Prototype Eclipse OMR Port Performance Evaluation on AArch64

by

Jean-Philippe Legault

Bachelor with Honors of Computer Science, UNB, 2018

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

Master of Computer Science

In the Graduate Academic Unit of Computer Science

Supervisor(s):	Kenneth B. Kent, Ph.D., Computer Science
Examining Board:	Gerhard Dueck, Ph.D., Computer Science, Chair
	Panos Patros, Ph.D., Computer Science
	Julian Cardenas-Barrera, Ph.D., ECE

This thesis is accepted by the

Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

November, 2020

Abstract

This thesis discusses the steps taken to build the prototype Eclipse OMR port to the AArch64 architecture. The AArch64 OMR port is evaluated using Eclipse OpenJ9 against an AMD64 counter-part (similar cache size and clock speed); The results are used to build a baseline for future research and provide an evaluation framework upon which further enhancements to the platform can be compared. This thesis also reviews the AArch64 hardware landscape and its suitability for a development platform. AArch64 is reviewed in terms of software availability and ease of use. Developing for embedded devices adds a layer of difficulty to software development that cannot be ignored and this thesis reviews the usability of modern development tools on the AArch64 ISA. This thesis offers an experience review on the AArch64 ISA with modern software as well as the viability of a high-performance runtime on AArch64.

Acknowledgments

I would like to thank Aaron Graham for guidance and help throughout the project. Without his support, both as a best friend and a coworker, this thesis could never have seen daylight. Words cannot say how thankful I am to him and his wife to have kept me sane through this and the pandemic. Hats off to a great man, and I hope our friendship continues for a long time, even across borders.

To my supervisor, Kenneth B. Kent, you have been a skilled and patient teacher. You have allowed me to realize my capacities and pushed me to do better. I have grown tremendously because of you, and I would never have gotten this far without your help. You have helped me become a better leader and have rekindled my passion for teaching. You have kicked down doors and showed me that there's absolutely nothing stopping me as long as I work for it. I will remember my time under your tutelage fondly.

To my honors mentor, Panos Patros, thank you for opening my eyes to the research happening at UNB. Your enthusiasm for teaching and your research got me interested in this. Without your passion, I wouldn't have walked this path. I am hopeful of working with you again.

To Stephen MacKay, the amazing editor behind the scenes, I apologize for the million french-isms and inconsistencies. Thank you for helping to make this piece a thousand times better. I have enjoyed working and sharing an office space with you, and I wish you the best.

To the team at IBM: Daryl Maier, Leonardo Banderali, and Kazuhiro Konno, it has been great working with you; you are a great source of inspiration, and your patience is remarkable. Your depth of knowledge is remarkable, and I am grateful you shared some with me.

To the people that have worked, are working in the Odin II team, thank you for your patience. With this piece done, I can finally spend more time with you and help.

To my family: Je l'ai fait! J'en ai un autre diplôme! C'est principalement grace à vous, sans votre support et votre persévérance, je n'aurais jamais réussi a aller aussi loin. Ce diplôme-ci je vous le dois aussi. Je vous aime maman et papa, merci. À Isabelle, Melanie, Karine: je m'ennuie vraiment beaucoup de vous. Ça me fend le coeur de devoir rester aussi loin, mais je reviens bientôt. Merci de m'avoir écouté sans juger, de m'avoir supporté et de m'aimer. À Arthur, Élodie, et Henri: vous êtes rendu tellement cool et je suis fière de vous autre. À bientôt.

Finally, to my fiancee, Julie Brown. You have seen me go through my honors, and now this. You have read each one of my pieces and saw me grow and get closer to my goals. You have patiently stood by me through thick and

thin. Thank you love, so much. I am looking forward to a lifetime with you, watch you achieve your dreams, and grow old with you; I do.

Although the last two years have been tough, they have also been an incredible source of growth and have armed me for what is next.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The workload demand on cloud infrastructures requires continuous refinement of virtual private server collocation techniques to prevent security breaches. With the increasing number of vulnerabilities discovered in virtualized environments, requests for physically segregated systems have been steadily growing. However, adding physical nodes in a data-center increases energy use and, with the power draw of high-performance computing nodes, adding a new node to data-center stresses the electrical infrastructure. A proposed solution is to move services onto low-powered devices with a reduced instruction set and clock speed, but this only defers the inherent problem.

Decentralizing services with peer-to-peer computing offers the possibility of better throughput, redundancy, and resiliency. The proposed technique is to move the computation closer to the originating point of the data—Edge

Computing. Doing so can prevent sensitive data from ever leaving their owner, improve security, and reduce the computational load put on cloud infrastructures. Moving the computation between different Instruction Set Architectures (ISAs) requires the use of portable languages.
A multitude of ISAs (MIPS [1], RISCV [2], ARM [3]) are the building blocks of smart and Internet-of-Things (IoT) devices. Adapting user applications onto these new ISAs requires development, recompilation, and testing of the relevant software components. Portable languages, such as Java and Python, can run on any platform supported by its interpreter.

1.1 Problem Statement and Contribution

This research adds basic support for the AArch64 platform to the Eclipse OMR Runtime toolkit [4] and evaluates its performance using Eclipse OpenJ9 [5]. This research is responsible for the evaluation and the validation of a high-performance runtime in a resource-constrained environment. This thesis evaluates the AArch64 platform landscape and its usability for development. With ARM devices leading the IoT movement, enabling a high-performance runtime like OpenJ9 simplifies software development on the ARM platform. Sharing the server runtime with IoT devices further simplifies moving computation between the client and the server.

1.1.1 High-Performance OpenJ9 for the IoT Space

While an enterprise-grade language runtime has a reduced target audience, this thesis ports Eclipse OMR and Eclipse OpenJ9 to AArch64 to show that a high-performance runtime is suitable for IoT devices. With the increasing demand for a secure and high-performance language runtime on low-powered systems, Eclipse OMR is selected since it fulfills the requirements for this effort. Giving OMR the ability to support an embedded platform adds this support to any language using the OMR runtime toolkit. Eclipse OpenJ9, a Java Virtual Machine, leverages OMR for its runtime core and is used for this thesis due to its robustness and years of development effort behind it. This thesis reviews the steps taken to create the initial building blocks of OMR on AArch64. Once completed, Eclipse OpenJ9 is benchmarked to review initial performance results and be used as a baseline for future research on improving the Eclipse OMR port to AArch64.

1.1.2 AArch64 Landscape and Suitability

To achieve an AArch64 version of OpenJ9, devices are acquired to evaluate the tool and help the development phase. This thesis serves both as a performance evaluation of OpenJ9 on Aarch64 and as an experience review of working with AARch64 as a development platform. Many may follow this path and this thesis gives guidelines to promote faster development, and prevents the same errors from being repeated. Four important facets of software development are reviewed to evaluate the AArch64 ISA as a developer's platform:

- Debugging
- Building
- Performance evaluation
- Reproducible builds

1.2 Summary of Thesis

Chapter 2 reviews the necessary knowledge required to understand the work achieved, benchmarking Java Virtual Machines, and the AArch64 landscape. Chapter 3 explains the work done on OMR, and the tools created to alleviate some pain points of the development cycle. With a minimum viable interpreted JVM in hand, Chapter 3 benchmarks its performance. The work completed to obtain a usable AArch64 device for this research is reviewed in Chapter 4 as well as some of the platform's drawbacks. Lastly, Chapter 5 lists the future work for consideration on OpenJ9 for AArch64.

Chapter 2

Background

This thesis offers software developers a cross-platform, high-performance runtime that allows seamless migration of logic between the host and the client. The following sections provide the necessary background in the areas relevant to the work. First, language runtimes are explained, then, OpenJ9, the language runtime selected for this work, is described. Then, the AArch64 ISA and its ecosystem are explored. Finally, the various benchmarks used for the evaluation of the runtime are explained.

2.1 Interpreted Languages

Using interpreted languages provides the ability to target many instruction set architectures (ISAs) easily. Unlike compiled languages, interpreted code is transpiled into a hardware-agnostic assembly-like language called intermediate language (IL). The software is then delivered as transpiled code to be interpreted on the client, and optimized for the target instruction set. Doing so provides the visible advantage that the software developer need not worry about the ISA, but only the interpreter and its lower-level API.

The most significant performance gain a language runtime can have is when hot regions of code can be compiled directly into binary instructions—Just-in-Time (JIT) compiled [6]—rather than being interpreted. Selectively compiling to machine code rather than interpreting improves the performance of the running software [6]. JIT compiling enables numerous optimizations like function in-lining, which inserts the function body of the callee into the caller to reduce the number of function calls and avoid building unnecessary stack frames [7].

The software responsible for interpreting the code and managing the system during its execution is the runtime virtual machine. The virtual machine implements a virtual instruction set. The runtime virtual machine is responsible for memory management, interfacing with the OS and the hardware, interpreting, JIT compiling, and garbage collection [8]. A long-running system requires the reclamation of allocated but unreachable memory, namely garbage collection, to allow continuous usage. Along with the JIT, garbage collection increases resource utilization in short bursts [8]. The JIT and the garbage collector put a strain on systems and lessen the appeal of language runtimes on resource-constrained devices [9]. Furthermore, many embedded devices use small instruction sets with a single-core processor (16-bit micro-controllers), which prevents parallelism from being used in language runtimes. With the rise of ubiquitous computing, IoT everywhere and smart devices, more powerful multi-core embedded processors with fully featured operating systems are now available.

2.2 Eclipse OpenJ9 Java Virtual Machine

Eclipse OpenJ9 [4] and Eclipse OMR [5] are Eclipse projects backed by IBM's decades of experience in language runtimes. The Eclipse OpenJ9 and Eclipse OMR duplet stem from IBM's effort to open-source their software and extract generalizable VM pieces into separate tools: OpenJ9 and OMR, respectively. The OpenJ9 JVM is the layer responsible for interpreting/compiling Java bytecodes (a transient language between high-level languages and machine code) into an Intermediate Language internally used by other components comprising the software [10, 4]. OpenJ9 implements the Java specification and, by consuming OMR, provides a complete virtual machine with many advanced features such as a shared class cache, Ahead of Time (AOT) compilation, and code caching.

The OMR toolkit reads an intermediate language and, from a set of instructions, optimizes and compiles the given code into architecture-specific code. OMR is also responsible for many of the runtime features of the OpenJ9 and OMR duplet.

OMR is a C++ library of runtime components that can be selectively used

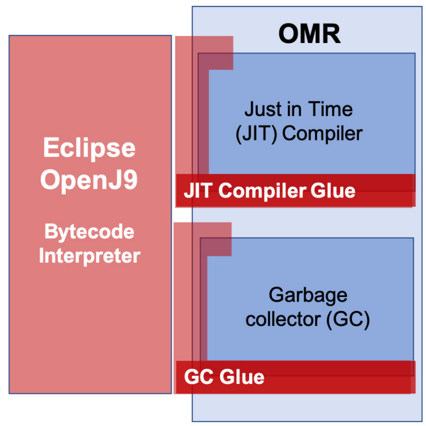

Figure 2.1: OMR Runtime Toolkit example infrastructure and components

by other tools to add high-performance garbage collection, Just-in-Time compilation, and other runtime components (Figure 2.1). It is a robust toolkit under continuous development that allows OMR to be both modern and mature by relying on the maintainers' previous experience. OMR is consumed at compile time via glue components. Moreover, it has platform support for many architectures, and more are under development. To allow different platform support, OMR uses build-system variability [11] and compile-time polymorphism. Build-system variability leverages folder structures to hide or include a particular part of the source code, thus, varying the build. The Linux kernel heavily relies on build-system variability and is an excellent example of the robustness of the technique. OMR uses build-system variability and static polymorphism to deliver product variability and keep the binary code size small.

OMR was developed to allow the rapid development of language runtimes (see

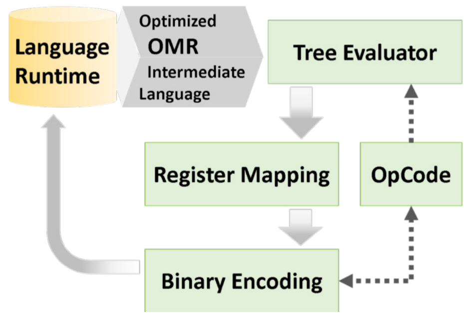

Figure 2.2: OMR infrastructure and components relation

the Ruby+OMR project [12], and the Lua+OMR project [13]). OMR offers a toolkit to rapidly develop a JIT for a language, namely JIT-Builder [14, 5]. OMR depends on the instruction set of each target architecture. OMR directly writes executable code by using the binary encoding of the target architecture without resorting to any external tool. OMR uses tree evaluators (Figure 2.2) to translate the Intermediate Language (IL) used for the OMR compiler to a set of hardware instructions. Hardware threads, locks, and OS support are added so that the OMR low-level API can interface with its host system. This thesis limits itself to supporting a Linux kernel-based operating system.

2.3 Linux Kernel

A kernel is responsible for providing the lowest level of interaction between an operating system and the hardware used. A kernel provides a low-level API to allow interaction between the connected devices and the software. The kernel manages the memory and the tasks of the operating system [15]. For our work, the Linux kernel is primarily used [16]. Linux was developed by Linus Torvald almost 30 years ago and is now the most used kernel in the world. Linux powers almost everything, from a toaster oven to the CERN cluster [17]. The Linux kernel offers a collection of tools to help debug and evaluate the performance of the kernel, the hardware, and any software running on it. Developers use tracing tools to obtain performance metrics on the hardware under test and to identify long-running or heavily used functions (hot regions) in the code base. Linux perf is the tracing tool of the Linux kernel and gives access to hardware and software performance counters at the kernel level [18]. This provides insight into the running software's memory use, cache misses, CPU cycles, and more. This thesis extensively uses Linux perf to evaluate the performance and review problems affecting certain hardware.

2.4 ARM and the AArch64 Instruction Set

The target of this research is the resource-constrained variants of the AArch64 ISA by ARM (Advanced RISC Machine) [3]. The ARM processor powers a large percentage of IoT and smart devices. The ARM Holdings company is responsible for designing microprocessor architectures. The ARM architectures, utilizing the RISC (Reduced Instruction Set Computer) model, are available in 32-bit (AArch32) or 64-bit (AArch64) configurations. ARM itself does not produce the processor, but licenses and collects royalties for its architectures [3]. Most of their offerings are multi-core and can support high-performance computing with complex floating-point operations in hardware. The 64-bit variant of the ARM platform is fully backward compatible with its 32-bit variant using the AArch32 instruction set. AArch64 offers 31 64-bit registers while AArch32 has 13 32-bit registers. AArch64 possesses floating-point arithmetic units on hardware as a standard. The AArch64 instruction set has high-performance core offerings with better cache and clock speeds and can pack tightly enough to fit hundreds of cores on a single CPU die. All these features place the ARM 64-bit ISA as the best contender to evaluate an enterprise-grade runtime on an embedded device. The increase in performance of this platform [19] and the growing demand for reduced instruction set devices (Figure 2.3) makes the AArch64 ISA a perfect target for OpenJ9. While the AArch64 ISA is an excellent candidate for this research, this particular ISA did not come without hurdles.

