Optimized arraycopy Implementation for Improved Runtime Performance of OpenJ9 in AArch64

by

Siri Sahithi Ponangi

Bachelor of Computer Science Engineering, Keshav Memorial Institute of Technology, Jawaharlal Nehru Technological University, Hyderabad, 2017

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(s):	Gerhard Dueck, Ph.D., Computer Science
	Kenneth Kent, Ph.D., Computer Science
Examining Board:	Michael Fleming, Ph.D., Computer Science
	Kalikinkar Mandal, Ph.D., Computer Science
	Eduardo Castillo-Guererra, Ph.D.,
	Electrical and Computer Engineering

This thesis is accepted by the

Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

April 2022

Abstract

The Eclipse OMR – OpenJ9 pair is used to build robust language runtimes that support various hardware and operating system platforms. One of the architectures supported by OpenJ9, the AArch64 platform, is widely used in electronic devices because of its reasonable price and resource efficiency. We propose adding an optimization in the Just-In-Time (JIT) compiler of OpenJ9, to copy arrays, including the System.arraycopy() method, efficiently through the arraycopyEvaluator. The arraycopy nodes generated from the bytecode in Intermediate Language (IL) during runtime help in copying values from a specified source to the referenced destination of a given length. The optimizing JIT compiler function, arraycopyEvaluator, separates inlinable code for better performance. Making use of Vector Floating Point registers helps in copying up to 128 bits of any data type in a single load/store instruction. The situations where primitive values are copied or Garbage Collection checks are required to access the reference fields are handled. A comparision of optimized loops with traditional copying—using Java Class Library function System.arraycopy and a loop, without arraycopy optimization in JIT is investigated.

We evaluate the benchmark results of the DaCapo Benchmark Suite and the BumbleBench Microbenchmarking test framework. We investigate the trace files and utilize the Perf tool to identify the reason for unexpected Benchmark results. We achieve an up to tenfold increase in performance. Also, we evaluate the performance improvement on AArch64, with arraycopyEvaluator added, against x86_64.

Compared to x86_64, AArch64 shows an improvement of up to 45.7%.

Dedication

To my loving parents, Ravinder Ponangi and Subhadra Ponangi, my brother Sameer Ponangi and my love Captain Shardul Dubey.

Acknowledgements

This research was conducted within the Centre for Advanced Studies–Atlantic, Faculty of Computer Science, University of New Brunswick. I am grateful for the colleagues and facilities of CAS–Atlantic in supporting our research. I would like to acknowledge the funding support provided by the Atlantic Canada Opportunities Agency (ACOA) through the Atlantic Innovation Fund (AIF) program. Furthermore, I would also like to thank the New Brunswick Innovation Foundation for contributing to this project.

I would like to thank my supervisors Dr. Gerhard Dueck and Dr. Kenneth B. Kent for guidance and support. I would like to extend my gratitude to my Technical Project Manger at CAS-Atlantic Stephen MacKay for proof reading and providing helpful suggestions. Furthermore, I would like to thank Aaron G. Graham for providing continuous and dedicated help in my research. I would also like to thank Kazuhiro Konno and Daryl Maier for their technical guidance, input and feedback time to time.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

AOT	Ahead-of-Time
CD	Continuous Deployment
CI	Continuous Integration
CISC	Complex Instruction Set Computer
CPU	Central Processing Unit
DDR	Direct Dump Reader
DTFJ	Diagnostic Tooling Framework for Java
FPU	Floating-Point Unit
GC	Garbage collection
GPU	Graphics Processing Unit
IL	Intermediate Language
ISA	Instruction Set Architecture
JCL	Java Class Library
JDK	Java Development Kit
JIT	Just-In-Time compiler
JNI	Java Native Interface
JRE	Java Runtime Environment
JVM	Java Virtual Machine
JVMTI	Java Virtual Machine Tool Interface
RISC	Reduced Instruction Set Computer

RPi	Raspberry Pi
SCC	Shared Class Cache
$SIMD$	Single Instruction Multiple Data
SoC	System-on-Chip
SSE	Streaming SIMD Extensions
$TRJIT$	Testarossa JIT
VFR	Vector Floating Point Registers
VSX	Vector Support Extension

Chapter 1

Introduction

Eclipse OMR is a set of components used to build a variety of language runtimes on different hardware [38]. Eclipse OpenJ9 is a Java Virtual Machine (JVM), built on top of OMR [10]. OpenJ9 is constantly being improved by code contributors for different hardware platforms and requires platform specific optimizations.

The AArch64 Instruction Set Architecture (ISA) is an extension of the ARM32 and Thumb32 ISAs with some enhancements [60]. AArch64 has 31 general purpose registers of 64 bits, of which the lower 32 bits are occupied in all cases and extended to 64 bits when required [60, 62]. It can use 32 bits for data types such as int, short and char instead of consuming resources on operations of the unused upper 32 bits [72]. The Neon technology of latest ARM CPUs includes SIMD instructions that operate on 128-bit registers carrying multiple data [95]. The presence of Neon technology in this architecture promises faster processing of data [7].

