRInstOpCodeEnum.hpp.

[104] Eclipse. *AArch64: Merged OMR Pull Requests*. Last accessed on 2021-05-29. 2021. URL: https://github.com/eclipse/omr/pulls?q=is%3Apr+is%3Aclosed+AArch64.

[105] Eclipse. *AArch64: Merged OpenJ9 Pull Requests*. Last accessed on 2021-05-29. 2021. URL: https://github.com/eclipse-openj9/openj9/pulls?q=is%3Apr+is%3Aclosed+AArch64.

[106] Eclipse. *AdoptOpenJDK. 2020. Adoptopenjdk jdk11 openj9 aarch64 early release 0.23 for AArch64*. Last accessed on 2021-05-29. 2021. URL: https://github.com/AdoptOpenJDK/openjdk11-binaries/releases/download/jdk-11.0.9%2B11_openj9-0.23.0/OpenJDK11U-jdk_aarch64_linux_openj9_11.0.9_11_openj9-0.23.0.tar.gz.

[107] Eclipse. *AdoptOpenJDK. 2020. Adoptopenjdk jdk11 openj9 aarch64 early release 0.23 for x86-64*. Last accessed on 2021-05-29. 2021. URL: https://github.com/AdoptOpenJDK/openjdk11-binaries/releases/download/jdk-11.0.9%2B11_openj9-0.23.0/OpenJDK11U-jdk_x64_linux_openj9_11.0.9_11_openj9-0.23.0.tar.gz.

[108] Myungsun Kim, Kibeom Kim, James R Geraci, and Seongsoo Hong. "Utilization-aware load balancing for the energy efficient operation of the big. LITTLE processor". In: *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2014, pp. 1–4.

[109] Edson Luiz Padoin, Laércio Lima Pilla, Márcio Castro, Francieli Z Boito, Philippe Olivier Alexandre Navaux, and Jean-François Méhaut. "Performance/energy trade-off in scientific computing: the case of ARM big. LITTLE and Intel Sandy Bridge". In: *IET Computers & Digital Techniques* 9.1 (2014), pp. 27–35.

[110] Marcus Hähnel and Hermann Härtig. "Heterogeneity by the Numbers: A Study of the ODROID XU+ E big. LITTLE Platform". In: *6th Workshop on Power-Aware Computing and Systems (HotPower 14)*. 2014.

[111] Maria Patrou, Jean-Philippe Legault, Aaron G Graham, and Kenneth B Kent. "Improving digital circuit simulation with batch-parallel logic evaluation". In: *2019 22nd Euromicro Conference on Digital System Design (DSD)*. IEEE. 2019, pp. 144–151.

[112] Kevin E Murray, Oleg Petelin, Sheng Zhong, Jia Min Wang, Mohamed Eldafrawy, Jean-Philippe Legault, Eugene Sha, Aaron G Graham, Jean Wu, Matthew JP Walker, et al. "Vtr 8: High-performance cad and customizable fpga architecture modelling". In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 13.2 (2020), pp. 1–55.

[113] Federico Sogaro, Eric Aubanel, Kenneth B Kent, Marius Pirvu, Vijay Sundaresan, and Peter 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, ON, Canada: IBM Corp., 2017, pp. 140–150.

[114] A. G. Graham, J.-P. Legault, Md. A. Noor, S. Ponangi, S. Moffatt, Md. A. Haque, H. Soontiens, J. Brown, M. Flawn, K. B. Kent, G. W. Dueck, S. MacKay, D. Maier, K. Konno, and T. Renaud. "IBM CAS (Centre for Advanced Studies) 2020 Project of the Year Award: Project 1035 - OMR in

Resource Constrained Environments (Embedded/IoT/Mobile OMR)". In: *Proceedings of the 30th Annual International Conference on Computer Science and Software Engineering (CASCON x EVOKE 2020)*. On: https://www-01.ibm.com/ibm/cas/canada/awards?award=PotY2020. IBM Corp. IBM Canada Lab, Markham, Ontario, Canada, Nov. 9, 2020.

[115] A. G. Graham, J.-P. Legault, H. Soontiens, J. Brown, S. MacKay, G. W. Dueck, K. B. Kent, K. Konno, and D. Maier. "Evaluating the Performance of the Eclipse OpenJ9 JVM JIT on AArch64". In: 31st Annual International Conference on Computer Science and Software Engineering (CASCON x EVOKE 2021). IBM Corp. Markham, Ontario, Canada, Nov. 22, 2021.

[116] Kevin E. Murray, Oleg Petelin, Sheng Zhong, Jia Min Wang, Mohamed Eldafrawy, Jean-Philippe Legault, Eugene Sha, Aaron G. Graham, Jean Wu, Matthew J. P. Walker, Hanqing Zeng, Panagiotis Patros, Jason Luu, Kenneth B. Kent, and Vaughn Betz. "VTR 8: High-Performance CAD and Customizable FPGA Architecture Modelling". In: New York, NY, USA: Association for Computing Machinery, June 1, 2020.