2.4.1 ARM Hardware Fragmentation

The ARM licensing model can hinder growth due to the considerable fragmentation of the physical System-on-Chip (SoC) implementation. Although ARM offers many advantages, the number of vendors implementing the ISA makes the performance widely different for the same core across different implementations. Vendors tend not to push their changes onto the main kernel repository for various reasons, and this provides varying performance. The state of AArch64 as a software development platform is evaluated for its

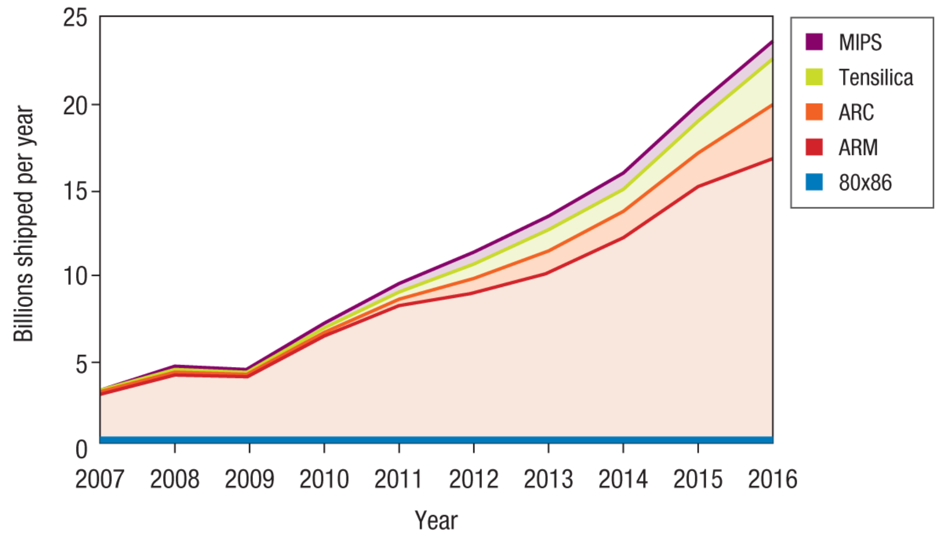

Figure 2.3: Billions of Devices Sold over Time with Breakdown of Architectures. [19]

suitability, and to gain insight into its slow entry as a mainstream platform. To do so, the use cases of the AArch64 ISA are divided into three profiles as per the ARM profiles available.

2.4.1.1 Embedded and Real-Time devices (ARM M-profile and R-profile)

Embedded controllers use a real-time operating system and have strict timing constraints. Embedded devices are expected to be always on and have low memory and compute power, e.g., Internet of Things (IoT) devices or smart controllers. The ARM has specific processors to serve this market called the Cortex-M profile for micro-controllers and Cortex-R profile for Real-Time processors [20]. Examples of competitors are Atmel micro-controllers in Arduino devices.

2.4.1.2 Entry-level compute devices (ARM A5x-profile, and big.LITTLE)

Entry level devices include TV-boxes, smart-phones, and entry-level computers. They use a Linux kernel and have esoteric graphic processors with mostly closed-source drivers [21]. The A5x profile is aimed at power-oriented devices, it provides low powered general-purpose cores. The big.LITTLE architecture uses a high-performance core from the A7x profile and a low-powered core from the A5x profile. The A7x profile is aimed at performance-oriented devices, it provides high-performance cores. Using the big.LITTLE schema allows both a low power usage and high performance when an appropriate operating system scheduler is in place.

2.4.1.3 Server class devices (A7x-profile)

Fewer chip providers are active in this spectrum since the demand is still low. The biggest advantage ARM-based processors offer to the server market is its high core count from the ISA's small footprint. Most notably, two manufacturers are involved: Qualcomm and Cavium. The server class processor offers mainline kernel and developer support, unlike many entry-level devices.

2.4.2 Cross Platform Development

As discussed previously, the largest drawback of the ARM processor compared to more mainstream processors is a large number of implementations, leading to thinly spread kernel developer effort. This can lead to erratic performance across different vendors, and even across devices from the same vendor. This issue is further exacerbated by a whole spectrum of untouched devices: middle-class hardware. Middle-class devices provide both high-performance cores and high-performance peripherals. Computers used for modern games are of this segment of hardware. There is a lack of devices that can achieve the minimum performance, peripheral support, and cost that mainstream platforms, e.g. AMD64, offer. The lack of availability leads to a more challenging time for developers when trying to generate executables for the target ISA. There are three main methods to generate executables for the platform:

- using a cross-compiler on a different host than the target architecture;
- using hardware emulation to provide a "native" host compiler within the emulated environment; and
- using a compiler on a native host.

Native host compilers are preferred when the hardware is available and performant enough, otherwise, cross-compilers are preferred. The biggest issue with cross-compilers is that they can be temperamental and still require a native host to test the software. Hardware emulation can provide a solution to both problems by interpreting the native executable for a different host.

2.4.3 QEMU

"*QEMU* is a generic and open source machine emulator and virtualizer" [22]. *QEMU* allows the user to emulate an ISA on a different ISA. *QEMU* offers two primary levels of emulation (more are available, but irrelevant to our work):

- full system emulation: emulates the ISA and requires a kernel to be installed with a root filesystem to do any useful work.
- User-space emulation: does rapid binary translation of a user application compiled for a different architecture.

QEMU is pivotal to creating software for an ISA not yet available or hard to obtain on good hardware, such as RISC-V or AArch64. Alongside *QEMU* user-space, *binfmt* automatically interprets an executable on the host.

2.4.4 Binfmt

Binfmt is a new addition to the Linux kernel [23]; it reads an executable and linkable format (ELF) binary and, from the magic number in the header, automatically hands it off to the right interpreter. This yields extremely flexible and powerful side-effects. When used in conjunction with an emulator, *binfmt* is capable of running binaries from any platform, without user interaction.

2.4.5 Docker

Docker [24] is a containerization tool and is, essentially, a *chroot* with virtualized devices. To understand Docker, a quick introduction to *chroot* is necessary. *Chroot* stands for "change root," and using the predicate "everything is a file," in Linux, change root allows us to hide everything above the specified new root directory. Docker provides the same functionality and allows a prebuilt and reconfigurable root filesystem to populate the *chroot* and give the API access to the hardware securely. Docker allows reproducible environments and packages to build and distribute software to the masses.

2.5 Benchmarking

According to research by Prokopec et al. [25], benchmark suites have played a crucial role throughout the JVM's history. Benchmark suites provide metrics that differentiate the effectiveness of various language runtime implementations and optimizations. This allows us to reveal the implementations that are worth pursuing. Earlier Java benchmarks include Java Grande [26], Jolden [27], Ashes [28], SPECjvm®98 [29], and SPECjbb®2000 [30][1]. The

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

SPECjbb®2000 output score is in terms of work completed over a fixed amount of time, instead of considering the elapsed time for a fixed workload [31]. However, Blackburn et al. [31] asserted that it was difficult to analyze performance if variable workloads were considered. The DaCapo suite (2006) [31] is considered a *de facto* standard [25] to analyze the performance based on the object and memory behavior in complex Java applications [25]. On the other hand, the SPECjvm®2008 benchmark suite focuses more on the core Java features [29]. The SPECjvm®2008 benchmark suite emphasizes optimizations based on the JIT compiler [32] instead of object-oriented abstractions. This allows the benchmark suite to be comparatively smaller than DaCapo [25]. For similar reasons, Blackburn et al. [31] stated that the DaCapo benchmarks were more substantial, more complex and richer than the commonly used SPEC® Java benchmarks. The SciMark benchmarks evaluate the performance of common scientific and engineering applications. SciMark heavily relies on matrices and floating-point arithmetic [33]. The SPECjvm®2008 benchmark itself contains the SciMark benchmark.

Chapter 3

OpenJ9 Performance Evaluation on AArch64

This chapter provides a baseline for future research with OpenJ9 on AArch64, but also sheds light on the performance discrepancies of the implementation in general. The interpreter is benchmarked separately from the JIT to evaluate both components independently. The AArch64 platform is compared to a mature platform—AMD64—to validate the effort. The AArch64 TRJIT evaluated in this thesis is a minimal prototype and only performs local code optimizations and generation. Many TRJIT performance enhancements have yet to be implemented for AArch64 such as: tiered method recompilation, object allocation, write barriers, “checkcast”, and “instanceof”.

3.1 Design and Implementation

This thesis primarily focuses on the viability of the AArch64 platform for Eclipse OpenJ9 and Eclipse OMR, and as such, the following sections explain the work completed as well as the necessary infrastructural components built.

3.1.1 OMR

This work added the necessary directories and stubbed the source files necessary to complete the main code path and the build path for AArch64. Specific to the AArch64 assembler, compiler and linker flags were added to the makefiles—ARMv8 version 3 with NEON [34] enabled. The project later moved to CMake to simplify the build system and to add multi-platform support (Windows, Linux and macOS). The complete range of opcodes and binary encoding for the AArch64 platform is arranged in a spreadsheet [35] to auto-generate the three tables—arrays and enums to provide reflective enumerations—required for OMR. The register map is written to allow the virtual register mapper to use the hardware registers. Both the portability library [36]—enabling OMR to interface with the Linux Operating System (OS)—and the thread library—responsible for threads—are written to complete the low-level API.

The build logic is written to allow OMR to compile the essential infrastructure and interface with the hardware. To quickly implement the tree evaluators described in Section 2.2, function prototypes containing no implementation body, also called "stubs", are added to have the minimum required declaration for the codebase to build. The ISA support is added to the Continuous Integration (CI) and Development (CD) test farm, namely Travis [37], to ensure that future changes do not break the ability to build. CI assures better progressive support for the platform. No sufficient AArch64 development boards are available and thus, builds have to be cross-compiled or emulated. In-house tool kits and scripts are used to abstract away the complexity of building a custom toolchain and a root filesystem for the target architecture.

In this phase, the tree evaluator "stubs" body are filled with their implementation. To assist in generating Testarossa JIT (TRJIT) Intermediate Language (IL) tests, a Domain Specific Language (DSL) called TRIL is used [38]. TRIL tests are developed to individually test (unit-test) each tree evaluator implemented. Using TRIL to verify correctness, the body of each tree evaluator is filled with its AArch64 implementation. The body of a tree evaluator combines the desired implementation in assembly using the binary encoding and the OMR specific function calls. The body uses method calls to the virtual register mapper and the binary encoding created before to generate real machine code. The binary encoding is later masked by the register mapper from the infinite virtual register set.

3.1.2 OpenJ9

In this section, Openj9 is attached to the prototype OMR to begin the evaluation. While a substantial effort was made on porting OpenJ9 to AArch64, this is outside the purview of this work—it aims to validate an eclipse OMR on AArch64—and as such, only a quick overview is offered. At this point, the necessary stubs and makefiles are added, similar to OMR, to build OpenJ9 on the AArch64 platform. This phase is completed blindly until the JVM works, which requires major components. Unlike the OMR build phase, OpenJ9 does not consume the AArch64 ISA but consumes the OMR tree evaluators that have been implemented or stubbed (Figure 3.1). Iteratively, the tree evaluators are built as the necessity arises, dictated by introducing new test cases.

3.2 Methodology

With a minimal OpenJ9, benchmarking is done to evaluate the viability of the prototype. Early results show that the AArch64 platform lends itself to be a good embedded device for OpenJ9.

Tools are created for the reader to reproduce the results in this thesis. The JVM benchmarking toolkit [39] is a collection of scripts to allow the reader to reproduce our tests. The tool locks the CPU speeds at 1.4 GHz to allow each board to be as close as possible in terms of speed. There is a low-level monitoring tool for temperatures since some AArch64 devices do not have

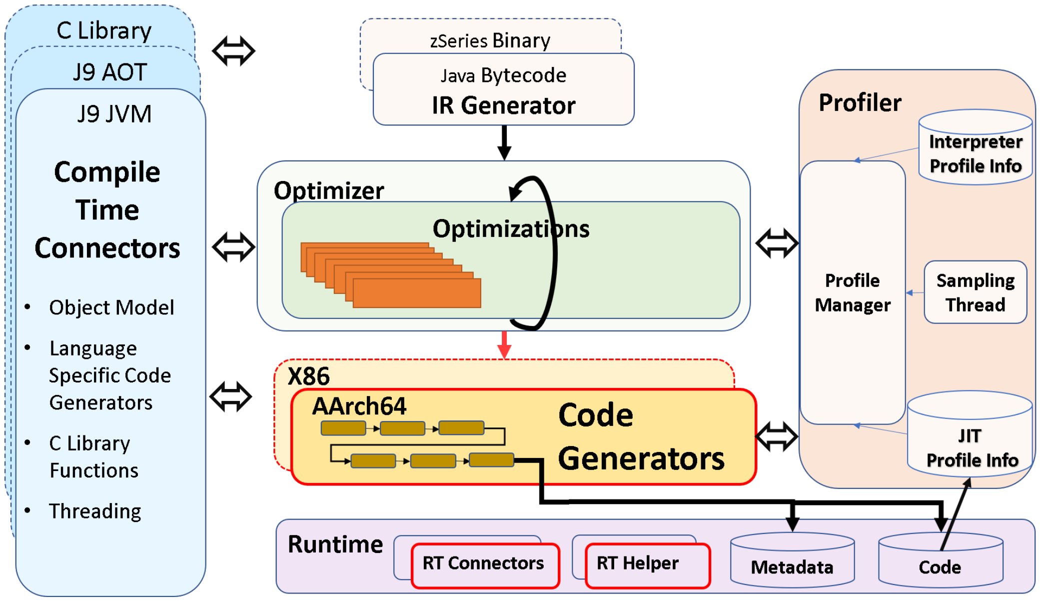

Figure 3.1: OpenJ9 and OMR infrastructure and components relation

any tools for this. The temperature monitoring is vital since the development board throttles the CPU if critical temperatures are attained. Furthermore, the tool is responsible for importing all the benchmarks and to automate the configuration. It uses pyBenchParser [40] to simplify the parsing of logs. The parse tool uses configuration files to dictate how to parse log files and generate CSV or JSON tables. It is currently extensively used in the vtr-verilog-to-routing [41] project and allows us to quickly aggregate our results by parsing standard log files.

For benchmarking, SPECjvm®2008 *Lagom* flag is used to make benchmark results portable across different benchmarks, architectures and VMs [42]. The *Lagom* flag changes the benchmarks to run a specific amount of workload; this allows us to use the total running time as our metric, rather than throughput;

Table 3.1: AMD64 embedded board specifications.

Architecture	AMD64
Core	4-core Intel X5—Z8350
Max Speed	1.92GHz
Locked Speed	1.44GHz
Cache	L1–32KB L2–2MB
Mem	4GB DDR3@1600mhz

Table 3.2: AArch64 embedded board specifications.

Architecture	ARMV8—AArch64	ARMV8—AArch64
Core	4-core A53	4xA73–2xA53
Max Speed	1.5GHz	2.2Ghz
Locked Speed	1.5GHz	1.48Ghz
Cache	L1–32KB L2–256KB	L1–32KB L2–1MB (shared)
Mem	4GB DDR3@1600mhz	4GB DDR4@3200mhz

this is done to obtain uniformity between all the benchmarks. The DaCapo benchmark suite is used to compare Java 11.0 performance of OpenJ9 on AArch64 and AMD64.

Tables 3.1 and 3.2 provide details of the systems that are used for the rest of this thesis.