AArch64 is a newer platform to OpenJ9. The AArch64 platform is widely used in electronic devices because of its reasonable price and resource efficiency [14]. AArch64 is Reduced Instruction Set Computer (RISC) based, hence consumes fewer resources compared to CISC [95]. On AArch64 platforms, OpenJ9 is sought-after due to its reduced memory footprint [2].

A Java program is firstly compiled by a static compiler that is a part of the Java Development Kit (JDK), to generate bytecode [1]. This bytecode is run by a JVM that is a part of the Java Runtime Environment (JRE). The bytecode generated by the compiler is the same on any Java supported machine, while the JVM is machine specific, thus ensuring interoperability. The Just-In-Time (JIT) compiler is the part of the JVM that compiles the bytecode to native machine code during runtime, which reduces CPU and memory costs through optimizations [73]. One of the reasons the JIT is triggered is after a certain number of times (by default, it is 1000) a method is called and the JVM uses this JIT compiled method instead of interpreting it each time [3].

One of the common operations with data is copying. When copying an array of elements, the number of instructions used to copy data varies according to factors such as element size, direction of copying and copying length. Data can be copied in different ways such as in loops and pre-defined functions.

The Java Class Library (JCL) "System" is the most commonly used class for methods such as standard input, output and error streams [31]. Such methods use resources of the JVM such as runtime data areas, registers of the native processors and stacks [30]. The "arraycopy()" is one such method of the System class that is used for copying an array efficiently. It uses a native implementation that is optimized to give improved results.

For better performance, the JIT compiler searches for ways to optimize some of the bytecode obtained after compilation by javac. The optimized code is then executed instead of executing the bytecode directly. This thesis focuses on adding an arraycopyEvaluator method to the AArch64 JIT compiler code. Using this method, the JIT compiler identifies array copying patterns including native methods like "System.arraycopy" and inlines it when applicable. Assembly functions are also added for copying arrays when it cannot be inlined. In addition, vectorization of loops

improves the performance [8].

While copying data, we need to check if the Garbage Collector is operating in the memory region we are copying from. The instructions to implement copying of arrays, that is non-inlinable, must also be added. We benchmark the results and compare them against an advanced and well-developed x86_64 architecture implementation. This gives us an insight into where AArch64 stands when compared to an established architecture like x86_64.

The second chapter discusses the OpenJ9 JVM on AArch64, the JIT compiler, copying of arrays and benchmarking. The third chapter explains the problem statement and implementation details. The fourth chapter includes the machine specifications on which the experiments are conducted. The fifth chapter contains evaluation and results. This is followed by Future Work and Conclusion chapters.

Chapter 2

Background

In this chapter, the background of OpenJ9 JVM on the AArch64 architecture, a brief introduction of Java, JIT optimizations on loops of copying arrays and the x86_64 architecture are discussed. Finally, testing and benchmarking procedures are presented.

2.1 Runtimes

The runtime of a language includes required tools and environment to run a program [5]. A runtime of a language can be implemented in different ways on an architecture [6]. Some of the components of a runtime include a JIT compiler, intermediate language, diagnostic tools and garbage collectors. Interpreted languages can convert the source code into hardware-agnostic code without it having to be compiled [14]. This code is in Intermediate Language (IL) that is similar to the assembly language. An IL code can be optimized and can be run on any supported target. This provides a layer of abstraction over an Instruction Set Architecture (ISA). The atomic unit of what a CPU can execute is called "Machine Instruction" and all the machine instructions together constitute an ISA.

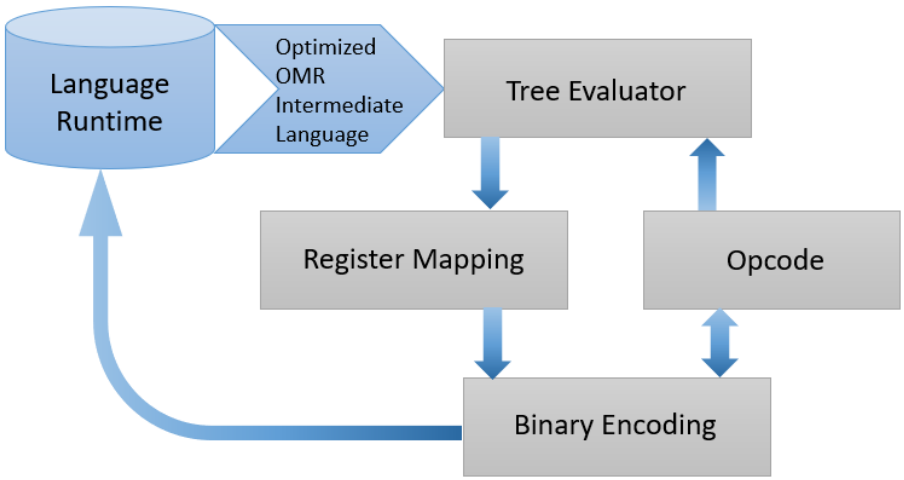

Figure 2.1: Backend of OMR [14]

2.2 Eclipse OMR

Eclipse OMR consists of many components used to build language runtimes, which support different platforms [38, 6]. It consists of a co-ordinated set of open source C and C++ modules that support features such as garbage collection, memory management, and native Just-In-Time compilation [4]. The aim of the OMR project is to provide a runtime technology that provides well-featured languages to enhance the number of options to programmers [4, 6]. For multiple portable programming languages, OMR provides the fundamental features and runs on multiple platforms [9, 6]. As shown in Figure 2.1, the OMR compiler uses its tree evaluators to implement the functionality of opcodes in an IL code [14].