[117] M. Patrou, J. Legault, A. G. Graham, and K. B. Kent. "Improving Digital Circuit Simulation with Batch-Parallel Logic Evaluation". In: *2019 22nd Euromicro Conference on Digital System Design (DSD)*. Chalkidiki, Greece, Greece: IEEE Xplore, Aug. 1, 2019, pp. 144–151. DOI: 10.1109/DSD.2019.00031.

[118] J.-P. Legault, A. G. Graham, K. B. Kent, D. Maier, and K. Konno. "Evaluating the OpenJ9 JIT on AArch64". In: 4th Workshop on Advances in Open Runtimes and Cloud Performance Technologies (AORCPT 2020). IBM Corp. Markham, Ontario, Canada, Nov. 10, 2021.

[119] J.-P. Legault, A. G. Graham, H. Soontiens, M. Flawn, Md.M. Rahman, K. B. Kent, D. Maier, and K. Konno. “Development and Evaluation of the Eclipse OMR Library on AArch64 using the Eclipse OpenJ9 Runtime”. In: 3rd Workshop on Advances in Open Runtime Technology for Cloud Computing (AORTCC 2019). IBM Corp. Markham, Ontario, Canada, Nov. 5, 2019.

[120] J.-P. Legault, A. G. Graham, Dr. K. B. Kent, D. Maier, and K. Konno. “AArch64 Support for the OMR Language Runtime Toolkit”. In: TURBO’18 Building Language Runtimes with Eclipse OMR Workshop, SPLASH 2018. IBM Corp. Boston, United States, Nov. 4, 2018.

[121] J.-P. Legault, A. G. Graham, Dr. K. B. Kent, D. Maier, and K. Konno. “AArch64 Support for the OMR Language Runtime Toolkit”. In: 2nd Annual Workshop on Advances in Open Runtime Technology for Cloud Environments, CASCON 2018. IBM Corp. Markham, Ontario, Canada, Oct. 29, 2018.

[122] S.S. Ponangi, A.G. Graham, Md.A. Noor, J.-P. Legault, J. Brown, M. Flawn, K.B. Kent, G.W. Dueck, K. Konno, and D. Maier. “Eclipse OpenJ9 and Eclipse OMR AArch64 Just-In-Time Compiler Implementation, Baseline and Optimization”. In: CASTLE 2021. IEEE Xplore. Markham, Ontario, Canada, May 10, 2021.

[123] A.G. Graham, J.-P. Legault, K.B. Kent, K. Konno, and D. Maier. “Evaluating the Performance of the Eclipse OpenJ9 JIT Compiler on the AArch64 Platform”. In: 30th Annual International Conference on Computer Science and Software Engineering (CASCON x EVOKE 2020). IBM Corp. Markham, Ontario, Canada, Nov. 9, 2020.

[124] Md.A. Noor, J.-P. Legault, Md.A. Haque, S.S. Ponangi, A. G. Graham, K. B. Kent, G. W. Dueck, and D. Maier. “AArch64 Support for the Eclipse OMR

Language Runtime Toolkit & Eclipse OpenJ9". In: CASTLE 2020. IEEE Xplore. Markham, Ontario, Canada, May 11, 2020.

[125] A. G. Graham, J.-P. Legault, H. Soontiens, M. Flawn, Md.M. Rahman, K. B. Kent, D. Maier, and K. Konno. "Building AArch64 Support for Eclipse OpenJ9 and Eclipse OMR". In: 29th Annual International Conference on Computer Science and Software Engineering (CASCON x EVOKE 2019). IBM Corp. Markham, Ontario, Canada, Nov. 5, 2019.

[126] J.-P. Legault, A. G. Graham, Kenneth B. Kent, and D. Maier. "XDocker: A Cross-Platform Tool for Continuous Development". In: 29th Annual International Conference on Computer Science and Software Engineering (CASCON x EVOKE 2019). IBM Corp. Markham, Ontario, Canada, Nov. 5, 2019.

[127] A. G. Graham, J.-P. Legault, M. Patrou, and K. B. Kent. "Improved Synthesis and Simulation for FPGAs: ODIN II for VTR 8.0 and Beyond". In: 16th Annual Research Exposition of the UNB Faculty of Computer Science. University of New Brunswick. Fredericton, New Brunswick, Canada, Apr. 12, 2019.

[128] J.-P. Legault, A. G. Graham, Dr. K. B. Kent, D. Maier, and K. Konno. "AArch64 Support for the OMR Language Runtime Toolkit". In: CASCON 2018. IBM Corp. Markham, Ontario, Canada, Oct. 29, 2018.

[129] J.-P. Legault, A. G. Graham, Dr. K. B. Kent, D. Maier, and J. Kingdon. "Language Runtimes on Embedded Architectures". In: CASTLE 2018. IBM Canada Lab. Markham, Ontario, Canada, May 8, 2018.