Arch Linux with kernel version 5.5 is used on all the development boards. The version of the JVM that is benchmarked is OpenJDK11 OpenJ9 [43]. The early access release 0.20.0 from AdoptOpenJDK is used to benchmark the JIT. The commit hash *efea76697* from AdoptOpenJDK is used for the interpreter. The JIT is also evaluated to provide an initial assessment and substantiate the endeavor's viability.

The parameters used for running OpenJ9 are:

```
–Xms2G –Xmx2G –Xgcpolicy:optthruput \
  –Xgcthreads2 –Xgc:excessiveGCratio=95
```

The *optthruput* garbage collector utilizes parallel threads for garbage collections and produces a similar result to *UseParallelGC* from *HotSpot* in Appendix 1 [45, 44]. 5% of the total time for GC and a throughput goal of 95%, as specified by *Xgc:excessiveGCratio=95*, is used for both JVMs, as used by default on OpenJ9. Using *GCTimeRatio=19* produces an equivalent proportion given *HotSpot* utilizes a denominator of 20. Two of the four available threads are used for both JVMs, as seen with *gcthreads2*, since the embedded devices under test freeze under heavy load. Using two out of the four threads available for the garbage collector results in stable throughput and performance. The memory limit and initial usage is set to 2 gigabytes, half the usable memory, for both JVMs, as seen by *-Xms2G -Xmx2G* (respectively). Most importantly, the JIT is turned off for all our tests using *-Xint*.

First the Linux perf executable execution time is compared against a lighter weight tool, *Gnu Time*, to assure that it does not cause a slowdown; this is done because the platform uses a custom kernel. As shown in Figure 3.2, Linux perf induces, on average, a 0.035% relative slowdown on the SPECjvm®2008 SciMark subset and DaCapo (Chapter 2) benchmarks with a standard deviation of 0.35%. Linux perf provides only a marginal slowdown, and the runtime seems to be unimpeded. Then, the platforms are benchmarked using perf. The benchmarks are run only once since they comprise many iterations of a single benchmark. The total execution time is recorded since it is of interest to analyze the complete execution time from the JVM startup to its teardown.

3.3 Results

The SciMark 2.0 Java Sparse matrix multiply (Sparse) and Jacobi Successive Over-relaxation (SOR) benchmarks in Figure 3.3 produces no notable differences in execution time between the interpreter and the JIT. The similarities are attributed to the incomplete nature of the JIT and the lack of performant floating-point support. The SOR and Sparse benchmarks perform floating-point computation on a massive grid; vectorized capabilities are essential to improve performance. Monte-Carlo integration in Figure 3.3 is problematic for the A73 board. These results are further explored in Section 3.3.0.1.

Surprisingly, in the DaCapo benchmarks shown in Figure 3.4, the A73-based machine's performance often matched or performed better than its AMD64 counterpart. DaCapo benchmarks are memory-bound [46], leading to much better performance when the memory frequency is higher as in the A73 system under test; the same behavior is displayed in the SPECjvm®2008 SciMark large subset (Figure 3.3), whereas the small subset (Figure 3.3) exercises cache and CPU speed. The SPECjvm®2008 *crypto RSA* and *signverify* (Figure 3.5) benchmarks on the A73 platform with the JIT enabled outper-

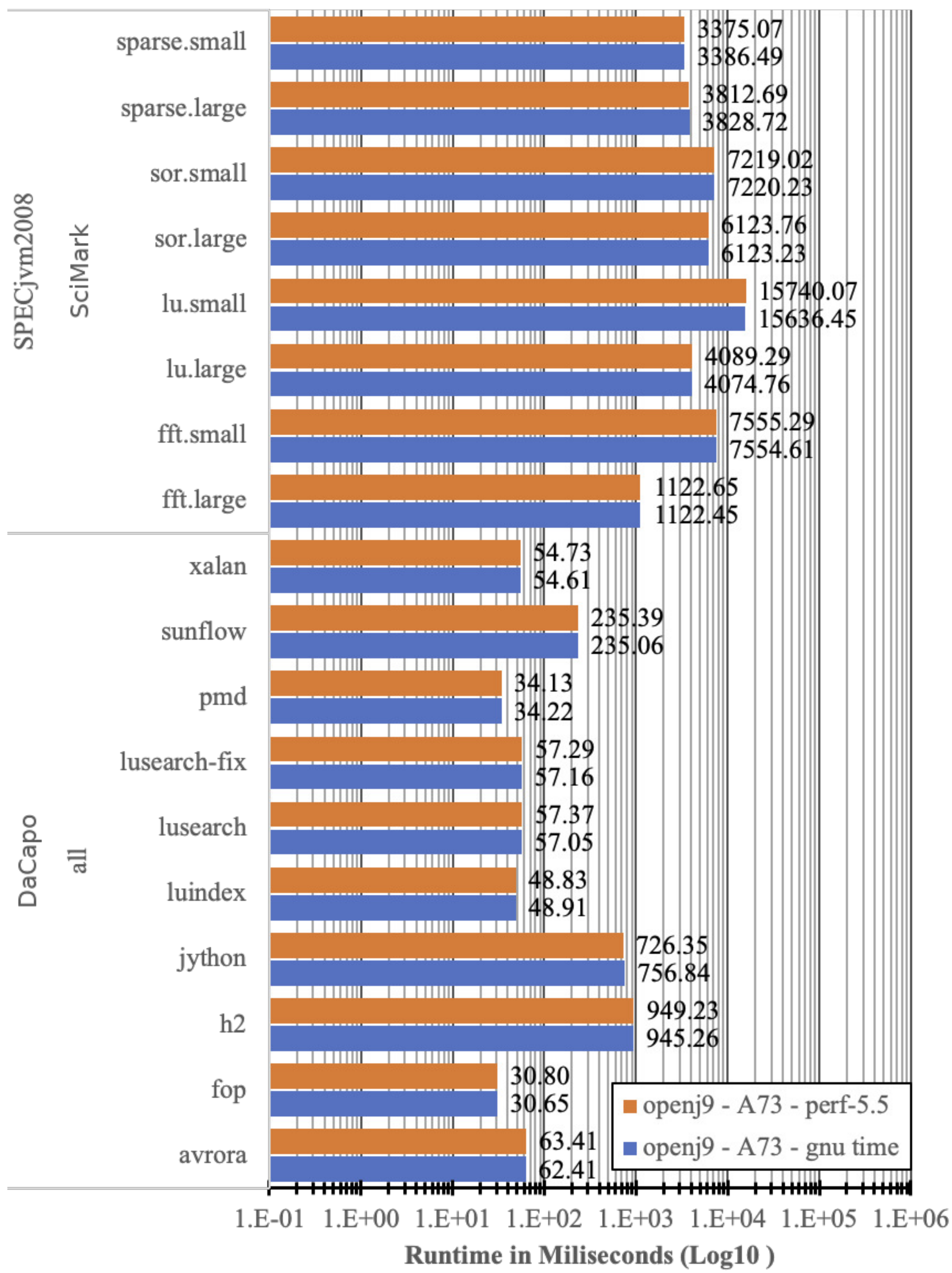

Figure 3.2: Comparing SPECjvm®2008 SciMark and DaCapo benchmarks execution times on A73 cores using GNU time and perf.

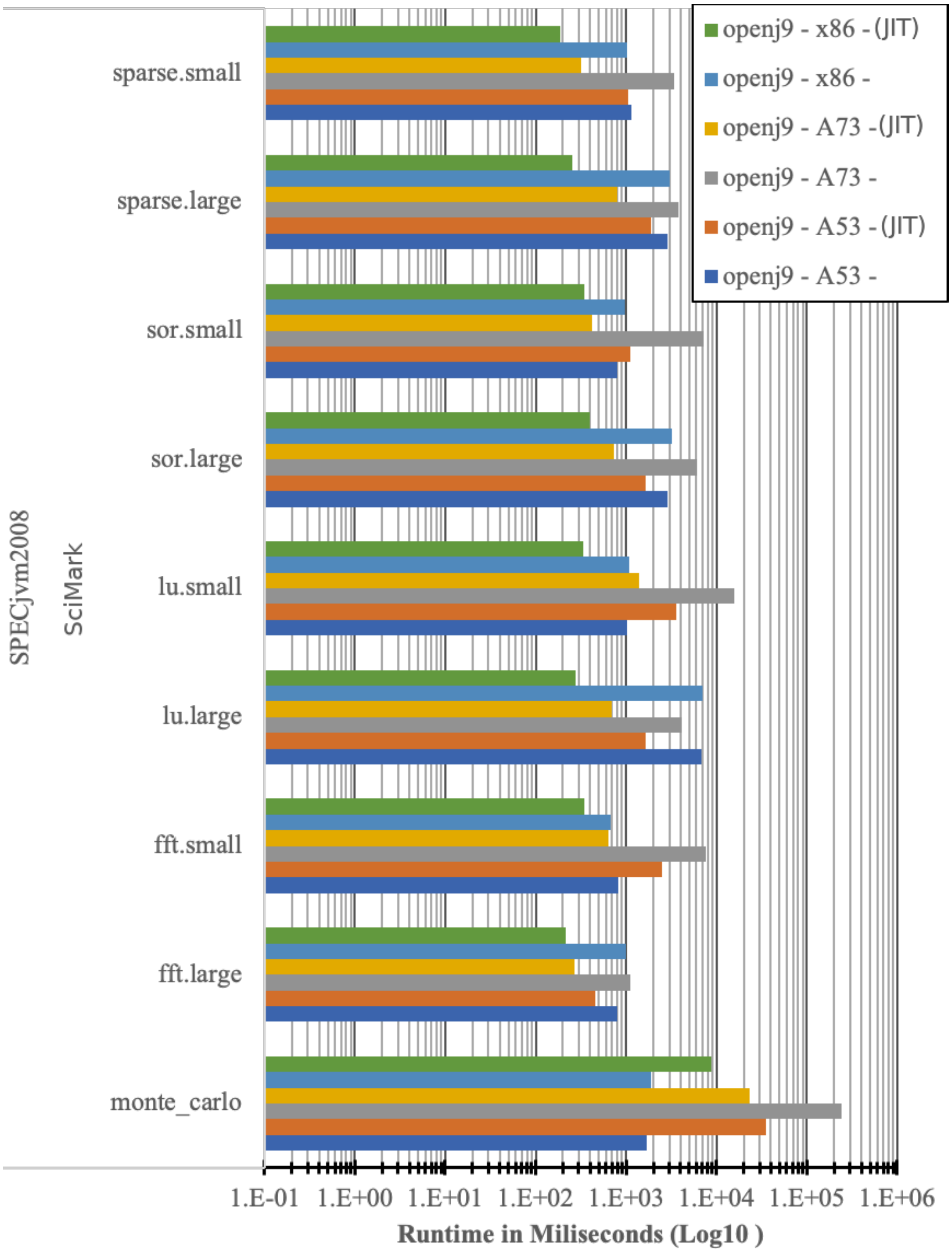

Figure 3.3: SciMark 2.0 benchmarks relative speedup for different architectures and different VMs

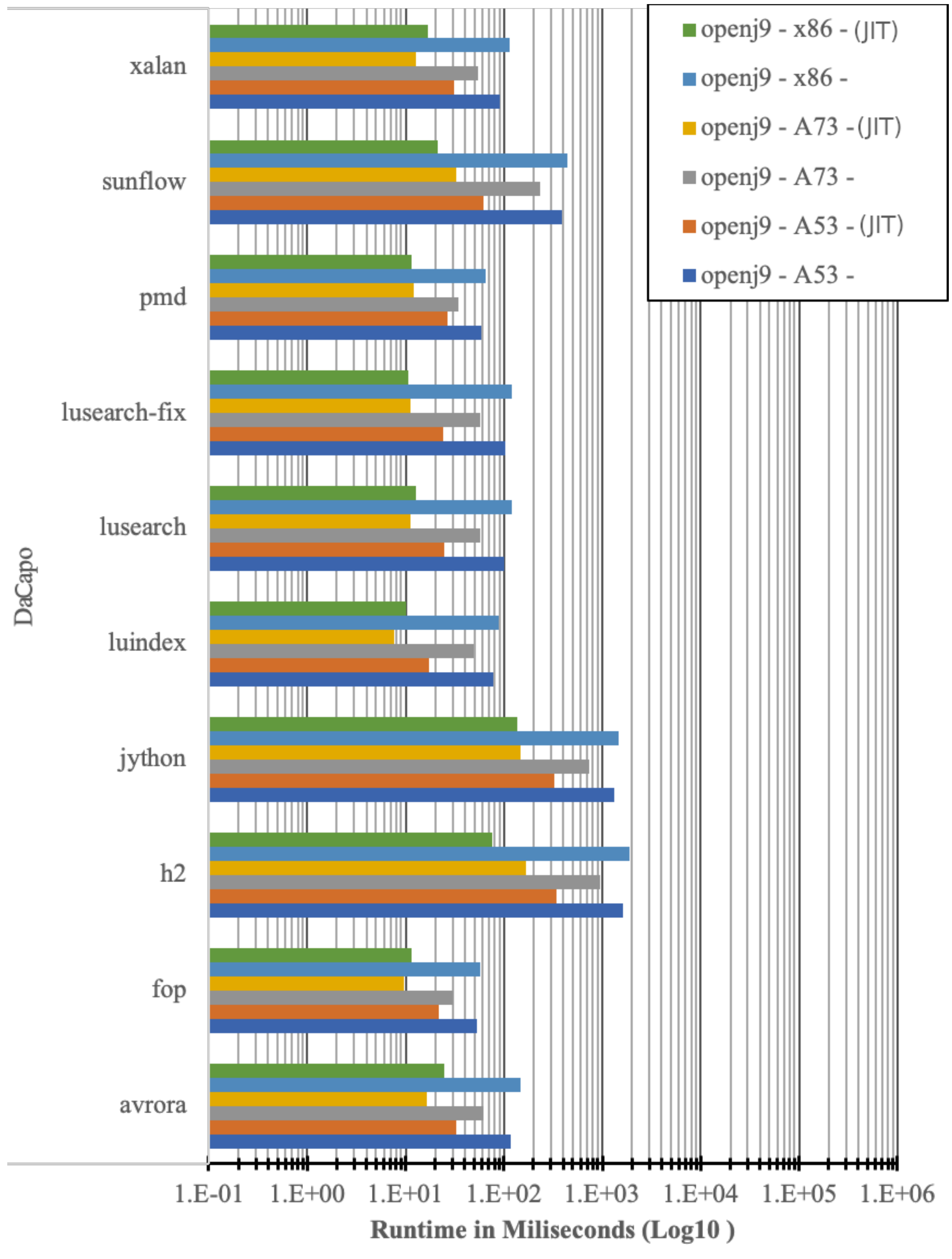

Figure 3.4: DaCapo benchmarks relative speedup for different architectures and different VMs