2.3 Eclipse OpenJ9

Eclipse OpenJ9 utilises OMR to build a Java specific runtime [38]. OpenJ9 is becoming popular as an "enterprise grade, production-level component," one of its

consumers being IBM middleware products [12]. OpenJ9 is compatible with OpenJDK for Java 8 and up, with enhancements being made for all the versions. OpenJ9 originated from IBM J9 JVM that has been engineered and in production for about twenty years [11].

OpenJ9 has support to dump the VM and execution state, called Diagnostic Tooling Framework for Java (DTFJ) [13]. Direct Dump Reader (DDR) is an implementation of DTFJ in Java. Any failures caused during the execution of Java programs in the OpenJ9 building process will result in a dump by DDR. DDR, which is a part of OpenJ9, also consumes some code of OpenJ9 to build itself.

An unforeseen exception or error in the OpenJ9 code base causes failure in the build, sometimes without the flaw being obvious. After the build is completed, a Java application can be compiled and executed, using javac and java respectively, which are found in the image folder within the build. OpenJ9 provides various command line options to configure the runtime environment to start a Java application, some of which are discussed in Chapters 4 & 5 [26].

2.3.1 Just-In-Time compiler

The JIT compiler in OpenJ9 is called Testarossa JIT (TRJIT), derived from the IBM J9 JVM [15, 17]. There are different optimization levels at which TRJIT compilation takes place, recognized and named related to temperature: cold, warm, hot, veryHot and scorching [15, 73]. TRJIT is composed of four key modules: IL Generator, Optimizer, Code Generator and Compilation Control [15]. The bytecode is converted into a tree format in IL Generator and Optimizer performs various optimizations such as loop optimizations, data flow optimizations and inlining [17, 15, 16]. The Code Generator converts this IL tree input to native instructions and Compilation Control decides the method optimization [15, 16]. The components of the J9 runtime environment that are not related to the Java language make OMR and the Java

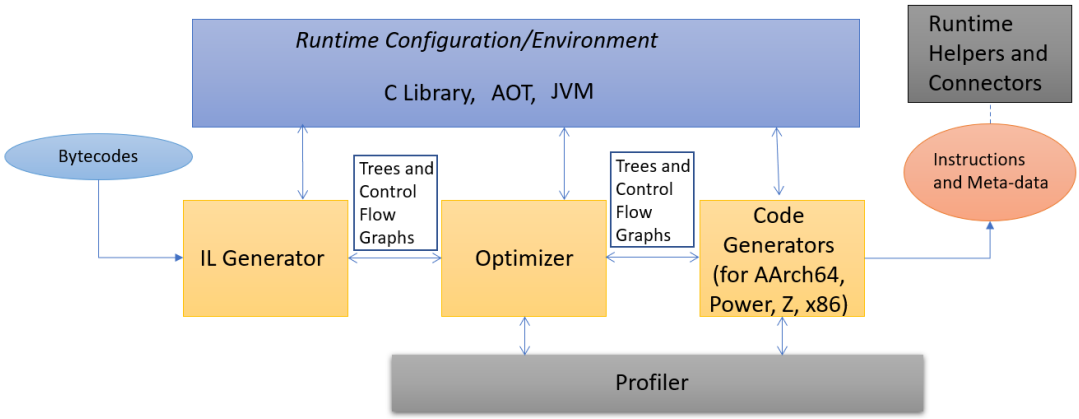

Figure 2.2: TRJIT Architecture [81, 23]

specific components come under OpenJ9 [83]. OpenJ9 has C++ classes that inherit classes from OMR and hence, methods of OpenJ9 override those of OMR. OpenJ9 and OMR connect through the OMR::TreeEvaluatorConnector namespace. Figure 2.2 demonstrates an overview of the TRJIT compiler.

When the JVM decides to optimize a method, the bytecode of that method is processed by the JIT compiler [22]. The bytecode is converted into assembly-like instructions and this IL is ordered as a tree. The code generator of the JIT compiler code has a TreeEvaluator class that contains a list of evaluators, each performing analysis and optimizations on the tree. Most of these evaluators are implemented in each of the architectures' compiler code. The evaluator methods that are not implemented contain a function call to two methods that generate dummy evaluators. They are either unImpOpEvaluator and badILOpEvaluator. The TreeEvaluator class is extended by OpenJ9 in its compiler code, with more opcodes specific to OpenJ9 implementation. Hence, OMR's implementation of an evaluator on a specific architecture is overridden by its counterpart in OpenJ9.