[130] J.-P. Legault, A. G. Graham, Dr. K. B. Kent, D. Maier, and J. Kingdon. "Language Runtimes on Embedded Architectures". In: 15th Annual Research Exposition of the UNB Faculty of Computer Science. University of New Brunswick. Fredericton, New Brunswick, Canada, Apr. 11, 2018.

Appendix A

Public Code Contributions

[A1] Aaron G. Graham. *AArch64: Initial Configure and Make/Build Systems*. Created on: May 4, 2018 - Merged on: May 12, 2018. 2018. URL: https://github.com/eclipse/omr/pull/2526.

[A2] Aaron G. Graham. *Update config.guess and config.sub to the Latest Versions*. Created on: May 11, 2018 - Merged on: May 11, 2018. 2018. URL: https://github.com/eclipse/omr/pull/2540.

[A3] Aaron G. Graham. *Build AArch64 Compiler Files*. Created on: Aug 15, 2018 - Merged on: Aug 16, 2018. 2018. URL: https://github.com/eclipse/omr/pull/2863.

[A4] Aaron G. Graham. *AArch64: Initial Port Library Changes*. Created on: May 2, 2018 - Merged on: May 12, 2018. 2018. URL: https://github.com/eclipse/omr/pull/2520.

[A5] Jean-Philippe Legault. *AArch64: Binary Encoding*. Created on: May 29, 2018 - Merged on: Jul 11, 2018. 2018. URL: https://github.com/eclipse/omr/pull/2600.

[A6] Md. Mahbubur Rahman. *AArch64: Binary Encoding Fix and Addition of 'B' and 'BL' Instructions*. Created on: Oct 18, 2018 - Merged on: Oct 31, 2018. 2018. URL: https://github.com/eclipse/omr/pull/3087.

[A7] Hillary Soontiens. *Add DSB instruction to AArch64 opcode tables*. Created on: Jul 8, 2019 - Merged on: Jul 19, 2019. 2019. URL: https://github.com/eclipse/omr/pull/4112.

[A8] Jean-Philippe Legault. *AArch64: This Fixes duplicate opcode in ARM64Ops*. Created on: Jul 20, 2018 - Merged on: Aug 7, 2018. 2018. URL: https://github.com/eclipse/omr/pull/2779.

[A9] Aaron G. Graham. *AArch64 TreeEvaluator*. Created on: Jul 26, 2018 - Merged on: Aug 20, 2018. 2018. URL: https://github.com/eclipse/omr/pull/2801.

[A10] Aaron G. Graham. *Implement AArch64 GCMaps*. Created on: May 9, 2019 - Merged on: Jun 11, 2019. 2019. URL: https://github.com/eclipse/omr/pull/3828.

[A11] Aaron G. Graham. *Implement AArch64 and ARMv8 Processor Information*. Created on: Jul 8, 2019 - Merged on: Jul 10, 2019. 2019. URL: https://github.com/eclipse/omr/pull/4111.

[A12] Aaron G. Graham. *Enabled Missed Tree Evaluators in AArch64 Compiler*. Created on: May 9, 2019 - Merged on: May 16, 2019. 2019. URL: https://github.com/eclipse/omr/pull/3829.

[A13] Aaron G. Graham. *Implement evaluators for checkcast and checkcastAndNULLCHK*. Created on: Jun 18, 2019 - Merged on: Jul 5, 2019. 2019. URL: https://github.com/eclipse/openj9/pull/6176.

[A14] Aaron G. Graham. *Fix DIVCHK Evaluator Function Declaration*. Created on: Jul 5, 2019 - Merged on: Jul 5, 2019. 2019. URL: https://github.com/eclipse/openj9/pull/6353.

[A15] Md. Mahbubur Rahman. *lneg evaluator triltest*. Created on: Dec 11, 2018 - Merged on: Jan 23, 2019. 2019. URL: https://github.com/eclipse/omr/pull/3321.

[A16] Md. Mahbubur Rahman. *AArch64: Implement lneg Evaluator*. Created on: Jan 14, 2019 - Merged on: Jan 21, 2019. 2019. URL: https://github.com/eclipse/omr/pull/3453.

[A17] Md. Mahbubur Rahman. *Fix lnegEvaluator to correctly generate 64-bit negate instruction*. Created on: Feb 25, 2019 - Merged on: Feb 26, 2019. 2019. URL: https://github.com/eclipse/omr/pull/3609.

[A18] Md. Mahbubur Rahman. *AArch64: Implementation of ArrayLengthEvaluator*. Created on: Jun 10, 2019 - Merged on: Jul 9, 2019. 2019. URL: https://github.com/eclipse/openj9/pull/6077.

[A19] Md. Mahbubur Rahman. *AArch64: Implementation of BNDCHK Evaluator*. Created on: Jul 11, 2019 - Merged on: Jul 23, 2019. 2019. URL: https://github.com/eclipse/openj9/pull/6427.

[A20] Hillary Soontiens. *Implemented lmul evaluator*. Created on: Jan 16, 2019 - Merged on: Apr 26, 2019. 2019. URL: https://github.com/eclipse/omr/pull/3467.