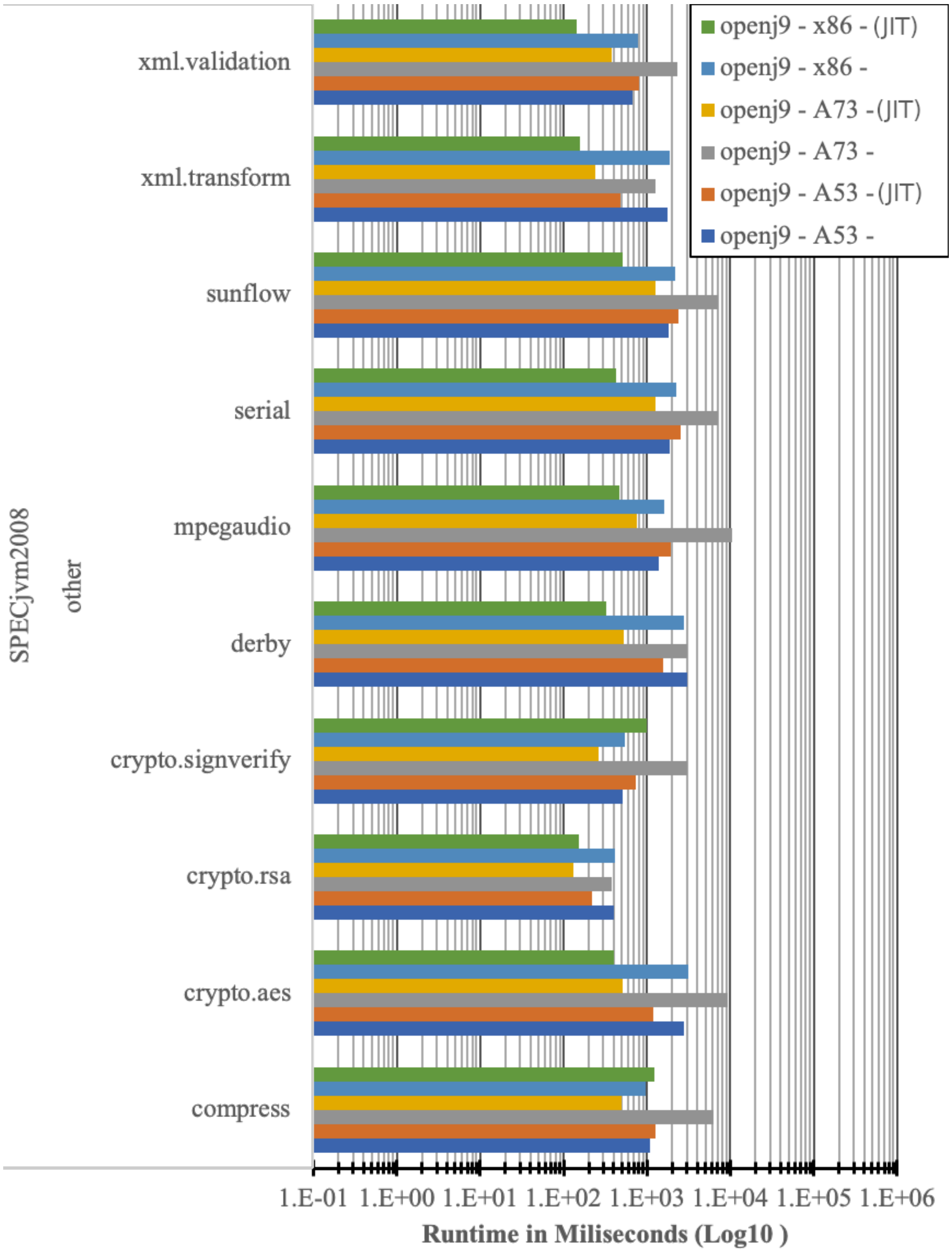

Figure 3.5: SPECjvm®2008 benchmarks relative speedup for different architectures and different VMs

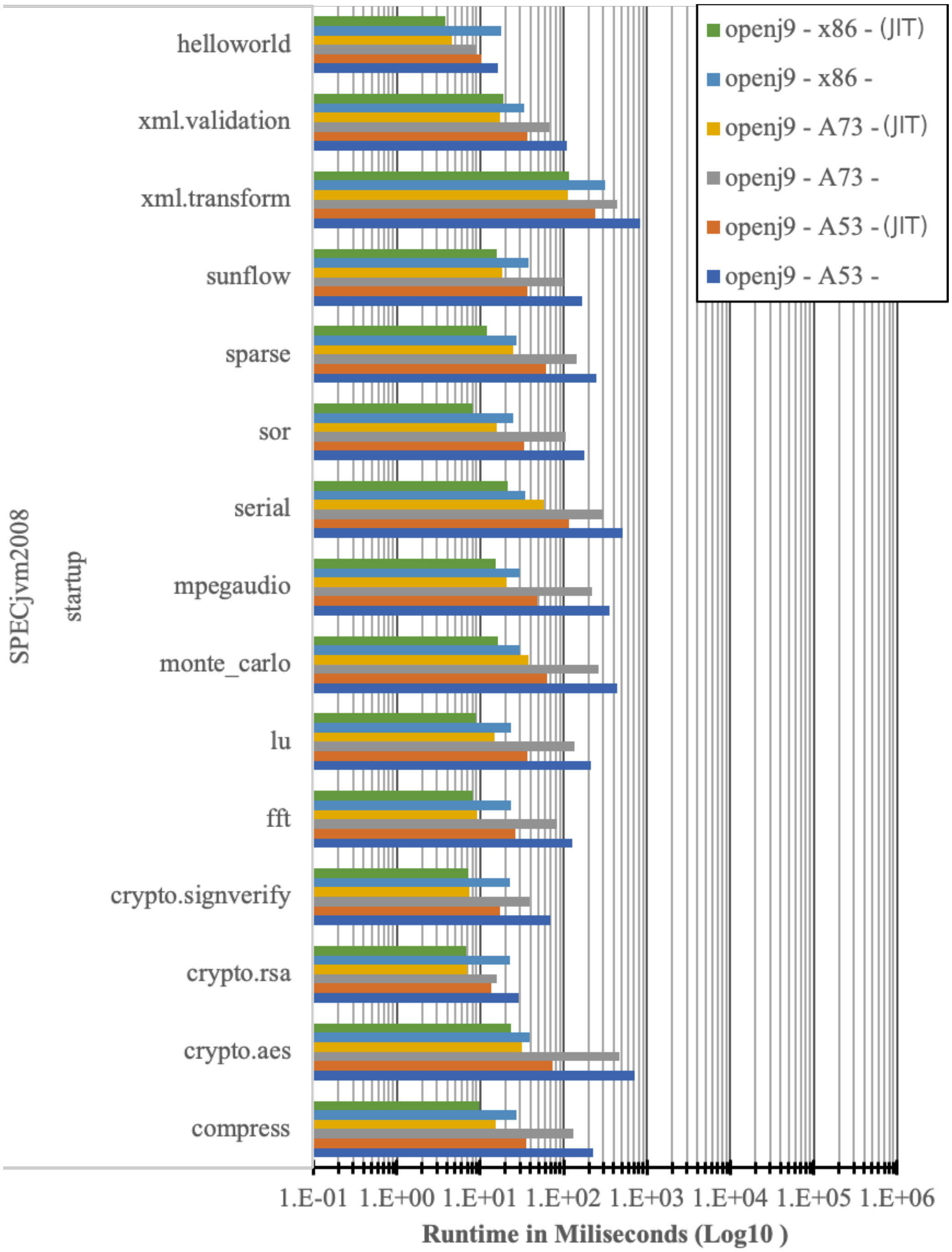

Figure 3.6: SPECjvm®2008 startup benchmarks relative speedup for different architectures and different VMs

forms the AMD64 platform; this is attributed to the high-performance cryptographic hardware available on the A73 SoC and the shared cache. Nonetheless, the performance of the AMD64 platform, the more mature platform, is on average better (Figures 3.5 and 3.6); the A73 hardware follows, with the A53 hardware in last, as expected, since the A53 cores offer lower performance. The JIT outperforms the interpreter in Figures 3.5 and 3.6 since JIT methods are faster once the methods have been executed a number of times above the JVM threshold (warmup).

Due to the maturity of the OpenJ9 AMD64 implementation and the extensive JIT support, performance on AArch64 is expected to be lower since the AArch64 JIT is not yet complete. Using the DaCapo and SPECjvm®2008 benchmarks, the JIT speedup over the interpreter is measured across each benchmark and platform. On average, the A73 platform achieves a 7.35 times speedup, close to the 6.63 times speedup of the AMD64 (Table 3.3). Notably, most implementations have a large variance in speedup, and as such, the average is a very loose metric by which to compare each. The average is used to show that, overall, OpenJ9 for AArch64 performs well within expectations. The A53 platform displays very poor improvements with the JIT across the board, and this is attributed to the very low performance of the A53 core. When the JIT is turned on, the A53 platform slows down dramatically and displays poorer performance because the JIT itself is taking much of the effort. While on longer running benchmarks, the JIT may improve performance, programs on embedded devices can be short-lived, and it is of interest to

Table 3.3: Average speedup of the JIT over the interpreter for SPECjvm®2008 and DaCapo benchmarks.

	A53	A73	AMD64
Average	287%	**735%**	663%
Std Dev	210.81%	397.30%	580.23%

also account for the JIT effort.

The *crypto.aes* benchmark shows improved performance on both AArch64 devices (Table 3.4) since they boast dedicated hardware for *AES* algorithms and other cryptographic instructions [47] available with the Linux kernel. The AArch64 platform on Linux has extensive support for compression, encoding algorithms, e.g., *mpegaudio* and *compress,* due to its shared nature with the Android market [48]. The very erratic performance of SciMark (Table 3.5) is attributed to the discrepancy of the memory between the devices and the incomplete floating-point arithmetic support on OpenJ9 for AArch64; this is further explored in Section 3.3.0.1. DaCapo (Table 3.6) clearly shows that better performance is obtained from the AArch64 platform with better JIT support. On average, AMD64 achieved a $10X$ speedup with the JIT, while the A73 only manages a $5X$ speedup (Table 3.3); this is largely due to the memory-heavy nature of the DaCapo benchmarks with large loads and stores that could be better improved with vectorized loads.

Table 3.4: Execution time speedup of the JIT over the interpreter.

	A53	A73	AMD64	Average	Std Dev
compress	**85%**	**1258%**	**78%**	474%	679%
crypto.aes	230%	**1816%**	**784%**	**943%**	**805%**
crypto.rsa	182%	287%	266%	245%	**56%**
crypto.signverify	**71%**	**1146%**	**54%**	424%	626%
derby	190%	579%	**850%**	540%	332%
mpegaudio	**72%**	**1363%**	348%	594%	680%
serial	**75%**	557%	521%	385%	269%
sunflow	**78%**	555%	428%	354%	247%
xml.transform	358%	514%	**1204%**	692%	450%
xml.validation	**82%**	614%	535%	410%	287%
Average	142%	**869%**	507%	506%	
Std Dev	96.44%	491.97%	359.01%		

Table 3.5: SPECjvm®2008 SciMark benchmark execution time speedup of the JIT over the interpreter.

	A53	A73	AMD64	Average	Std Dev
fft.large	176%	422%	460%	353%	155%
fft.small	**33%**	**1174%**	194%	467%	617%
lu.large	419%	584%	**2540%**	**1181%**	**1180%**
lu.small	**28%**	**1139%**	317%	495%	576%
sor.large	171%	**829%**	**792%**	598%	370%
sor.small	**72%**	**1716%**	284%	691%	**894%**
sparse.large	151%	480%	**1191%**	608%	532%
sparse.small	109%	**1068%**	540%	572%	480%
Average	145%	**926%**	**790%**	620%	
Std Dev	125.30%	435.80%	777.17%		

Table 3.6: DaCapo execution time speedup of the JIT over the interpreter.

	A53	A73	AMD64	Average	Std Dev
avrora	358%	381%	588%	442%	127%
fop	241%	318%	497%	352%	131%
h2	469%	568%	**2438%**	**1158%**	**1109%**
jython	403%	491%	**1060%**	651%	357%
luindex	457%	634%	**874%**	655%	209%
lusearch	412%	505%	**961%**	626%	294%
lusearch-fix	422%	512%	**1133%**	689%	387%
pmd	220%	284%	570%	358%	186%
sunflow	629%	**716%**	**2115%**	**1153%**	**834%**
xalan	294%	433%	672%	466%	191%
Average	391%	484%	**1091%**	655%	
Std Dev	120.72%	135.46%	665.26%		

Table 3.7: SPECjvm®2008 startup benchmark execution time speedup of the JIT over the interpreter.

	A53	A73	AMD64	Average	Std Dev
compress	641%	**868%**	277%	595%	298%
crypto.aes	**959%**	**1492%**	166%	**872%**	667%
crypto.rsa	209%	227%	335%	257%	**68%**
crypto.signverify	407%	522%	310%	413%	106%
fft	488%	**924%**	288%	567%	325%
lu	578%	**886%**	268%	578%	309%
mpegaudio	**723%**	**1058%**	195%	659%	435%
serial	434%	516%	163%	371%	185%
sor	519%	664%	305%	496%	181%
sparse	399%	576%	231%	402%	173%
sunflow	445%	546%	235%	408%	159%
xml.transform	347%	384%	272%	334%	**57%**
xml.validation	297%	390%	182%	290%	105%
helloworld	160%	192%	478%	276%	175%
Average	472%	660%	265%	466%	
St. Dev.	208.82%	354.09%	82.60%		

3.3.0.1 SciMark C and Java

The SciMark benchmarks are further explored to analyze the problems that Figure 3.3 displayed. The Monte-Carlo benchmark execution time is much longer than expected based on the AMD64 execution time. Interestingly enough, the A73 and the A53 platform both struggled with the benchmark even though the A73 is much more performant. The benchmark is executed from its source and its C equivalent to review if the JVM under test is at fault or the hardware. The SciMark benchmarks were rewritten [49, 50] to run each sub-benchmark individually and pass a fixed workload, like SPECjvm®2008 and DaCapo.

Figure 3.7 shows that the VM has issues with the Monte Carlo, SOR, and Sparse benchmarks—referred to as kernels by the SciMark 2.0 suite. Overall, Monte Carlo, SOR, and Sparse are expected to run faster on the A73 hardware as shown with the SciMark C implementation in Figure 3.7, yet they perform poorly; they perform as slowly as their interpreter-only counterpart, revealing that the code has very few JITed methods.

The OpenJDK11 OpenJ9 AArch64 TRJIT is an initial product in an early access release—a minimally viable product (MVP) implementation—but shows great promise. Much work is yet to be done for OpenJ9 and the TRJIT on the AArch64 ISA. Further hardware exploitation is to be done before a mature release and reductions in performance gaps is needed such as in floating-point arithmetic benchmarks.