The tree evaluation process includes operations on the generated IL tree [18]. These

operations consists of unary operations, binary operations, various checks and optimizations. Each evaluator has a unique functionality that operates from a tree top. The usage of virtual registers and instruction assignments is decided in this phase. Subsequently, the evaluator has a counter that keeps a count of the children of its sub-tree. The counter is decreased by 1 when a child is no longer used and the child's node is eliminated once the count reaches 0.

The Just-In-Time (JIT) compiler is a primary element of OpenJ9 that improves performance by compiling bytecode to native code during runtime [73]. The JIT compiler selectively compiles methods that are repeatedly called for a preset threshold count [14]. The JIT compiler starts with the VM, so any JIT options must be given while starting the VM [73]. The verbose log option allows the JIT to generate the list of methods that are compiled by the JIT. By turning the JIT on and off and analyzing the verbose log, the source of a build failure can be identified.

2.4 AArch64

AArch64 is one of the ARM architectures, which have evolved to focus on implementation, security and resource consumption [60]. AArch64 is a Reduced Instruction Set Computer (RISC) with data-processing happening only on registers and not on memory [32]. ARM's latest Neon technology includes advanced Single Instruction Multiple Data (SIMD) and its related impositions [24]. Neon registers act as vectors, which can hold multiple elements of same data type. Neon instructions operate on multiple elements at the same time. These registers are the same registers used for Vector Floating Point values [25]. This technology is intended to improve performance of activities such as audio and video processing, user interface, and visual processing [24]. Advanced SIMD instructions and floating-point intructions perform operations on files of SIMD and floating-point registers [32]. Using SIMD instruc-

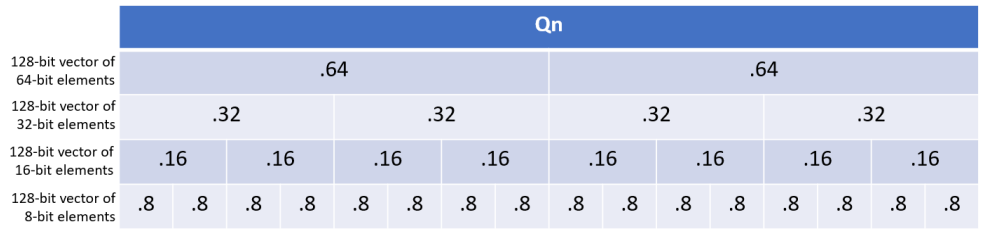

Figure 2.3: Vector in ARMv8-A [32]

tions can improve an application's performance up to 64.49% [91]. This thesis is focussed on the application profile of AArch64, called as ARMv8-A in 64-bit execution state, which supports Virtual Memory System Architectures [32]. The integer data types supported by the ARMv8 architecture are byte, halfword, word, doubleword, quadword along with half, single and double precision floating points.

A Reduced Instruction Set Computer (RISC) has an optimized set of instructions, which are fewer than in the Complex Instruction Set Computer (CISC) [35]. As there are fewer instructions in RISC, compilers tend to generate more lines of code for RISC architectures. RISC designs are adopted by architectures such as ARM and the Power ISA. It is also alternatively referred to as a load/store architecture due to the use of load and store to fetch the memory.

Vector registers (128 bits) are also supported, which can hold multiple elements of the same data type as shown in Figure **??**. This architectural model permits the operating system to access system resources in a distributed and protected way. A processing element of ARMv8-A has 31 general purpose registers which can be used as 64-bit or 32-bit registers.

Single-copy and multi-copy atomicity is maintained with load/store instructions, load/store pair instructions and SIMD instructions by serialized writes for a location [32].

As in shown in Figure 2.1, the different addressing modes take a 64-bit base address

Table 2.1: Addressing modes in AArch64 [32]

	Offset		
Addressing mode	**Immediate**	**Register**	**Extended Register**
Base register only (no offset)	[base, #0]	-	-
Base plus offset	[base, #imm]	[base, Xm ,LSL #imm]	[base, Wm, (S—U)XTW #imm]
Pre-indexed	[base, #imm]!	-	-
Post-indexed	[base], #imm	[base], Xm	-
Literal (PC-relative)	label	-	-

and stack pointer while the offset is optional for immediate and register [32]. The cache stores information about a main memory address (tag) and data associated with it, which is responsible for increasing memory access speed. Cache considers spatial and temporal locality, hence, any data that needs to be accessed, present in cache (cache hit) will result in better performance.