[A21] Hillary Soontiens. *Implemented binary evaluator helper for AArch64*. Created on: Jan 28, 2019 - Merged on: Feb 26, 2019. 2019. URL: https://github.com/eclipse/omr/pull/3517.

[A22] Hillary Soontiens. *Optimized imul evaluator to maximize register re-usage*. Created on: May 7, 2019 - Merged on: May 17, 2019. 2019. URL: https://github.com/eclipse/omr/pull/3813.

[A23] Hillary Soontiens. *Implement NULLCHKEvaluator for AArch64*. Created on: Aug 1, 2019 - Merged on: Aug 18, 2019. 2019. URL: https://github.com/eclipse/omr/pull/4187.

[A24] Hillary Soontiens. *Implement DIVCHKEvaluator for AArch64*. Created on: Jun 7, 2019 - Merged on: Jul 4, 2019. 2019. URL: https://github.com/eclipse/openj9/pull/6055.

[A25] Michael Flawn. *Removed monexitfenceEvaluator from ARM and AArch64*. Created on: Jul 12, 2019 - Merged on: Jul 17, 2019. 2019. URL: https://github.com/eclipse/omr/pull/4130.

[A26] Michael Flawn. *Common monexitfenceEvaluator, removed arch specific versions*. Created on: Jun 26, 2019 - Merged on: Jul 12, 2019. 2019. URL: https://github.com/eclipse/openj9/pull/6280.

[A27] Michael Flawn. *AArch64: Implementation of monentEvaluator*. Created on: Jul 4, 2019 - Merged on: Jul 8, 2019. 2019. URL: https://github.com/eclipse/openj9/pull/6345.

[A28] Michael Flawn. *AArch64: Implementation of instanceofEvaluator*. Created on: Jul 5, 2019 - Merged on: Jul 11, 2019. 2019. URL: https://github.com/eclipse/openj9/pull/6354.

[A29] Michael Flawn. *AArch64: Implementation of monexitEvaluator*. Created on: Jul 5, 2019 - Merged on: Jul 5, 2019. 2019. URL: https://github.com/eclipse/openj9/pull/6355.

[A30] Michael Flawn. *AArch64: Implementation of newObjectEvaluator*. Created on: Jul 5, 2019 - Merged on: Jul 5, 2019. 2019. URL: https://github.com/eclipse/openj9/pull/6356.

[A31] Michael Flawn. *AArch64: Implementation of newArrayEvaluator*. Created on: Jul 5, 2019 - Merged on: Jul 8, 2019. 2019. URL: https://github.com/eclipse/openj9/pull/6357.

[A32] Michael Flawn. *AArch64: Implementation of anewArrayEvaluator*. Created on: Jul 5, 2019 - Merged on: Jul 8, 2019. 2019. URL: https://github.com/eclipse/openj9/pull/6358.

[A33] Michael Flawn. *AArch64: Implementation of multianewArrayEvaluator*. Created on: Jul 5, 2019 - Merged on: Jul 11, 2019. 2019. URL: https://github.com/eclipse/openj9/pull/6359.

[A34] Michael Flawn. *AArch64: Fixed missing type on newArrayEvaluator*. Created on: Jul 8, 2019 - Merged on: Jul 8, 2019. 2019. URL: https://github.com/eclipse/openj9/pull/6372.

[A35] Michael Flawn. *AArch64: Fixed Syntax Error in newObjectEvaluator*. Created on: Jul 8, 2019 - Merged on: Jul 8, 2019. 2019. URL: https://github.com/eclipse/openj9/pull/6383.

[A36] Michael Flawn. *AArch64: Implementation of awrtbarEvaluator in AArch64*. Created on: Jul 16, 2019 - Merged on: Aug 21, 2019. 2019. URL: https://github.com/eclipse/openj9/pull/6495.

[A37] Michael Flawn. *AArch64: Implementation of awrtbariEvaluator in AArch64*. Created on: Jul 23, 2019 - Merged on: Aug 22, 2019. 2019. URL: https://github.com/eclipse/openj9/pull/6559.

[A38] Michael Flawn. *AArch64: Implementation of MethodEnterHook/MethodExitHook*. Created on: Aug 14, 2019 - Merged on: Aug 22, 2019. 2019. URL: https://github.com/eclipse/openj9/pull/6739.

[A39] Md. Ariful Haque. *Move NullChkEvaluators to OpenJ9 for ARM, Power, Z and Aarch64…* Created on: Nov 20, 2019 - Merged on: Nov 28, 2019. 2019. URL: https://github.com/eclipse/omr/pull/4581.

[A40] Md. Ariful Haque. *Move NullChkEvaluators from OMR for ARM, Power, Z and Aarch64 c…* Created on: Nov 20, 2019 - Merged on: Nov 28, 2019. 2019. URL: https://github.com/eclipse/openj9/pull/7810.