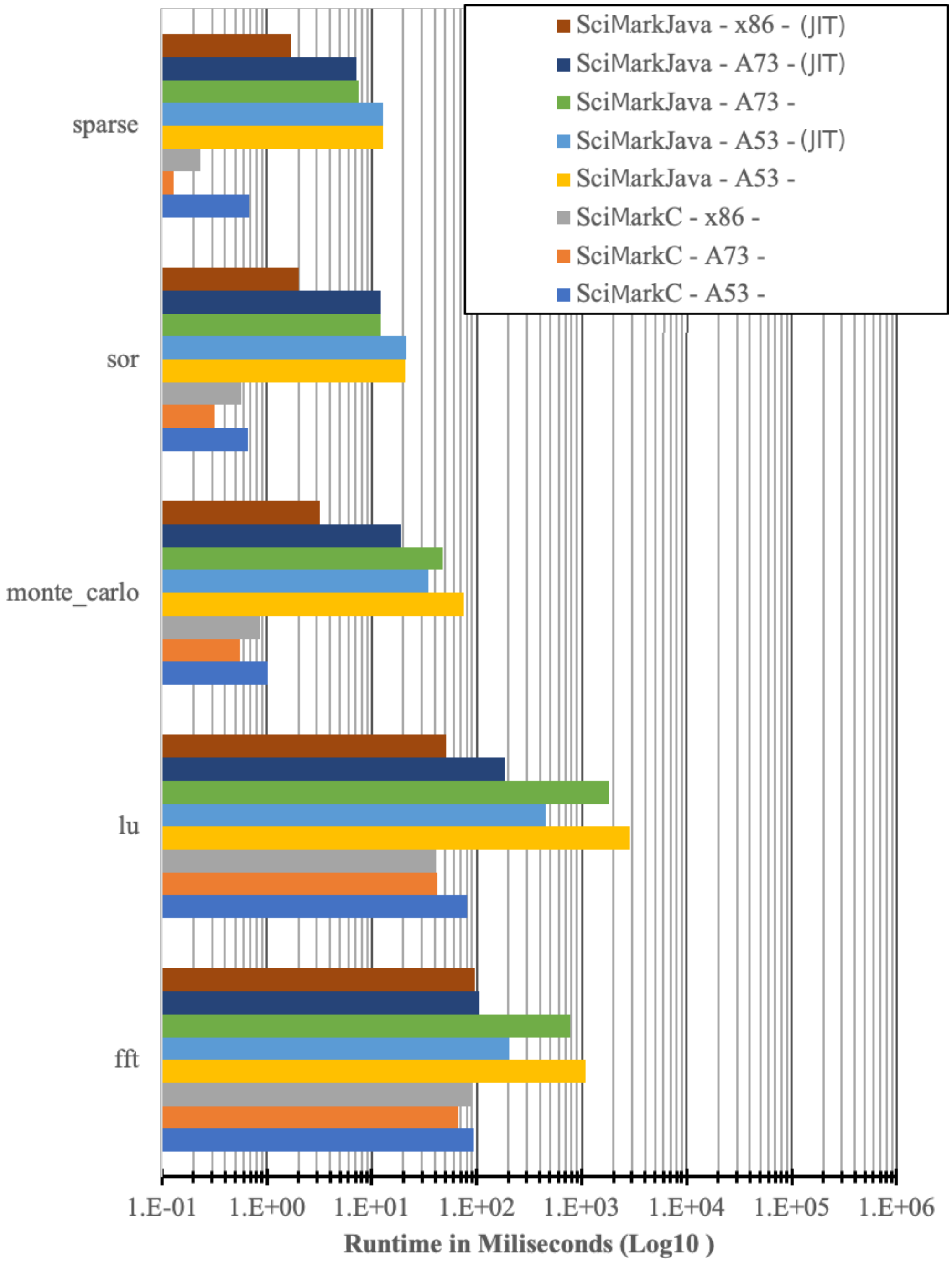

Figure 3.7: SciMark.0 Java vs SciMark.0 C benchmarks for different VM

Chapter 4

AArch64 Usability for Software Engineering

This chapter explores the usability of the AArch64 platform in a software development environment and aims to guide future exploration of AArch64 software. While the experimental results show promise, the current state of the AArch64 software landscape does not lend itself to be a good development platform. The main features required for a proper development device are availability, tracing, build speeds, reproducibility, and reliability. With this in mind, AArch64 poses a few challenges elaborated upon in this chapter.

4.1 Cross Platform Development

To ship software, a developer must build the source code into executable code. The executable code, being platform-dependent, requires the compiler to generate machine code for the target; this makes compiled executables non-portable. The compiler, itself an executable, must run on the system used to build the executable; this leads compilers to have three key variables when they, themselves, are built: build architecture, host architecture, target architecture. The build architecture is the machine used to build the compiler itself, and the host is the machine used to build the executable for the target hardware. When the build machine is not the same as the target machine, a *cross-compiler* is used. With this in mind, there are three main methods to generate executables for a target platform:

- using a cross-compiler built for the host and target hardware.
- using dynamic binary translation to translate the target hardware binary to the host machine code.
- using native hardware hosts that are equivalent to the target hardware.

The defacto standard to build software is to use a native host. The choices in systems that are powerful enough to do development are relatively scarce in the AArch64 landscape. Native hosts provide rapid deployment and ease of use, with the right set of software and operating systems.

Resource-constrained devices are not performant enough and, thus, are ineffective when building large modern software. Cross-compilers solve this problem by offering a means of using any hardware as a host to target a specific ISA. Cross-compilers allow for rapid build speeds when the host provides more performance than the target architecture, but this can also affect the build's robustness.

> "Real-time developers most often use tools hosted on PCs and workstations but target their applications to smaller, less capable computer platforms. This means that they must use cross-compiler tools, which are often more temperamental than the more widely used desktop tools. [...] The lack of sophisticated debugging tools on most small targets complicates testing [51]."

A two-tiered approach, where software is cross-compiled on a host and debugged on the target, is advantageous when the right software is available. The difficulties associated with cross-compilers [51] can be circumvented using a native host.

Dynamic binary translation provides a way to interpret native executable code on a different host [52]. Using dynamic binary translation, binaries can run on a different host and gain debuggability compared to cross-compiled software alone. The translator also allows the installation of dependencies on the host for the target ISA and is more performant than an embedded device. The cost of dynamic binary translation is steep—more than twice as

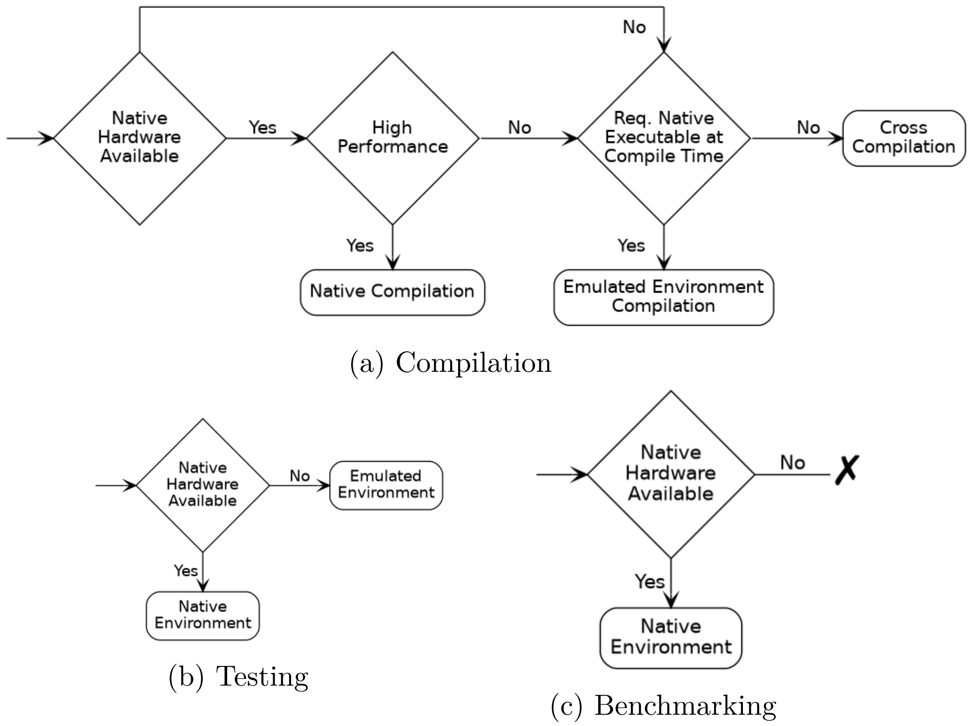

Figure 4.1: Embedded Developement Decision Graphs

slow on equivalent hardware [52]—but it can still offer better performance than resource-constrained devices.

To summarize cross-platform development, three major phases are of importance: building, testing, and benchmarking software. Building software can be done in 3 different approaches (Figure 4.1a): on a native host, using binary translation, or using a cross-compiler. Testing software can be done in 2 ways (Figure 4.1b): on a native host or using binary translation. Benchmarking software can only be done using a native host (Figure 4.1c).

4.2 Debugging and Stack Tracing Software

To identify hot regions of code, developers use tracing programs and frequent snapshots of the call stack to view the amount of time a program is spending on each function (hot regions). Periodic call-stack snapshots are used to gain insight into the hot regions of the OpenJ9 codebase during the JVM runtime. OpenJ9 is tested with SPECjvm®2008 startup.serial and gathers statistics with Linux perf, taking periodic snapshots to reveal long-running or frequently executed functions. The call stack histogram of the software is represented using flame graphs [53]. Flame graphs display the call stack on the Y axis and the timeline on the X axis. Using flame graphs, we can quickly identify long-running functions (hot regions) since they take a more significant portion of the graph. Comparing Figure 4.2 with Figure 4.3 reveals that the class load time affects the startup of AArch64 when compared to AMD64. Looking at the execution time of Figure 4.2 against the original value in Figure 3.6 demonstrates that the Linux perf events cause a slowdown on the AArch64 platform. Deeper exploration reveals that the hardware interrupts for the kernel were not compiled for this platform. Ishii et al. outline the work necessary to achieve better tracing capability on the AArch64 hardware [54] and improve debuggability of the platform.

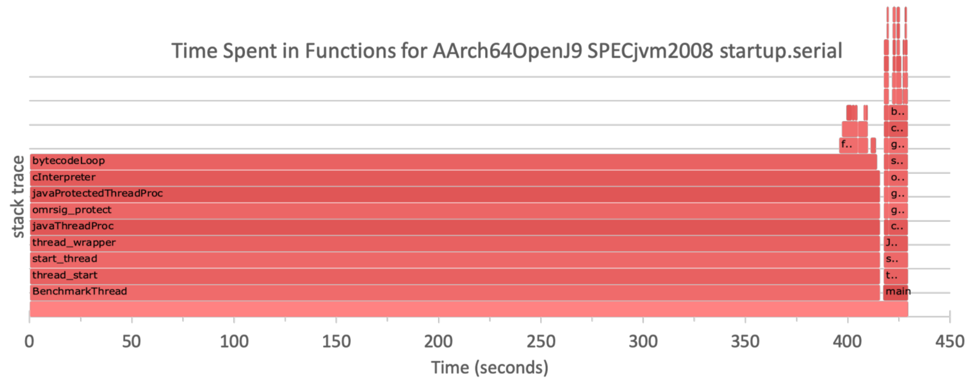

Figure 4.2: Flamegraph representation of the AArch64 OpenJ9 stack trace for the helloworld.startup benchmark from SPECjvm®2008

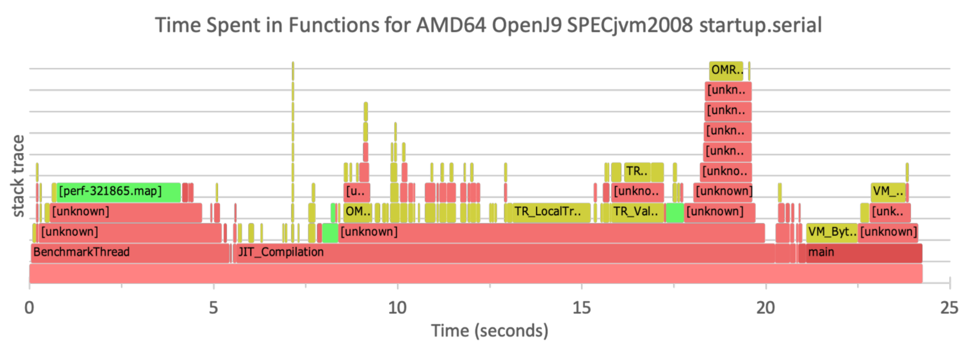

Figure 4.3: Flamegraph representation of the AMD64 OpenJ9 stack trace for the helloworld.startup benchmark from SPECjvm®2008

4.2.1 Benchmarking on AArch64 Embedded Devices

Given that Linux perf is built alongside the kernel [18], forks and patches to the kernel require Linux perf to be built when the kernel changes. Immature embedded devices use forks from the Linux kernel while the vendor is trying to get their patches into the Linux kernel; this means that the kernel must be compiled from its source using patches and binaries for each device. Furthermore, dynamic tracing capabilities using Kprobes (kernel event probes) on AArch64 were not introduced until kernel version 4.21—at the time of writing, the *Debian* stable release (one of the oldest Linux operating systems and the most used basis for other Linux distributions) is running 4.19.

4.2.1.1 Compile the Kernel from Development Source

Kprobes are essential to reduce the number of context switches that Linux perf generates on the system. Kprobes allow buffered CPU code to execute on a breakpoint event rather than a trap. This code seamlessly executes at the return of a function, allowing us to avoid performing a system call. Alongside Kprobes, BPF (Berkley Packet Filter) [55] was set up to leverage the better tracing capability and improve Linux perf performance. As with most components associated with AArch64, the tools are all moving targets. So, to gain the ability to benchmark on low-performance devices, Kprobes are essential, and the kernel must be built from the source when patches or changes are required. While we hoped to achieve better stack traces and improve perf performance we were unable to consume *libbpf* (BPF) with

Linux perf, which uses the dynamic tracer ability. Different tracing frontends were tried with *libbpf* (e.g., Linux perf) but were unsuccessful, since crashes or unavailable libraries prevented benchmarking. Many of the performance tools and development tools often necessary for performance evaluation are tightly coupled with the kernel and the hardware support.

4.2.1.2 Heat Management On Embedded Devices

Embedded devices often overheat due to improper heat management, and it is necessary to provide extra means of cooling to assure functionality. While the device under stress will not display signs of overheating (merely warm to the touch), the packages (the physical silicon devices) are often poor thermal conductors; this can often result in misdiagnosed issues. Frequent "lock-ups" and erratic performance are signs of improper heat management. Heat management problems start with an under-specified integrated circuit package. Embedded devices are marketed as low-powered and general-purpose devices, but the packaging material becomes a problem if used for high-performance computing. The packaging, often made from an epoxy compound, has worse thermal conductivity than bare die packaging [56] as in mainstream AMD64 CPUs (Table 4.1) Modern processors are delivered as ball-grid array (BGA) chip using silver solder balls as contact points as shown in Figure 4.4. To avoid using a heat sink with the devices (*Cu* in Table 4.1), a BGA chip disperses heat to the printed circuit board (PCB) using the *Cu traces.* As seen in Table 4.1, *Cu* traces offer equivalent thermal conductivity as *Cu* on the

Table 4.1: Integrated Circuit Packaging Material Properties [56].

Component	Material	Thermal Cond. (W/mK)
Substrate	BT-Cu Laminate	N/A
Mold Cap	**Epoxy Compound**	**0.8**
Die	**Si**	**100.0**
Die Attach	Silver-Epoxy	1.2
Top Metal (Trace)	Cu (20% Cover.)	75.4
Internal Planes	Cu	377.0
Substrate Core	BT	0.2
Bottom Metal	Cu	377.0
Solder Balls	**Pb-Sn**	**40.0**
Substrate	FR4-CU Laminate	N/A
Top Metal	Cu	377.0

die. The issue arises when the PCB is heat soaked (thermal saturation) from over-utilizing the CPU. Figure 4.5 shows the CPU temperature bleeding onto the PCB affecting other components when heated and their characteristics start to drift when pushed beyond their intended operating temperature. Increasing thermal mass on the die using an external heat-sink with forced airflow (active) is a minimum to alleviate the problem, but, given the traces' thermal conductivity (Figure 4.4), the PCB also requires active cooling.