2.5 Java

Java®[1] is a general-purpose programming language, which is not specialized for any particular domain [28]. A Java program can be "written once and run anywhere" without recompiling, across all Java supported platforms. The Java program is converted into bytecode by the compiler that can run on any architecture using a virtual machine, irrespective of the platform [27]. A Java Virtual Machine (JVM) transpiles the Java bytecode to machine code to be executed by the processor [14, 29]. There are various JVMs such as GraalVM, HotSpot, OpenJ9 with differences like the languages they are implemented in and their purposes [18, 19, 20]. A Java Development Kit (JDK) is comprised of a JVM and other resources for development of a Java application [1]. A Java Runtime Environment (JRE) provides necessary

[1]Java is a registered trademarks of Oracle and/or its affiliates

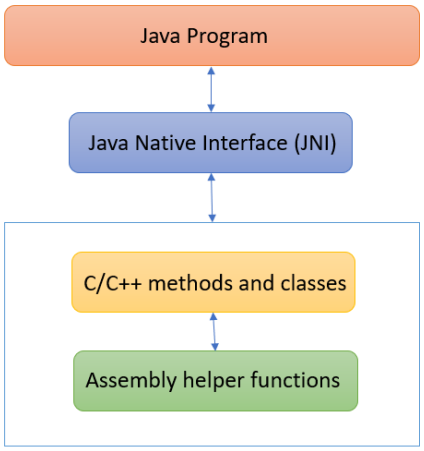

Figure 2.4: Java Native Interface

libraries to execute the Java program using a JVM [37].

2.5.1 Java Native Interface

The Java Native Interface (JNI) allows a running Java program to utilize native methods written in Java, C/C++, or native code that can be compiled, specific to the hardware and software, available in libraries [40, 82, 81]. The main objective of JNI is to provide optimized native support to the methods for a compatible platform, in all available JVMs [43]. JNI is used to give platform dependent support, accessing libraries in another language, such as C, or creating a function with the help of assembly language [41]. As shown in Figure 2.4, the native methods and assembly functions have information helpful in debugging and in adding more native support to Java code.

2.5.2 Java Class Library

The Java Class Library (JCL) is a collection of loadable libraries used by Java programs during runtime [42]. JCL is provided in the JDK and is written in Java,

making use of JNI as needed. The class System extends class java.lang.Object that is defined in the JCL, and holds multiple frequently and commonly used methods and variables, which can be imported and used in a Java program [44]. The most commonly used methods of the System class include input and output streams. System.arraycopy calls VM specific native methods during runtime using JNI. These native methods are optimized to give better performance while using methods defined in JCL. The native methods would further call memory copying functions tailored for different cases such as forward and backward copying.

2.6 The arraycopy Method

The arraycopy method is also one of the frequently used static methods of the System class [44]. This method takes five arguments, which are source array, the position from which the array should be copied, the destination array, the position in the destination array to which the array has to be copied and the length, which is the number of elements to be copied [44]. This method throws exceptions, such as NullPointerException, ArrayStoreException, or IndexOutOfBoundsException, when the source or destination arrays are null, the index of the array does not exist or if any of the arguments are negative [31]. These exceptions have backend support from the compiler written in C, C++ or assembly. There are architecture specific assembly helper routines used during runtime, which perform checks to generate exceptions. Certain checks need to be implemented before arraycopy is called, which are identified in this thesis. Optimizations in the compiler will recognize a call to System.arraycopy() and then turn it into an "arraycopy" IL opcode (either primitive or reference). The arraycopy optimization, which is implemented in this research, will be utilized to implement this IL opcode for efficiency.

The arraycopy implementation of the OMR and OpenJ9 pair provides architecture

Listing 2.1: Signature of System.arraycopy() method

```
/**
 * Copies the contents of array1 starting at offset start1
 * into array2 starting at offset start2 for
 * length elements.
 *
 * @param  array1       the array to copy out of
 * @param  start1       the starting index in array1
 * @param  array2       the array to copy into
 * @param  start2       the starting index in array2
 * @param  length       the number of elements in the array to copy
 */
public static native void arraycopy(Object array1, int start1, Object
array2, int start2, int length);
```

specific optimizations for better performance. Built-in functions from libraries are more efficient compared to user-defined functions, as the effectiveness and competence is boosted as they get updated [33]. Using a loop for copying an array cannot perform as well as System.arraycopy unless the size of the array to copy is small enough that there is an overhead caused by calling native methods and performing various checks. The native method performs better for large array sizes because the native methods can optimize the code in ways that are unachievable in Java. System.arraycopy shows significant performance and time improvement in contrast to manual copying [34]. Any copying of an array, including using loops and the System.arraycopy method is identified by the JIT and are turned into arraycopy nodes depending on the optimization level of the JIT. In this thesis, we focus on the ARMv8-A (AArch64) architecture to build a runtime for Java with optimized arraycopy implementation. The arraycopy method is implemented in Power, x86_64 and Z compilers and has two modes, which are primitive arraycopy and reference arraycopy. The Intermediate Language (IL) tree nodes created for these two variations give the input details for the arraycopy implementation. For both types, a source byte address, destination byte address and the length in bytes are in common. In the reference copying node, two additional children, source and destination array objects