[A41] Md. Mahbubur Rahman. *Implementation of ARM64CondTrg1Src2Instruction instruction.* Created on: Jun 27, 2019 - Merged on: Jul 5, 2019. 2019. URL: https://github.com/eclipse/omr/pull/4082.

[A42] Md. Ariful Haque. *Use CBZ/CBNZ instructions in ificmpHelper function.* Created on: Jun 19, 2020 - Merged on: Jun 27, 2020. 2020. URL: https://github.com/eclipse/omr/pull/5329.

[A43] Aaron G. Graham. *Tril Tests for ladd / lsub.* Created on: Nov 19, 2018 - Merged on: Nov 26, 2018. 2018. URL: https://github.com/eclipse/omr/pull/3214.

[A44] Aaron G. Graham. *Tril Tests for iand and ixor.* Created on: Dec 13, 2018 - Merged on: Dec 14, 2018. 2018. URL: https://github.com/eclipse/omr/pull/3338.

[A45] Aaron G. Graham. *Add UsingLoadParam, Unary and Binary, LogicalTest Tril Tests.* Created on: Jan 15, 2019 - Merged on: Jan 15, 2019. 2019. URL: https://github.com/eclipse/omr/pull/3459.

[A46] Hillary Soontiens. *Implemented tril tests for lmul, lrem, and ldiv.* Created on: Jan 15, 2019 - Merged on: Jan 15, 2019. 2019. URL: https://github.com/eclipse/omr/pull/3457.

[A47] Hillary Soontiens. *Implemented land/lor/lxor tril tests*. Created on: Jan 15, 2019 - Merged on: Jan 17, 2019. 2019. URL: https://github.com/eclipse/omr/pull/3461.

[A48] Hillary Soontiens. *Implement trampoline support for AArch64*. Created on: May 23, 2019 - Merged on: Jun 5, 2019. 2019. URL: https://github.com/eclipse/openj9/pull/5859.

[A49] Hillary Soontiens. *Fix Trampoline.cpp AArch64 build error*. Created on: Jun 6, 2019 - Merged on: Jun 11, 2019. 2019. URL: https://github.com/eclipse/openj9/pull/6037.

[A50] Hillary Soontiens. *Implement store/load/full/allocationFence for AArch64*. Created on: Jun 24, 2019 - Merged on: Jul 22, 2019. 2019. URL: https://github.com/eclipse/openj9/pull/6250.

[A51] Hillary Soontiens. *Implement Unsafe_compareAndSwapInt_jlObjectJII_Z support for AArch64*. Created on: Aug 16, 2019 - Merged on: Aug 23, 2019. 2019. URL: https://github.com/eclipse/openj9/pull/6752.

Appendix B

In this appendix we see the raw data for Figures 4.6 to 4.13.

Table 6.1: Branch bytecode test execution time in milliseconds for 1,000,000 iterations.

ByteCode Test	Beelink	Pi 4 B	VIM3	Rock64
BranchTests.gotoTest<I>I	143722	155616	132086	224392
BranchTests.ifeqTest<I>I	144920	153990	139384	224631
BranchTests.ifgeTest<I>I	144999	158315	143370	223300
BranchTests.ifgtTest<I>I	144287	155393	143564	223354
BranchTests.if_icmpeqTest<I>I	143084	159884	132567	222706
BranchTests.if_icmpgeTest<I>I	145103	153247	137649	223089
BranchTests.if_icmpgtTest<I>I	145735	155272	140791	224736
BranchTests.if_icmpleTest<I>I	143392	152804	145350	223700
BranchTests.if_icmpltTest<I>I	144612	157800	129658	224068
BranchTests.if_icmpneTest<I>I	142726	155716	139822	223118
BranchTests.ifleTest<I>I	144754	153628	142469	222908
BranchTests.ifltTest<I>I	142969	157963	135078	222646
BranchTests.ifneTest<I>I	142033	158155	136419	223868

ByteCode Test	Beelink	Pi 4 B	VIM3	Rock64

Table 6.2: Conversion Branch bytecode test execution time in milliseconds for 1,000,000 iterations.

ByteCode Test	Beelink	Pi 4 B	VIM3	Rock64
ConversionTests.d2f<D>F	283559	309913	275882	440327
ConversionTests.d2i<D>I	356182	389037	335671	552814
ConversionTests.d2l<D>J	319962	340984	307566	501176
ConversionTests.f2d<F>D	177519	190940	168473	278146
ConversionTests.f2i<F>I	812459	884343	802588	1275244
ConversionTests.f2l<F>J	703507	760204	708348	1104660
ConversionTests.i2b<I>B	143283	155035	139284	222945
ConversionTests.i2c<I>C	143993	153904	153692	223099
ConversionTests.i2d<I>D	143728	152689	149378	226809
ConversionTests.i2f<I>F	177911	190677	186756	278076
ConversionTests.i2l<I>J	143708	156822	150359	223179
ConversionTests.i2s<I>S	213320	230154	208303	334584
ConversionTests.l2d<J>D	144176	155728	135456	224105
ConversionTests.l2f<J>F	251188	284447	246668	392678
ConversionTests.l2i<J>I	144764	154601	137375	224506

Table 6.3: Double Branch bytecode test execution time in milliseconds for 1,000,000 iterations.