4.3 Final Thoughts on AArch64

When looking into purchasing a new development board, the reader must look for ongoing support from the manufacturer and a vibrant community. There are numerous boards available at a relatively low price and may sound appealing, but be warned that without "mainline" Linux kernel support (patches

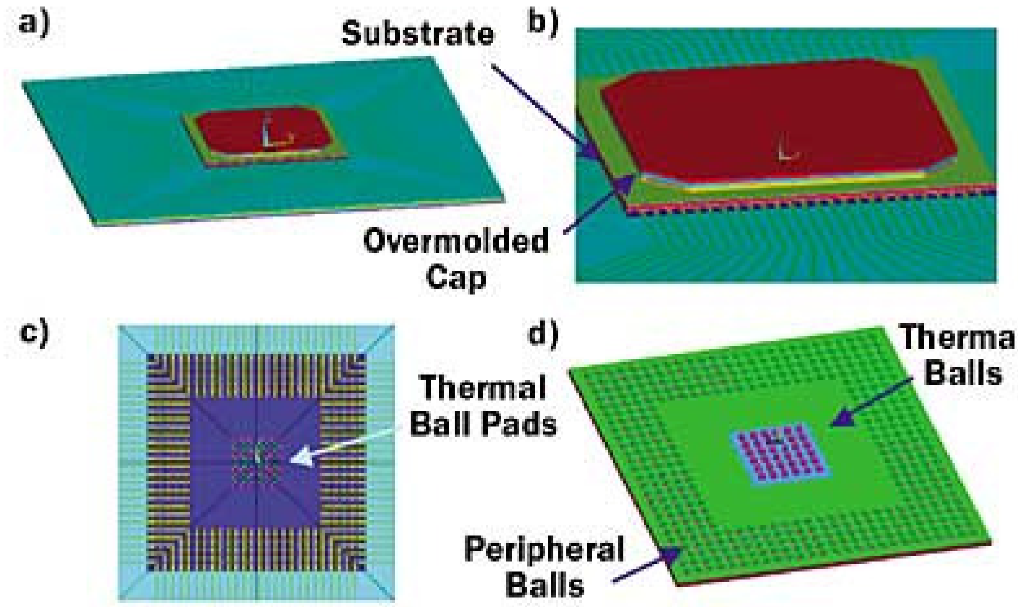

Figure 4.4: Integrated Circuit Packaging Thermal Properties [56].

are already applied on the main code base) for them, they could fall out of, or lack, support. To create a usable build platform, the developer must prioritize devices with extendable and reliable storage support. The device's thermal management must be considered during the requirement and planning phase of a project, and an exposed die should be chosen to improve thermals (Table 4.1).

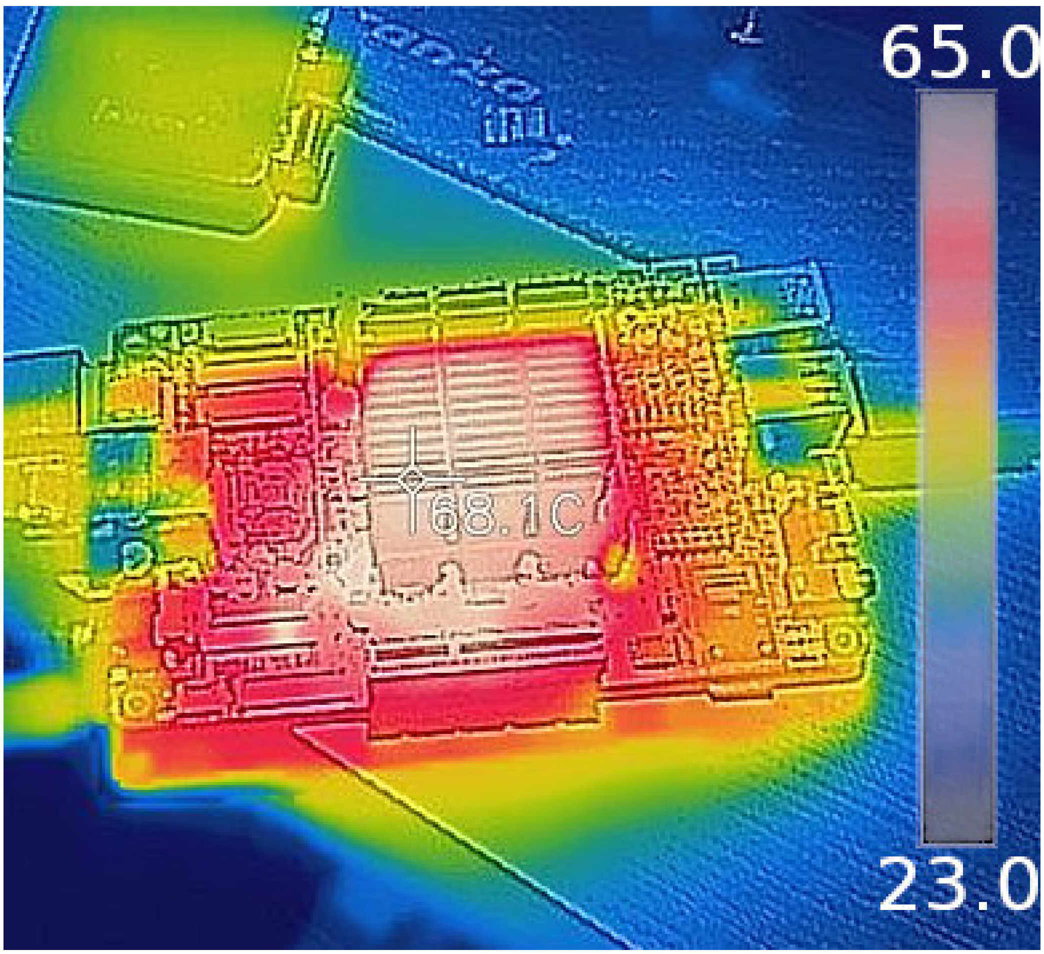

Figure 4.5: Thermal image of Rock64Pro while compiling the Linux kernel [57].

Chapter 5

Conclusion and Future Work

This thesis set out to explore the viability of the AArch64 platform as an environment for the OMR language runtime. It was shown that the A73 core provides comparable or better performance than its AMD64 counterpart; the A53 core is a viable alternative to the A73 when power efficiency is critical. The results provide insight into ARM devices' performance coupled with a high-performance runtime. This provides a basis for comparing, evaluating, and analyzing future implementations. While OMR displays promising results on AArch64, the architecture and its ecosystem are immature. The distribution model of ARM ISAs' explains the rapid growth but also the thinly spread developer efforts on kernel and software support of AArch64. The platform usability is further hindered by the lack of middle-class devices that can perform as a developer platform. Popular AArch64 embedded devices target the "DIY" market, and as such, they suffer from a lack of professional support. Furthermore, the devices often suffer instabilities from improper thermal management and software maturity. Debugging and performance evaluation on AArch64 is more challenging than on its AMD64 counterpart, but the recent influx of the AArch64 ISA will resolve these problems. On June 22, 2020, Apple announced it would deprecate the usage of AMD64 in favour of AArch64 on all their Mac computers [58]. On September 13, 2020, NVIDIA announced it would acquire ARM Holdings "continu[ing] ARM's open-licensing model, customer neutrality, and expand ARM's IP licensing portfolio with NVIDIA technology" [59]. With large companies using the ISA, overall adoption and support is expected to increase in the near future. This thesis has shown that the AArch64 platform lends itself as a viable platform for the OMR language runtime. While current development work on the platform can be tedious due to the underdeveloped software landscape, increased interest of major vendors will improve its usability over time and push the envelope.

5.1 Future Work

Many unimplemented optimizations are available for OMR on the AArch64 platform and are of interest. More support for floating-point in the JIT is necessary to bring the performance closer to the expected results shown in Figure 3.7. Future work on OpenJ9 must focus on improving floating support, adding ARM NEON [34] support, and enabling vectorized floating-point operations. The ARM Advanced SIMD capabilities are unexploited in the VM, and vectorized/scalar support can significantly improve performance. The SOR and Sparse benchmarks are matrix heavy [49], and having vectorized operations will improve the execution speed of these types of benchmarks. Exploiting vectorized loads and stores could vastly improve data movement speeds. One possible avenue that could be of great interest is to exploit the onboard GPU. GPUs can do vector-heavy benchmarks faster than CPUs, as shown by Dobravec et al. [60]. Furthermore, the AArch64 ISA offers cryptographic support, improving OMR performance when used in security-sensitive applications. Exploiting big.LITTLE architectures can further improve OMR power usage and overall performance. Having OMR aware of the heterogeneous cores available will be invaluable as it allows prioritizing program threads to the big cores and the background tasks—garbage collector threads, busy-wait—to the little cores. For power-constrained devices, the Thumb32 (T32) instruction set will be of interest. T32 is a compressed instruction set (heterogeneous 16-bit and 32-bit instructions) available in the AArch32 state of the ARMv8 ISA. The ARMv8 cores in AArch32 (32-bit) mode "decompress" the T32 instructions on the fly, thus reducing binary size, cache usage, and improving performance.

With AArch64 expansion in the middle-class device space, given major vendors' interest, OMR is an excellent platform for performance-driven research having its baseline now in place.

Bibliography

[1] Inc MIPS Technologies. *MIPS IV Instruction Set (Revision 3.2)*. Sept. 1, 1995. URL: https://www.mips.com.

[2] Krste Waterman Andrew; Asanović. *The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA version 2.2. University of California, Berkeley. EECS-2016-118*. May 25, 2017. URL: https://riscv.org.

[3] *ARM*. Sept. 19, 2019. URL: https://www.arm.com/.

[4] *Eclipse OpenJ9*. Sept. 19, 2019. URL: https://www.eclipse.org/openj9/.

[5] *Eclipse OMR*. URL: https://www.eclipse.org/omr/.

[6] Oracle. *Java Developer's Guide Release 19*. Apr. 5, 2020. URL: https://docs.oracle.com/en/database/oracle/oracle-database/19/jjdev/Oracle-JVM-JIT.html#GUID-9466BE4E-E7EE-486F-9DF8-//D331B316359D.

[7] Feng Qian and Laurie Hendren. "A Study of Type Analysis for Speculative Method Inlining in a JIT Environment". In: *Compiler Construction*. Ed. by Rastislav Bodik. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 255–270. ISBN: 978-3-540-31985-6.

[8] Haoyu Li, Mingyu Wu, and Haibo Chen. "Analysis and Optimizations of Java Full Garbage Collection". In: *Proceedings of the 9th Asia-Pacific Workshop on Systems*. APSys '18. Jeju Island, Republic of Korea: Association for Computing Machinery, 2018. ISBN: 9781450360067. DOI: 10.1145/3265723.3265735. URL: https://doi.org/10.1145/3265723.3265735.

[10] *Eclipse OpenJ9*. Sept. 19, 2019. URL: https://github.com/eclipse/openj9.

[11] Samer AL Masri, Sarah Nadi, Matthew Gaudet, Xiaoli Liang, and Robert W. Young. "Using Static Analysis to Support Variability Implementation Decisions in C++". In: *Proceedings of the 22nd International Systems and Software Product Line Conference - Volume 1*. SPLC '18. Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 236–245. ISBN: 9781450364645. DOI: 10.1145/3233027.3233043. URL: https://doi.org/10.1145/3233027.3233043.

[12] *OMR+Ruby*. Sept. 19, 2019. URL: https://github.com/rubyomr-preview/rbjitglue.

[13] *lua vermelha*. Sept. 19, 2019. URL: https://github.com/Leonardo2718/lua-vermelha.

[14] *Eclipse OMR*. Sept. 19, 2019. URL: https://github.com/eclipse/omr.

[15] osdevi. *introduction the the kernel*. July 6, 2020. URL: https://wiki.osdev.org.

[16] Linux Kernel. *Linux*. May 25, 2020. URL: https://www.kernel.org.

[17] Matt Welsh, Mathhias Kalle Dalheimer, and Lar Kaufman. *Running Linux (3rd Ed.)* USA: O'Reilly & Associates, Inc., 1999. ISBN: 156592469X.

[18] Linux Kernel. *perf*. May 25, 2020. URL: https://github.com/torvalds/linux/tree/master/tools/perf.

[19] Daniel Yokoyama, Bruno Schulze, Fabio Borges, and Giacomo Mc Evoy. "The survey on ARM processors for HPC". In: *The Journal of Supercomputing* (June 2019). ISSN: 1573-0484. DOI: 10.1007/s11227-019-02911-9. URL: https://doi.org/10.1007/s11227-019-02911-9.

[20] ARM Holdings. *Processors*. July 13, 2020. URL: https://developer.arm.com/ip-products/processors.

[21] nullr0ute. *The state of open source accelerated graphics on ARM devices*. July 13, 2020. URL: https://nullr0ute.com/2017/09/the-

`state-of-open-source-%5C%5Caccelerated-graphics-on-arm-devices`.

[22] Qemu. *What is qemu?* May 25, 2020. URL: `https://www.qemu.org`.

[23] Linux Kernel. *binfmt-misc*. May 25, 2020. URL: `https://www.kernel.org/doc/html/latest/admin-guide/binfmt-misc.html`.

[24] Dirk Merkel. "Docker: lightweight linux containers for consistent development and deployment". In: *Linux journal* 2014 (2014), no. 239, p. 2.

[25] Aleksandar Prokopec, Andrea Rosa, David Leopoldseder, Gilles Duboscq, Petr Tuma, Martin Studener, Lubomir Bulej, Yudi Zheng, Alex Villazon, 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.

[26] *Java Grande*. June 12, 2020. URL: `http://www.javagrande.org`.

[27] Brendon Cahoon and Kathryn S McKinley. "Data flow analysis for software prefetching linked data structures in Java". In: *Proceedings 2001 International Conference on Parallel Architectures and Compilation Techniques*. IEEE. 2001, pp. 280–291.

[28] Bruno Dufour, Karel Driesen, Laurie Hendren, and Clark Verbrugge. "Dynamic metrics for Java". In: *Proceedings of the 18th annual ACM*

SIGPLAN conference on Object-oriented programing, systems, languages, and applications. 2003, pp. 149–168.

[29] SPEC. *SPECjvm® 2008*. June 12, 2020. URL: `https://www.spec.org/jvm2008`.

[30] SPEC. *SPECjbb® 2000*. June 12, 2020. URL: `https://www.spec.org/jbb2000`.

[31] 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.

[32] Gilles Duboscq. "Combining Speculative Optimizations with Flexible Scheduling of Side-effects". PhD thesis. Ph. D. Dissertation. Johannes Kepler University, Linz, 2016.

[33] Math nist. *SciMark*. July 11, 2020. URL: `https://math.nist.gov/scimark2/about.html`.

[34] ARM. *NEON*. June 11, 2020. URL: `https://www.arm.com/why-arm/technologies/neon`.

[35] Jean-Philippe Legault, Aaron Graham, and Md Mahbubur Rahman. *AArch64 Encoding*. July 11, 2020. URL: https://github.com/CAS-Atlantic/AArch64-Encoding.