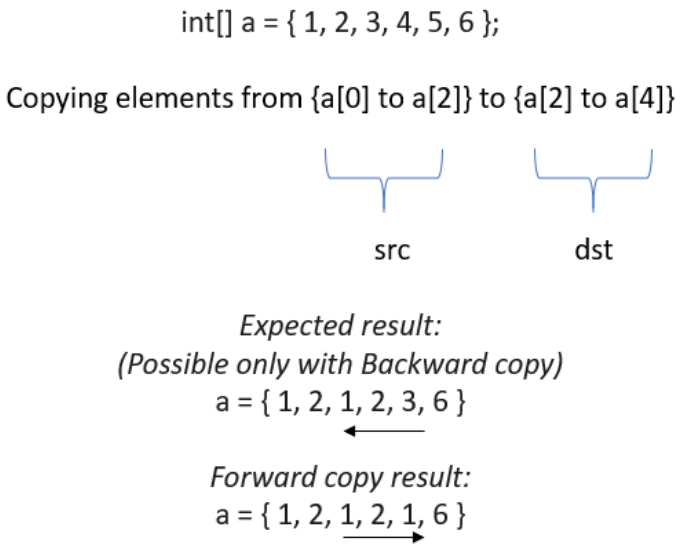

Figure 2.5: Forward copying of an array vs. Backward copying of an array

are included. There are two ways the array can be copied, forward and backward. In the backward copy, the source and destination pointers point to the last element and the copying is done starting with the last element and moving to the first. Forward copy starts with the first element of the array to the last element. Most of the array copies are carried out in the forward direction. Backward arraycopy is not triggered unless it is either given in the program, or there is a need to do backward copy. The need for backward arraycopy arises when there is an overlap in the addresses of the ending of source and the beginning of the destination arrays. As shown in Figure 2.5, the results vary for backward and forward arraycopy. With the forward copy, the value at the second index is updated before it can be copied to the destination. Backward copying provides the expected results in this case. The arraycopy implementation is an optimization that is generated at various rates based on the levels of optimization. This optimization can be enabled or disabled from OpenJ9's command line options.

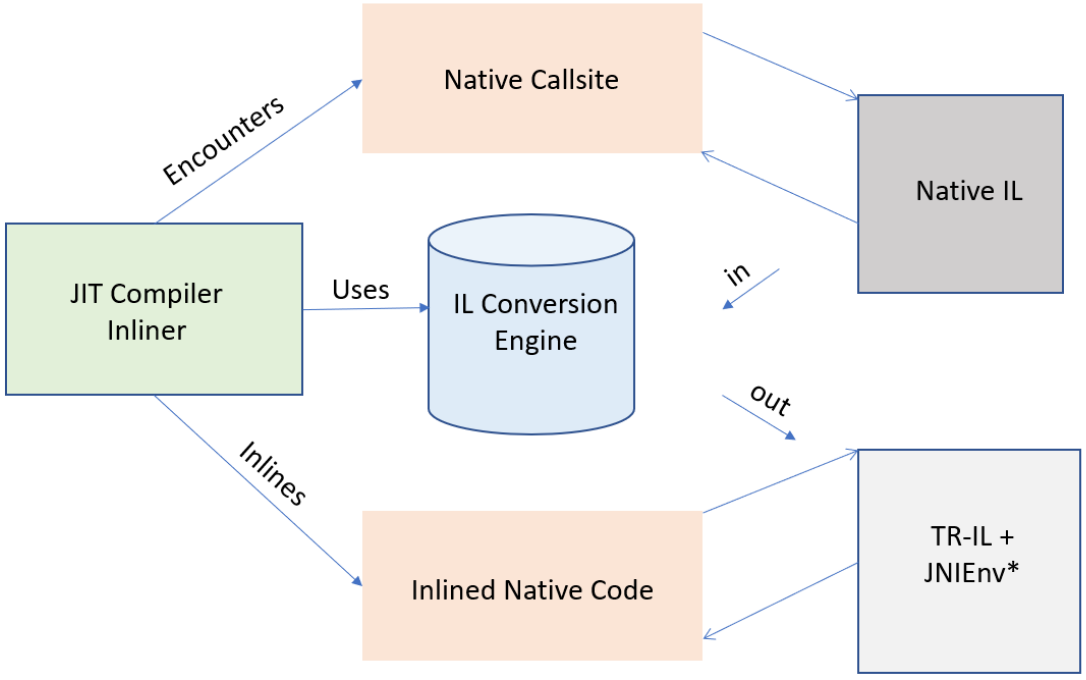

Figure 2.6: Inlining Native Methods [16]

2.7 Inlining methods

Inlining is an optimization applied during compile time, to simplify the code to be executed [64]. When a method is inlined, the method call is replaced by the code of the method to be called. This reduces the time and memory overhead of calling the method. Inlining is one of the phases in which JIT optimizes the code [84]. Since the JIT compiler processes the code in the form of trees, the tree is restructured such that the callee method tree replaces their callers. According to Kaser et al. [85] and McFarling [86], there are indirect effects of inlining such as its positive effects on cache and virtual memory behaviour. In this thesis, we venture to inline vectorized copying of arrays for better performance. Figure 2.6 shows the process of inlining native methods after they are converted to IL in TRJIT [16]. The callsite of the native method is identified by a pointer (JNIEnv). The JIT compiler identifies the

native functions to be inlined and converts them to IL to optimize them. This happens during the runtime of a Java program, and the IL of native functions resembles the IL of Java bytecode. Thus the native functions are further optimizable by the JIT compiler based on the level of optimization, when they are converted to IL.