ByteCode Test	Beelink	Pi 4 B	VIM3	Rock64
DoubleTests.add<DD>D	879906	1000976	874697	1406823
DoubleTests.dcmpg<DD>I	674880	Segfault	Segfault	Segfault
DoubleTests.dcmpl<DD>I	633950	Segfault	Segfault	Segfault
DoubleTests.dconst_0<>D	38034	51372	36461	58551
DoubleTests.dconst_1<>D	38059	57574	37618	57932
DoubleTests.div<DD>D	846622	965744	819346	1341265
DoubleTests.mul<DD>D	599381	677084	600401	943628
DoubleTests.neg<D>D	320719	358474	327917	497798
DoubleTests.rem<DD>D	1688491	1903972	1664926	2702425
DoubleTests.sub<DD>D	636115	681128	623854	993272

Table 6.4: Float Branch bytecode test execution time in milliseconds for 1,000,000 iterations.

ByteCode Test	Beelink	Pi 4 B	VIM3	Rock64
FloatTests.add<FF>F	883255	1014585	842158	1385740
FloatTests.div<FF>F	847488	981002	822564	1337910
FloatTests.fcmpg<FF>I	568092	Segfault	Segfault	Segfault
FloatTests.fcmpl<FF>I	533441	Segfault	Segfault	Segfault
FloatTests.fconst_0<>F	38109	57723	35853	59467

ByteCode Test	Beelink	Pi 4 B	VIM3	Rock64
FloatTests.fconst_1<>F	38475	43614	34677	59138
FloatTests.fconst_2<>F	38399	40430	35758	59194
FloatTests.mul<FF>F	603363	650832	569338	938556
FloatTests.neg<F>F	284895	303764	270478	441414
FloatTests.rem<FF>F	1696310	1869537	1625641	2679360
FloatTests.sub<FF>F	710487	774624	678572	1104647

Table 6.5: Int Branch bytecode test execution time in milliseconds for 1,000,000 iterations.

ByteCode Test	Beelink	Pi 4 B	VIM3	Rock64
IntTests.add<IIII>I	178675	192119	168575	280960
IntTests.add<III>I	179160	190984	168139	278341
IntTests.add<II>I	178862	195031	169047	280807
IntTests.and<II>I	250412	275892	239372	389025
IntTests.bipush_31<>I	38788	41187	37457	60217
IntTests.bipush_m32<>I	38459	40982	33698	59669
IntTests.div<II>I	299946	306162	283743	444802
IntTests.iconst_0<>I	38790	41109	34089	60059
IntTests.iconst_1<>I	38476	42404	35917	59755
IntTests.iconst_2<>I	39248	41133	38560	60518
IntTests.iconst_3<>I	39400	41206	36878	59654

ByteCode Test	Beelink	Pi 4 B	VIM3	Rock64
IntTests.iconst_4<>I	37978	40699	36230	60002
IntTests.iconst_5<>I	39223	41634	38531	60128
IntTests.iconst_m1<>I	39149	40928	39198	60313
IntTests.inc<I>I	213409	231773	208442	334504
IntTests.mul<II>I	287793	305710	268191	441619
IntTests.neg<I>I	250850	270808	241579	389269
IntTests.or<II>I	251415	272416	245037	388274
IntTests.rem<II>I	367647	386494	333039	550949
IntTests.shl<II>I	496137	532556	482919	770044
IntTests.shr<II>I	674273	718640	635424	1044989
IntTests.sipush_32767<>I	38152	40993	36015	59989
IntTests.sipush_m32768<>I	37725	40632	38829	59780
IntTests.sub<II>I	213743	236175	205847	331214
IntTests.ushr<II>I	667923	721871	650730	1047103
IntTests.xor<II>I	251844	268536	244926	389357

Table 6.6: Invoke Branch bytecode test execution time in milliseconds for 1,000,000 iterations.

ByteCode Test	Beelink	Pi 4 B	VIM3	Rock64
InvokeTests.indirectAdd<III>I	356148	390046	331646	553759
InvokeTests.indirectAdd<II>I	354774	382597	344659	552858

ByteCode Test	Beelink	Pi 4 B	VIM3	Rock64
InvokeTests.indirectAdd<IIJ>J	354018	380093	345078	552297
InvokeTests.indirectAdd_<JII>I	354125	380071	338045	555465
InvokeTests.indirectAdd<JII>J	355115	386159	334625	554474
InvokeTests.indirectDAdd<DD>D	1763919	1884931	1692839	2774904
InvokeTests.indirectDiv<II>I	542328	570017	500133	826913
InvokeTests.indirectFAdd<FF>F	1763447	1944323	1696768	2756988
InvokeTests.indirectLAdd<JJ>J	493998	533256	478922	773461
InvokeTests.indirectMul<II>I	565839	612995	558944	888550
InvokeTests.indirectSub<II>I	423602	454601	408694	661629

Table 6.7: Long Branch bytecode test execution time in milliseconds for 1,000,000 iterations.