[36] Eclipse OMR. *Portability library*. July 11, 2020. URL: https://www.eclipse.org/omr/reference/port.html.

[37] Travis CI GmbH. *Travis CI*. July 11, 2020. URL: https://travis-ci.org/.

[38] Eclipse OMR. *Tril tests*. July 13, 2020. URL: https://github.com/eclipse/omr/tree/master/fvtest/tril#tril.

[39] Jean-Philippe Legault. *aarch64 vm benchmark*. July 11, 2020. URL: https://github.com/jeanlego/aarch64_omr_benchmark.

[40] Jean-Philippe Legault. *benchmark parser*. July 11, 2020. URL: https://github.com/jeanlego/bench_parser.

[41] 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.

[42] SPEC. *SPECjvm®2008 User's Guide*. June 12, 2020. URL: https://www.spec.org/jvm2008/docs/UserGuide.html.

[43] AdoptJDK. *Openjdk11 for AArch64 Nightly build*. Oct. 16, 2019. URL: https://github.com/AdoptOpenJDK/openjdk11-binaries/releases/download/jdk11u-2019-10-16-10-43/OpenJDK11U-jdk_aarch64_linux_openj9_2019-10-16-10-43.tar.gz.

[44] Oracle. *Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide*. July 12, 2020. URL: https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/collectors.html.

[45] IBM Corporation. *-Xgcpolicy*. July 11, 2020. URL: https://www.ibm.com/support/knowledgecenter/SSYKE2_8.0.0/openj9/xgcpolicy/index.html.

[46] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tuma, 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.

[47] *Accelerated AES for the Arm64 Linux kernel*. Sept. 19, 2019. URL: https://www.linaro.org/blog/accelerated-aes-for-the-arm64-linux-kernel/.

[48] *Is there any Mali-V61/Mali-V550 board out there*. Sept. 19, 2019. URL: https://community.arm.com/developer/tools-software/

graphics/f/discussions/8616/is-there-any-mali-v61-mali-v550-board-out-there.

[49] Jean-Philippe Legault. *SciMarkC*. July 11, 2020. URL: https://github.com/jeanlego/scimarkC.

[50] Jean-Philippe Legault. *SciMarkJava*. July 11, 2020. URL: https://github.com/jeanlego/scimarkJava.

[51] B.P. Douglass. *Doing Hard Time: Developing Real-time Systems with UML, Objects, Frameworks, and Patterns*. Addison-Wesley object (3 1/2 in.)series. Addison-Wesley, 1999. ISBN: 9780201498370. URL: https://books.google.ca/books?id=xoVXALii7o0C.

[53] Brendan Gregg. "The Flame Graph". In: *Commun. ACM* 59 (May 2016), no. 6, pp. 48–57. ISSN: 0001-0782. DOI: 10.1145/2909476. URL: https://doi.org/10.1145/2909476.

[54] Hiroyuki ISHII. "Dynamic Tracing Tools on ARM/AArch64 platform Updates and Challenges". In: (Embedded Linux Conference). Panasonic Corporation, Feb. 21, 2017.

[55] Steven McCanne and Van Jacobson. "The BSD Packet Filter: A New Architecture for User-level Packet Capture." In: *USENIX winter*. Vol. 46. 1993.

[56] Bruce Guenin. *A Funny Thing Happened On The Way To The Heatsink*. July 2019. URL: https://www.electronics-

cooling.com/2005/08/a-funny-thing-happened-on-the-way-to-the-heatsink/.

[57] Piotr Maliński. *Are cheap low power x86 CPUs an alternative to higher end makers ARM boards?* Sept. 29, 2020. URL: https://rk.edu.pl/en/are-cheap-low-power-x86-cpus-n-alternative-higher-end-makers-arm-boards/.

[58] Apple. *Apple announces Mac transition to Apple silicon*. July 22, 2020. URL: https://www.apple.com/newsroom/2020/06/apple-announces-mac-transition-to-apple-silicon/.

[59] NVIDIA. *NVIDIA to Acquire Arm for 40 Billion, Creating World's Premier Computing Company for the Age of AI*. Sept. 13, 2020. URL: https://nvidianews.nvidia.com/news/nvidia-to-acquire-arm-for-40-billion-creating-worlds-premier-computing-company-for-the-age-of-ai.

[60] Tomaž Dobravec and Patricio Bulić. "Comparing CPU and GPU implementations of a simple matrix multiplication algorithm". In: *Int. Jour. of Computer and Electrical Engineering* 9 (2017), no. 2, pp. 430–438.

[61] Jean-Philippe Legault. *XDocker*. July 11, 2020. URL: https://github.com/CAS-Atlantic/xdocker.

[62] Bastidas F Andrés and María Pérez. "Transpiler-based architecture for multi-platform web applications". In: *2017 IEEE Second Ecuador Technical Chapters Meeting (ETCM)*. IEEE. 2017, pp. 1–6.

[63] Hiroyuki ISHII. *Dynamic Tracing Tools on ARM/AArch64 platform*. Apr. 5, 2020. URL: `https://elinux.org/images/3/32/ELC_2017_NA_dynamic_tracing_tools_on_arm_aarch64_platform.pdf`.

[64] Matthew Arnold, Stephen J Fink, David Grove, Michael Hind, and Peter F Sweeney. "A survey of adaptive optimization in virtual machines". In: *Proceedings of the IEEE* 93 (2005), no. 2, pp. 449–466.

[65] Fabrice Bellard. "QEMU, a Fast and Portable Dynamic Translator". In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATEC '05. Anaheim, CA: USENIX Association, 2005, pp. 41–41. URL: `http://dl.acm.org/citation.cfm?id=1247360.1247401`.

[66] The Computer Language Benchmarks Game. *How many times slower?* Apr. 5, 2020. URL: `https://benchmarksgame-team.pages.debian.net/benchmarksgame/which-programs-are-fastest.html`.

[67] ARM. "Processing Architecture for Power Efficiency and Performance". In: (May 25, 2020). URL: `https://www.arm.com/why-arm/technologies/big-little`.

[68] D. Patterson. "Reduced Instruction Set Computers Then and Now". In: *Computer* 50 (Dec. 2017), no. 12, pp. 10–12. ISSN: 1558-0814. DOI: 10.1109/MC.2017.4451206.

[69] gnu-binutils. *fd3619828e94a24a92cddec42cbc0ab33352eeb4*. Apr. 5, 2020. URL: https://sourceware.org/git/gitweb.cgi?p=binutils-gdb.git;a=commitdiff;h=fd3619828e94a24a92cddec42cbc0ab33352eeb4.

[70] carlosedp. Sept. 19, 2019. URL: https://medium.com/@carlosedp/java-benchmarks-on-arm64-17edd8b9ff79.

[71] *BumbleBench-and-HumbleBench*. Sept. 19, 2019. URL: https://github.com/AdoptOpenJDK/bumblebench/wiki/BumbleBench-and-HumbleBench.

[72] 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.

[73] Arnaldo Carvalho De Melo. "The new linux'perf'tools". In: *Slides from Linux Kongress*. Vol. 18. 2010, pp. 1–42.

[74] 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.

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

[76] Ahmed Imteaj, Tanveer Rahman, Muhammad Kamrul Hossain, Mohammed Shamsul Alam, and Saad Ahmad Rahat. "An IoT based fire alarming and authentication system for workhouse using Raspberry Pi 3". In: *2017 International conference on electrical, computer and communication engineering (ECCE)*. IEEE. 2017, pp. 899–904.

[77] Jean-Philippe Legault. *j9builder*. July 11, 2020. URL: `https://github.com/jeanlego/j9_build`.

[78] *jmh*. Sept. 19, 2019. URL: `https://openjdk.java.net/projects/code-tools/jmh/`.

[79] linaro. *kprobes-event-tracing-armv8*. Apr. 5, 2020. URL: `https://www.linaro.org/blog/kprobes-event-tracing-armv8/`.

[80] 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.

[81] Linaro. *GCC toolchain*. July 11, 2020. URL: `https://www.linaro.org/about/`.

[82] David Lion, Adrian Chiu, Hailong Sun, Xin Zhuang, Nikola Grcevski, and Ding Yuan. "Don't get caught in the cold, warm-up your {JVM}: Understand and eliminate {JVM} warm-up overhead in data-parallel systems". In: *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 2016, pp. 383–400.

[83] Pejman Lotfi-Kamran, Boris Grot, Michael Ferdman, Stavros Volos, Onur Kocberber, Javier Picorel, Almutaz Adileh, Djordje Jevdjic, Sachin Idgunji, Emre Ozer, et al. "Scale-out processors". In: *ACM SIGARCH Computer Architecture News* 40 (2012), no. 3, pp. 500–511.

[84] Mirjana Maksimović, Vladimir Vujović, Nikola Davidović, Vladimir Milošević, and Branko Perišić. "Raspberry Pi as Internet of things hardware: performances and constraints". In: *design issues* 3 (2014), no. 8.

[85] Marius Pirvu. *Optimize JVMstart-up with Eclipse OpenJ9*. June 2, 2020. URL: `https://developer.ibm.com/articles/optimize-jvm-startup-with-eclipse-openjj9/`.

[86] Daniel McDermott. "Evaluating a Cluster of Low-Power ARM64 Single-Board Computers with MapReduce". In: (2018).

[87] Roberto Metere, Andreas Lindner, and Roberto Guanciale. "Sound transpilation from binary to machine-independent code". In: *Brazilian Symposium on Formal Methods*. Springer. 2017, pp. 197–214.

[88] Zhonghong Ou, Bo Pang, Yang Deng, Jukka K Nurminen, Antti Ylä-Jääski, and Pan Hui. "Energy-and cost-efficiency analysis of arm-based clusters". In: *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*. IEEE. 2012, pp. 115–123.

[90] David Patterson. "Reduced Instruction Set Computers Then and Now". In: *Computer* 50 (2017), no. 12, pp. 10–12.

[91] perf-tools. *perf-tools*. July 12, 2020. URL: `https://github.com/brendangregg/perf-tools`.

[92] Qemu. *User mode emulation*. May 25, 2020. URL: `https://www.qemu.org/docs/master/user/index.html`.

[93] Qualcomm. Feb. 10, 2020. URL: `https://www.qualcomm.com/media/documents/files/qualcomm-centriq-2400-processor.pdf,%20pp.%2015`.

[94] Pine 64. *Rock 64*. July 13, 2020. URL: `https://www.pine64.org/devices/single-board-computers/rock64/`.

[95] Hwajeong Seo, Ilwoong Jeong, Jungkeun Lee, and Woo-Hwan Kim. "Compact implementations of ARX-based block ciphers on IoT processors". In: *ACM Transactions on Embedded Computing Systems (TECS)* 17 (2018), no. 3, pp. 1–16.

[96] Christopher Small and Margo I Seltzer. “A Comparison of OS Extension Technologies.” In: *USENIX Annual Technical Conference*. 1996, pp. 41–54.

[97] *toolchain-construction*. Sept. 19, 2019. URL: https://crosstool-ng.github.io/docs/toolchain-construction/.

[98] Samer AL Masri, Sarah Nadi, Matthew Gaudet, Xiaoli Liang, and Robert W. Young. “Using Static Analysis to Support Variability Implementation Decisions in C++”. In: *Proceedings of the 22nd International Systems and Software Product Line Conference - Volume 1*. SPLC ’18. Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 236–245. ISBN: 9781450364645. DOI: 10.1145/3233027.3233043. URL: https://doi.org/10.1145/3233027.3233043.

[99] 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.

Appendix 1

XDocker

XDocker [61] allows the user to seamlessly leverage existing dockerfiles in their infrastructure for different platforms on any host supported by Docker. XDocker is short for cross-platform Docker, it allows any to any platform compilation using Docker [24], binfmt [23], and userland QEMU [22]. XDocker officially supports macOS, Linux, and Windows using Windows subsystem for Linux (WSL). XDocker specializes the users' dockerfile in three phases. The first phase specializes the dockerfile for the host-target platform. The second phase specializes the dockerfile for the user to keep file permissions intact. The third phase specializes the dockerfile for the directory such that the path between the host and the container is the same, allowing seamless transition. Using a three-phase build grants XDocker the ability to reuse the base container if it was generated by another user and shares the intermediate builds; this allows XDocker to boot faster and use less space. XDocker provides the same advantages as Docker [24]: sandboxing, reproducibility, and portability. XDocker improves the cross-platform support of Docker using QEMU as a user-space emulator (Section 2.4.3).

Appendix 2

HotSpot Benchmarks

Other benchmarks were run earlier during experimentation and OpenJ9 on AArch64 was compared to HotSpot on AArch64. Including HotSpot did not provide further insight, and, as such, are simply omitted from the main body. The following results are left here for posterity.