2.8 Garbage Collection

Garbage collection (GC) is a technique whereby an object in memory that cannot be reached using another reachable object and when its reference is not on a root variable, is collected by garbage collectors [46]. There are different garbage collection policies supported by OpenJ9, such as gencon and balanced [47]. These policies are designed such that they are focussed to optimize in a particular regard during the runtime of an application. [48]. The policies are distinguished based on the GC cycles and layout. A GC cycle is a set of steps followed to identify reachable objects and moving them, such as a 'mark' operation.

The GC policy to be used can be chosen using a command line option; the gencon policy is the default and need not be specified.

The gencon policy has two heap areas called nursery and tenure, with 'stop the world' scavenge operation, concurrent scavenge, optional compact operation after concurrent mark-sweep operation.

The policy is decided based on factors such as the lifetime of the objects, heap size and pause times of a GC policy under certain conditions. In this thesis, we use a particular variant of gencon, called Concurrent Scavenge, known to reduce 'stop the world' pauses. 'Stop the world' pauses occur during GC when the application is stopped temporarily while GC transfers existing objects to new locations [49].

The tenure area contains old objects with long life whereas nursery contains newer

objects [53]. The nursery area has an allocate area for allocating newer objects and a survivor space where the objects are copied to during GC. As the objects are moved from the allocate area to the survivor area, the object's age is incremented. When the object in the survivor area increases to a certain threshold, it is moved to the tenure space. The generational collection cycle comes to an end as the objects in the allocate area are emptied into the survivor area or from survivor area into the tenure space. The survivor and allocate areas are then swapped so that the survior area becomes the new allocate area and vice-versa.

When data needs to be copied from one memory area to another, if the destination object field is made to point to the source object value reference instead of creating a new reference, it is called shallow copying [51]. Copying arrays is a shallow copy, and the GC can interfere with copying by relocating objects during its run. The primitive data type values are stored on a stack in contrast to reference objects, which are stored in the heap [50]. Hence, GC checks are involved only during reference copying. All the objects that are generated using the keyword 'new' are allocated on the heap, although references to them can be on stack. Hence, in the case of reference array copying, it must be first checked if the GC is active [53]. The GC is required to carry-out object relocation and book-keeping, which is handled by the 'read barrier' and 'write barrier' VM helpers. The 'read barrier' check is performed before reference copying to load data. A 'write barrier' check is carried out after storing data, to protect it from GC operations [52]. Figure 2.7 illustrates how reference fields are accessed. The decision to call the VM helper is made by the JIT compiler depending on if the latest reference is from a GC region.

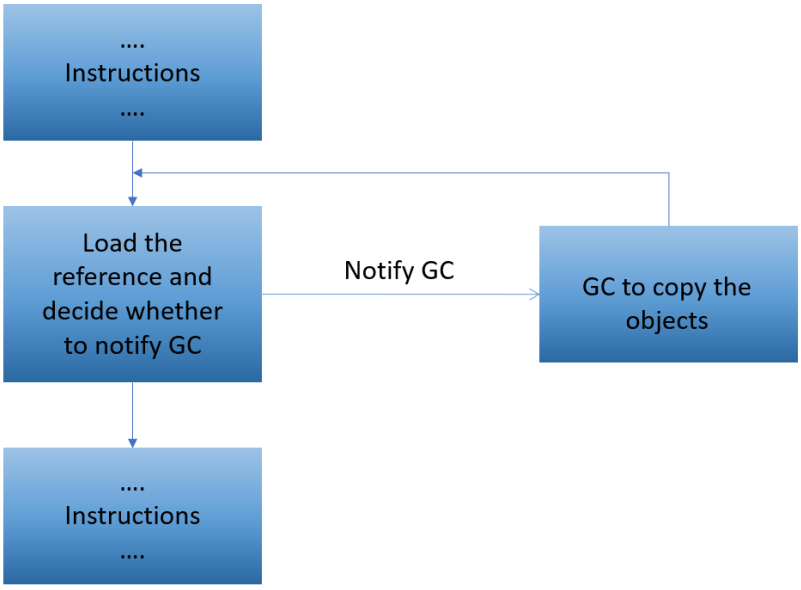

Figure 2.7: Accessing a reference field under Concurrent Scavenge policy [53]

2.9 x86_64

The x86_64 ISA is one of the widely used CISC architectures, initially developed by Intel® [69, 70]. In x86_64, arraycopy is implemented using its Streaming SIMD Extensions (SSE). In this thesis, we compare the performance improvement on x86_64 to that of AArch64.

2.10 Software Development Process

In this thesis, we follow the steps mentioned in the Appendix D to clone OMR and OpenJ9 repositories. We refer to the existing implementation on other architectures to understand the semantics of OMR and OpenJ9. We also explore the 'Arm® Architecture Reference Manual - Armv8, for A-profile architecture' to know AArch64 connotation. We follow the design mentioned in the Chapter 3 to implement arraycopyEvaluator.