ByteCode Test	Beelink	Pi 4 B	VIM3	Rock64
LongTests.add<IIJ>J	180024	192780	172344	279245
LongTests.add_<JII>I	180690	190548	167216	280323
LongTests.add<JII>J	178238	194834	167247	498516
LongTests.add<JJJ>J	179676	193432	170784	283054
LongTests.add<JJ>J	319976	339090	312492	278721
LongTests.and<JJJ>J	37441	39925	35628	498766
LongTests.and<JJ>J	321089	347356	309127	57800
LongTests.div<JJ>J	440858	456323	403834	662488

ByteCode Test	Beelink	Pi 4 B	VIM3	Rock64
LongTests.lcmp<JJ>I	318488	353426	307286	498776
LongTests.lconst_0<>J	38948	41613	34166	59895
LongTests.lconst_1<>J	38746	40695	37356	59023
LongTests.mul<JJ>J	320857	347299	307054	501981
LongTests.neg<J>J	284388	302332	268611	441308
LongTests.or<JJ>J	323674	348179	310483	500520
LongTests.rem<JJ>J	371273	381680	341043	553272
LongTests.shl<JI>J	457375	493765	437809	718963
LongTests.shr<JI>J	666710	731162	638416	1054633
LongTests.sub<JJ>J	319419	353125	307234	497396
LongTests.ushr<JI>J	671048	728577	646922	1046495
LongTests.xor<JJ>J	320123	344408	316156	500158

Table 6.8: Stack bytecode test execution time in milliseconds for 1,000,000 iterations.

ByteCode Test	Beelink	Pi 4 B	VIM3	Rock64
StackTests.dup2Form1<II>I	144487	157293	134831	225492
StackTests.dup2Form2<JJ>J	144850	154280	136717	226834
StackTests.dup2_x1Form1<III>I	143579	155192	136149	223406
StackTests.dup2_x2Form2<IIJ>J	144528	158962	134640	224693
StackTests.dup2_x2Form3<JII>I	143841	159613	141168	225564

ByteCode Test	Beelink	Pi 4 B	VIM3	Rock64
StackTests.dup2_x2Form4<JJ>J	144779	156306	133346	226486
StackTests.dup<II>I	144434	155238	137656	223360
StackTests.dup_x1<II>I	143032	154993	137980	225435
StackTests.dup_x2Form1<III>I	143397	152278	145766	226180
StackTests.dup_x2Form2<JI>I	144453	156448	140599	224539
StackTests.pop2<J>V	177357	188822	149595	273210
StackTests.pop<I>V	142239	157291	112990	219933
StackTests.swap<III>I	141418	153663	104147	219530

Vita

Candidate's full name

Aaron Gary Arnold Graham

Universities attended (with dates and degrees obtained)

- University of New Brunswick, 2013, Bachelors of Computer Science (Cooperative Education Program)
- University of New Brunswick, 2021, Masters of Computer Science

Awards

- IBM CAS (Centre for Advanced Studies) 2020 Project of the Year Award: Project 1035 - OMR in Resource Constrained Environments (Embedded/IoT/-Mobile OMR) (`https://www-01.ibm.com/ibm/cas/canada/awards?award=PotY2020`) [114].

Papers In Progress

- A. G. Graham, J.-P. Legault, H. Soontiens, J. Brown, S. MacKay, G. W. Dueck, K. B. Kent, K. Konno, and D. Maier. "Evaluating the Performance of the Eclipse OpenJ9 JVM JIT on AArch64". Submitted To: 31st Annual International Conference on Computer Science and Software Engineering (CASCON x EVOKE 2021). IBM Corp. Markham, Ontario, Canada, Nov. 22, 2021 [115].

Published Journal Papers

- Kevin E. Murray, Oleg Petelin, Sheng Zhong, Jia Min Wang, Mohamed Eldafrawy, Jean-Philippe Legault, Eugene Sha, Aaron G. Graham, Jean Wu, Matthew J. P. Walker, Hanqing Zeng, Panagiotis Patros, Jason Luu, Kenneth B. Kent, and Vaughn Betz. "VTR 8: High-Performance CAD and Customizable FPGA Architecture Modelling". In: New York, NY, USA: Association for Computing Machinery, June 1, 2020 [116].

Published Conference Papers

- M. Patrou, J. Legault, A. G. Graham, and K. B. Kent. "Improving Digital Circuit Simulation with Batch-Parallel Logic Evaluation". In: 2019 22nd Euromicro Conference on Digital System Design (DSD). Chalkidiki, Greece, Greece: IEEE Xplore, Aug. 1, 2019, pp. 144–151. doi: 10.1109/DSD.2019.00031 [117].