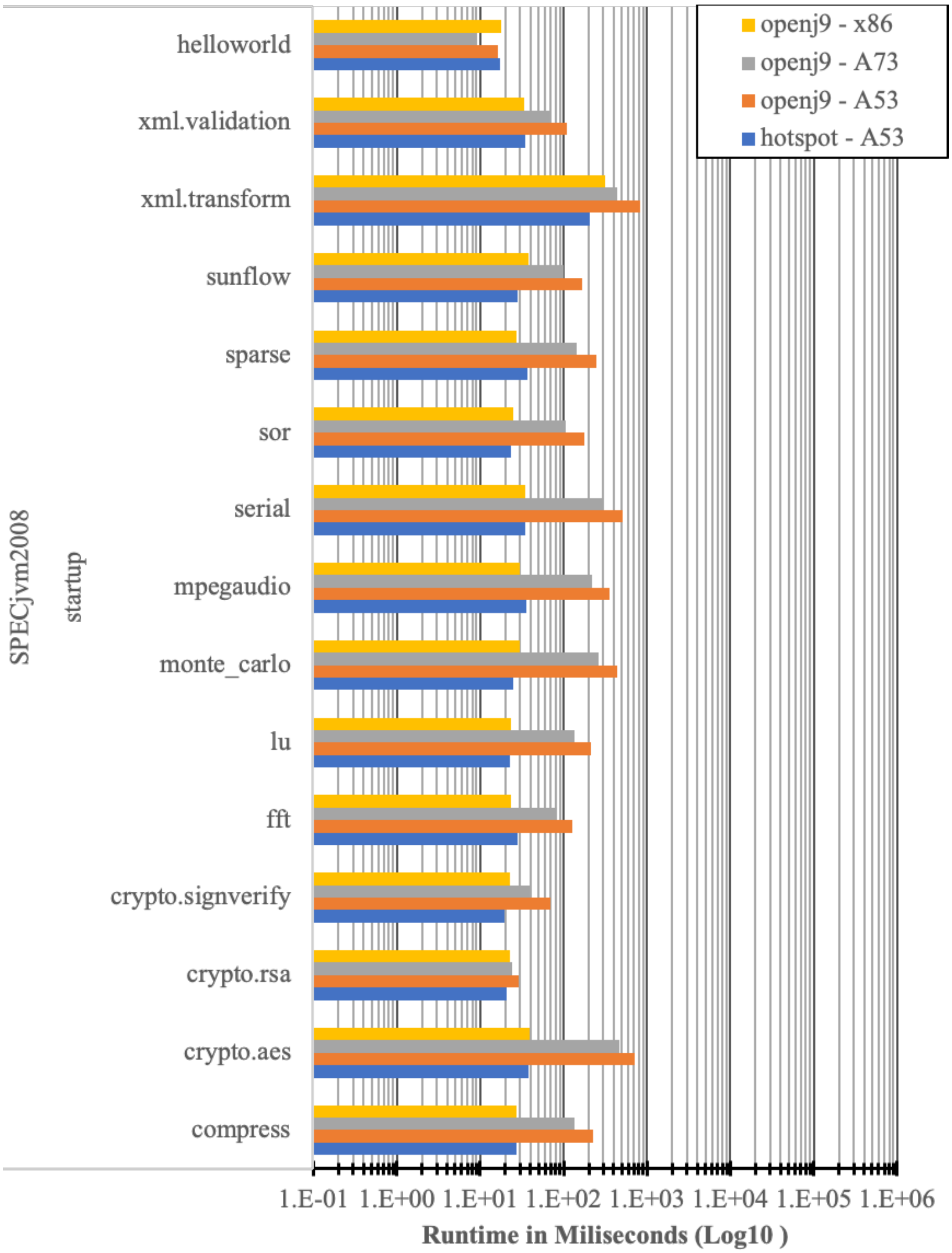

Figure 5.1: SPECjvm®2008 startup benchmarks execution times using perf

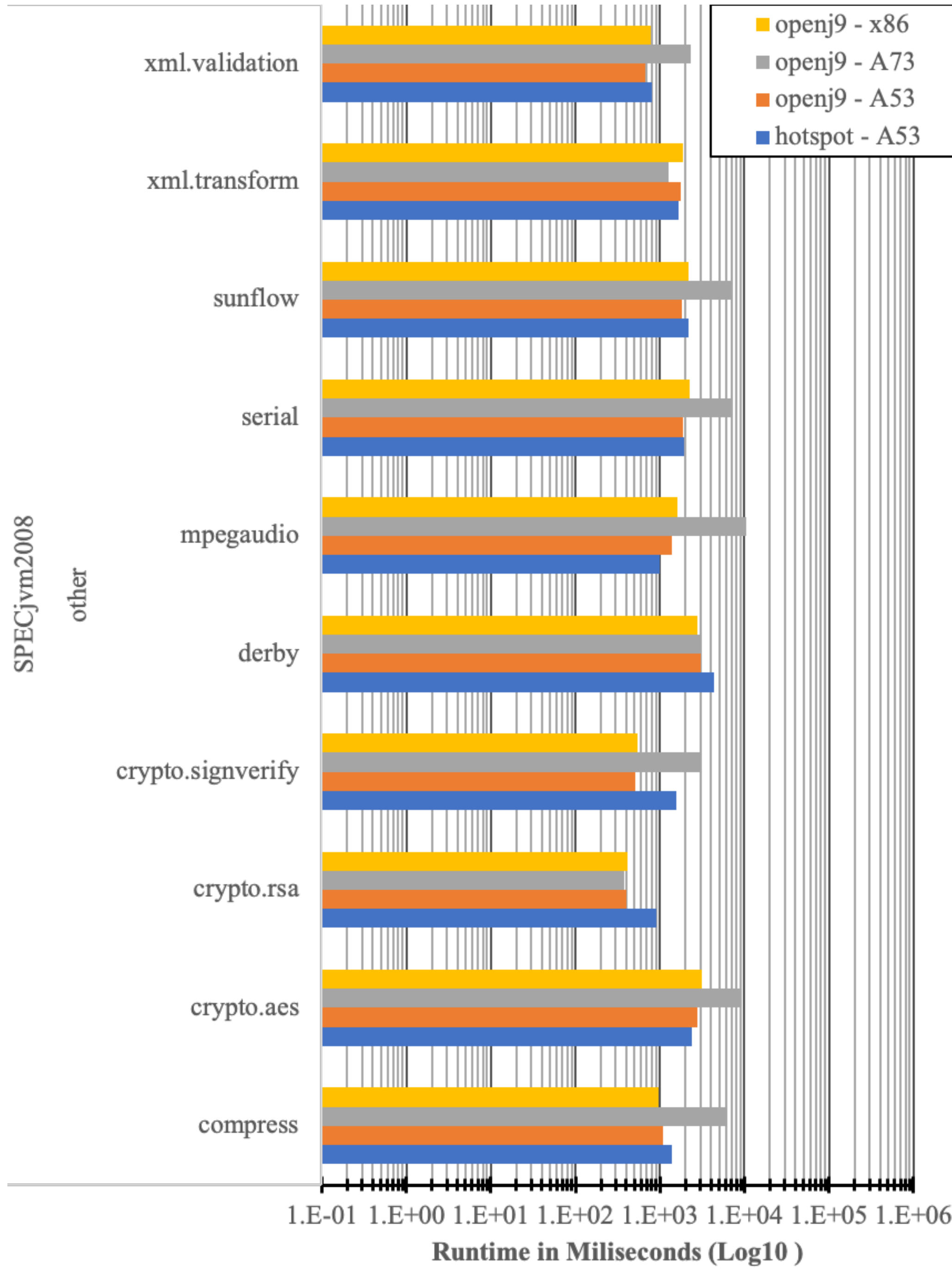

Figure 5.2: SPECjvm®2008 benchmarks execution times using perf

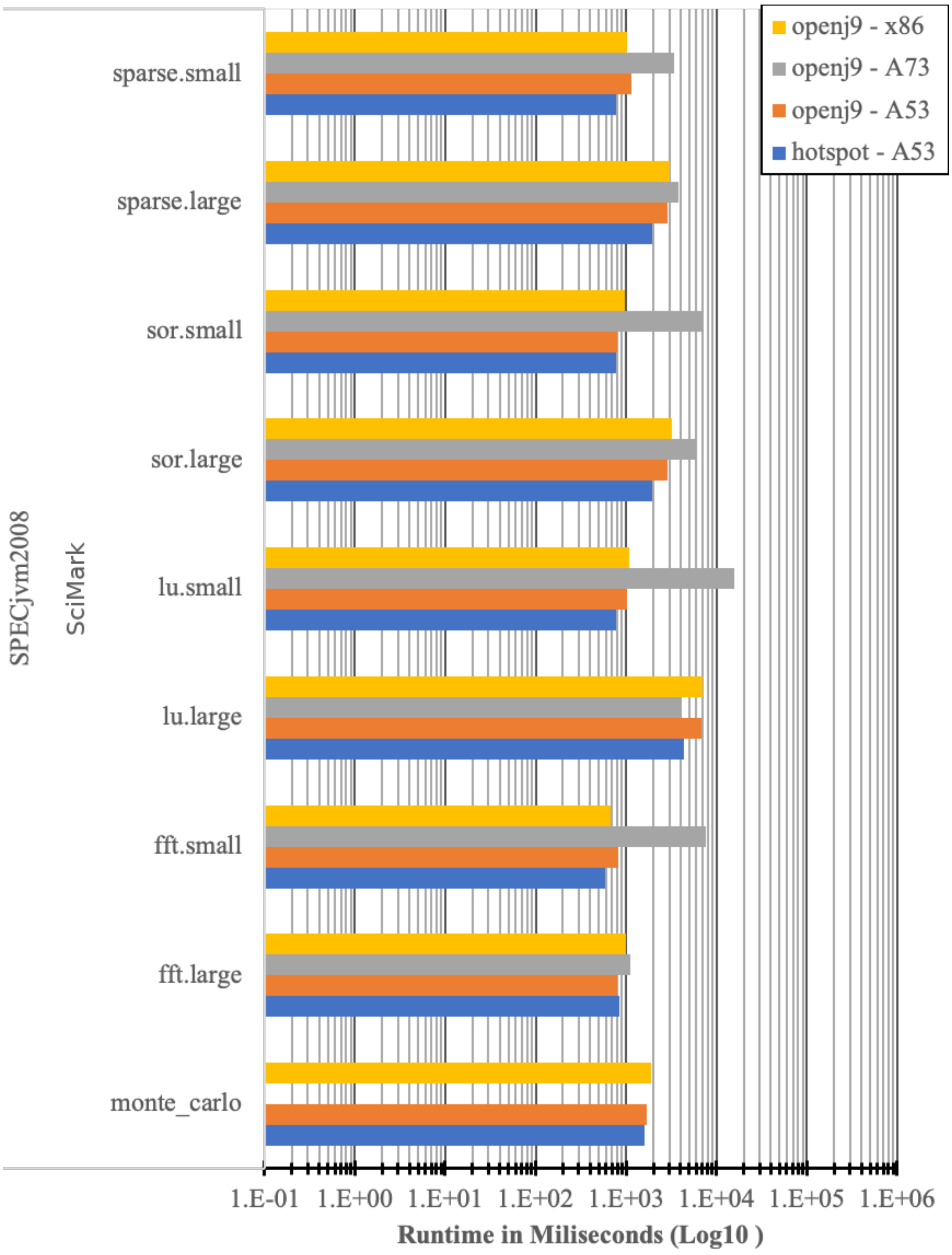

Figure 5.3: SPECjvm®2008 SciMark benchmarks execution times using perf

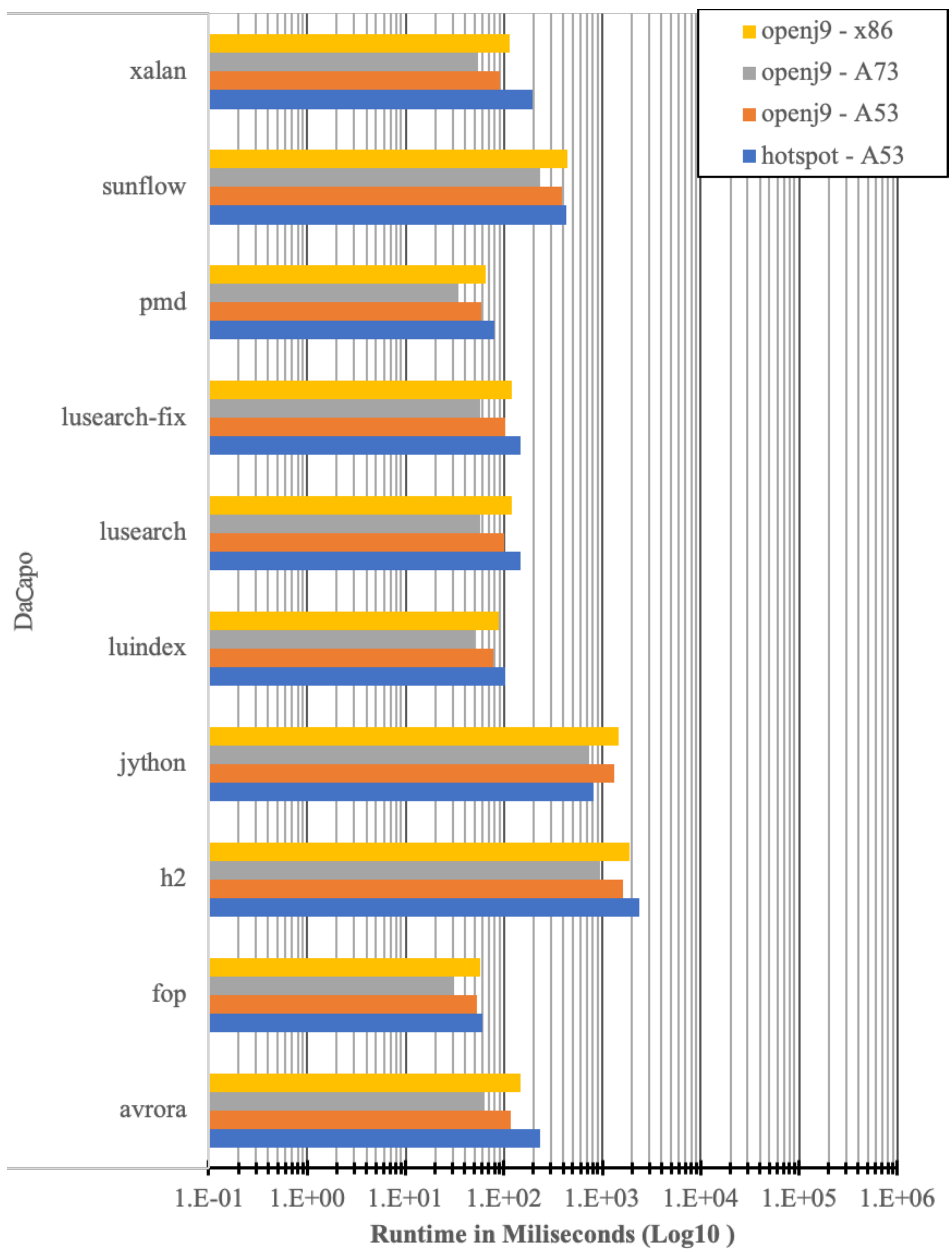

Figure 5.4: DaCapo startup benchmarks execution times using perf

Vita

Candidate's full name

Jean-Philippe Legault

University attended (with dates and degrees obtained)

Bachelors with Honors in Computer Science, University of New Brunswick 2018

Published Papers

[41] 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.

[105] <u>G. Krylov</u>, J.P. Legault, and K.B. Kent. "Hard and Soft Logic Trade-offs for Multipliers in VTR." In: Euromicro DSD/SEAA 2020. IEEE Xplore. Portorož, Slovenia, Aug. 26, 2020.

Presentors are <u>underlined</u>

[108] Young, Scott, Alexandrea Demmings, Nasrin Eshraghi Ivari, Jean-Philippe Legault, and Kenneth B. Kent. “Verilog Loop Unrolling, Module Generation, Part-Select and Arithmetic Right Shift Support in Odin II”. In: *Proceedings of the 30th International Workshop on Rapid System Prototyping (RSP’19)*. New York, NY, USA: Association for Computing Machinery, Oct. 17, 2019.

[114] J.-P. Legault, P. Patros, and K. B. Kent. “Towards Trainable Synthesis for Optimized Circuit Deployment on FPGA”. In: *2018 International Symposium on Rapid System Prototyping (RSP)*. Torino, Italy: IEEE Xplore, Oct. 4, 2018, pp. 90–96.

Dissertations

[106] J.-P. Legault. “Experimental verilog synthesis features for Odin II and alternative multiplier hard-block for fpga”. In: Honours Thesis defended at the University of New Brunswick. University of New Brunswick. Fredericton, New Brunwick, Canada, Sept. 1, 2018.

Talks

[100] 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.

[107] J.-P. Legault. “Experimental verilog synthesis features for Odin II and alternative multiplier hard-block for fpga”. In: 12th Annual Conference for Science Atlantic Mathematics, Statistics and Computer Science by the Atlantic Association for Research in the Mathematical Sciences (AARMS). University of New Brunswick. Fredericton, New Brunwick, Canada, Oct. 1, 2018.

[110] J.-P. Legault, A. 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.

[112] J.-P. Legault, A. 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.

Conference Posters

[101] 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.

[102] J.-P. Legault, A. 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.

[103] 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.

[104] 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.

[109] 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.

[111] J.-P. Legault, A. 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.

[113] J.-P. Legault, P. Patros, and Dr. K.B. Kent. “Towards Trainable Synthesis for Optimized Circuit Deployment on FPGA”. In: Innovation 360, Texpo 2018. CMC. Toronto, Ontario, Canada, Oct. 23, 2018.

[115] J.-P. Legault, A. 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.

[116] J.-P. Legault, Aaron 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.