Invited Talks

- J.-P. Legault, A.G. Graham, K.B. Kent, D. Maier, K. Konno. "Evaluating the OpenJ9 JIT on AArch64". In: 4th Workshop on Advances in Open Runtimes and Cloud Performance Technologies (AORCPT 2020), 30th Annual International Conference on Computer Science and Software Engineering (CASCON x EVOKE 2020), Markham, Canada, November 10-13, 2020 [118].

- J.-P. Legault, A. G. Graham, H. Soontiens, M. Flawn, Md.M. Rahman, K. B. Kent, D. Maier, and K. Konno. "Development and Evaluation of the Eclipse OMR Library on AArch64 using the Eclipse OpenJ9 Runtime". In: 3rd Workshop on Advances in Open Runtime Technology for Cloud Computing (AORTCC 2019). IBM Corp. Markham, Ontario, Canada, Nov. 5, 2019 [119].

- J.-P. Legault, A. G. Graham, Dr. K. B. Kent, D. Maier, and K. Konno. "AArch64 Support for the OMR Language Runtime Toolkit". In: TURBO'18 Building Language Runtimes with Eclipse OMR Workshop, SPLASH 2018. IBM Corp. Boston, United States, Nov. 4, 2018 [120].

- A. G. Graham, J.-P. Legault, Dr. K. B. Kent, D. Maier, and K. Konno. "AArch64 Support for the OMR Language Runtime Toolkit". In: 2nd Annual Workshop on Advances in Open Runtime Technology for Cloud Environments, CASCON 2018. IBM Corp. Markham, Ontario, Canada, Oct. 29, 2018 [121].

Conference Posters

- S.S. Ponangi, A.G. Graham, Md.A. Noor, J.-P. Legault, J. Brown, M. Flawn, K.B. Kent, G.W. Dueck, K. Konno and D. Maier, "Eclipse OpenJ9 and Eclipse OMR AArch64 Just-In-Time Compiler Implementation, Baseline and Optimization". In: IBM CASTLE (IBM Centre for Advanced Studies Technical Link Event) 2021, May 10-14, 2021 [122].

- A.G. Graham, J.-P. Legault, K.B. Kent, K. Konno, D. Maier. "Evaluating the Performance of the Eclipse OpenJ9 JIT Compiler on the AArch64 Platform". In: 30th Annual International Conference on Computer Science and Software Engineering (CASCON x EVOKE 2020), Markham, Canada, November 10-13, 2020 [123].

- Md.A. Noor, J.-P. Legault, Md.A. Haque, S.S. Ponangi, A. G. Graham, K. B. Kent, G. W. Dueck, and D. Maier. "AArch64 Support for the Eclipse OMR Language Runtime Toolkit & Eclipse OpenJ9". In: IBM CASTLE (IBM Centre

for Advanced Studies Technical Link Event) 2020. IEEE Xplore. Markham, Ontario, Canada, May 11, 2020 [124].

- A. G. Graham, J.-P. Legault, H. Soontiens, M. Flawn, Md.M. Rahman, K. B. Kent, D. Maier, and K. Konno. "Building AArch64 Support for Eclipse OpenJ9 and Eclipse OMR". In: 29th Annual International Conference on Computer Science and Software Engineering (CASCON x EVOKE 2019). IBM Corp. Markham, Ontario, Canada, Nov. 5, 2019 [125].

- J.-P. Legault, A. G. Graham, Kenneth B. Kent, and D. Maier. "XDocker: A Cross-Platform Tool for Continuous Development". In: 29th Annual International Conference on Computer Science and Software Engineering (CASCON x EVOKE 2019). IBM Corp. Markham, Ontario, Canada, Nov. 5, 2019 [126].

- A. G. Graham, J.-P. Legault, M. Patrou, and K. B. Kent. "Improved Synthesis and Simulation for FPGAs: ODIN II for VTR 8.0 and Beyond". In: 16th Annual Research Exposition of the UNB Faculty of Computer Science. University of New Brunswick. Fredericton, New Brunswick, Canada, Apr. 12, 2019 [127].

- J.-P. Legault, A. G. Graham, Dr. K. B. Kent, D. Maier, and K. Konno. "AArch64 Support for the OMR Language Runtime Toolkit". In: CASCON 2018. IBM Corp. Markham, Ontario, Canada, Oct. 29, 2018 [128].

- J.-P. Legault, A. G. Graham, Dr. K. B. Kent, D. Maier, and J. Kingdon. "Language Runtimes on Embedded Architectures". In: IBM CASTLE (IBM Centre for Advanced Studies Technical Link Event) 2018. IBM Canada Lab. Markham, Ontario, Canada, May 8, 2018 [129].

- J.-P. Legault, A. G. Graham, Dr. K. B. Kent, D. Maier, and J. Kingdon. "Language Runtimes on Embedded Architectures". In: 15th Annual Research Exposition of the UNB Faculty of Computer Science. University of New Brunswick. Fredericton, New Brunswick, Canada, Apr. 11, 2018 [130].