2.11 Evaluation

The code is primarily tested by creating a successful build. The code is then verified by testing the JVM with different tests available on OpenJ9 or written specifically to test the implementation. For the code to be merged in the main repository, it goes through a series of predefined tests to substantiate its stability. This process is explained in detail in the following subsections.

2.11.1 Continuous Integration and Continuous Deployment

Continuous Integration (CI) is the process of integrating the working branch code with the main branch in a testing environment [45]. Continuous Deployment (CD) is a process that involves tests such as user acceptance and performance test. As shown in Figure 2.8, after all the tests are passed, the code is merged into the production environment and the new features go live instantly. The code written for arraycopy is tested natively and locally, and then it goes through a series of tests in the CI/CD pipeline, where additional tests are performed.

2.11.2 Debugging

The generated OMR Intermediate Language (IL) code is studied from trace logs to validate that the expected opcodes are generated. The trace logs or trace files include the JIT optimized trees in a readable format. We can learn about JITed code such as how the arraycopy nodes are formed and the instructions they use when inlined. For building the code natively and to generate trace files, a Raspberry Pi 4 Model B Rev 1.4 is used. The builds are either in debug or release mode.

GDB is a debugger that is used to observe the instructions in a program step by step while the program is being run [54]. It has features such as stopping at a set

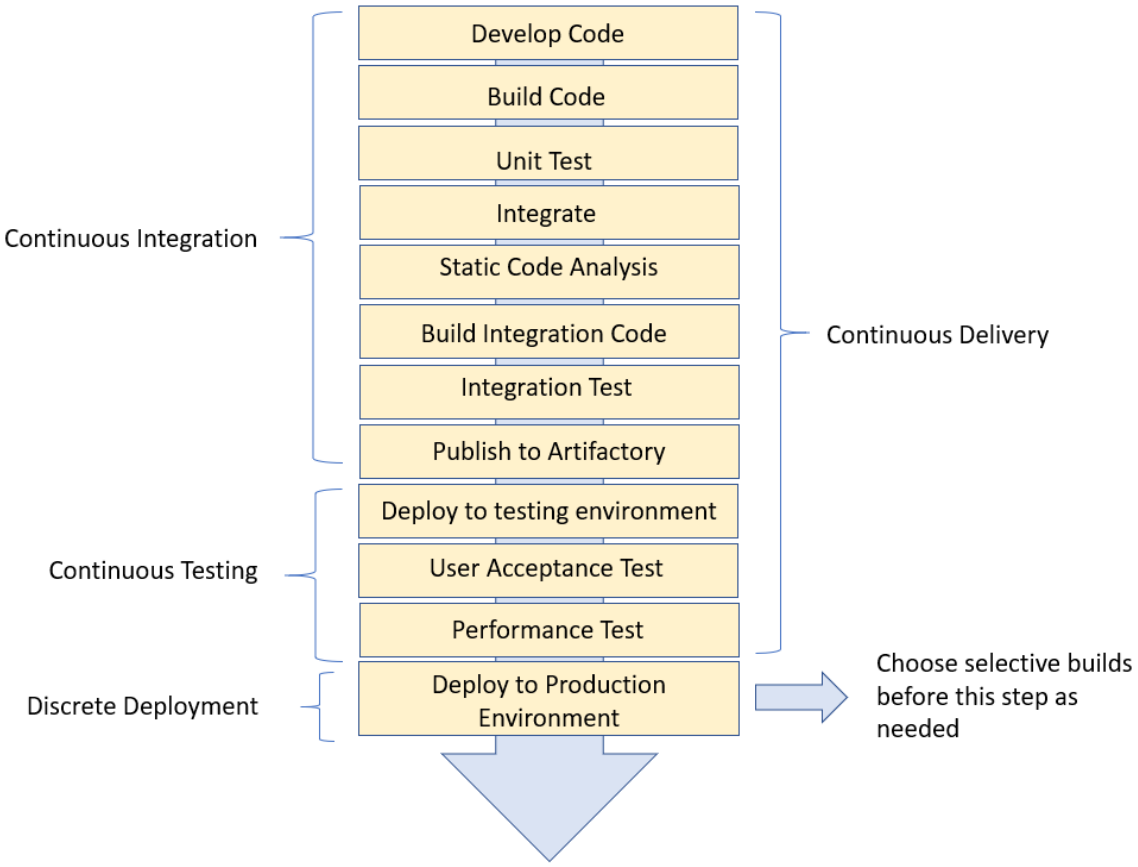

Figure 2.8: Continuous Integration and Deployment [45]

breakpoint, display values at a particular variable when the program is paused. It supports languages such as C, C++ and assembly. Since most of the code is written in C/C++, it is possible to see the flow of instructions under a debugger like GDB. Also as described in Section 2.11.4, the Perf profiling tool is used to see the assembly notation.

2.11.3 Benchmarking

Benchmarking is a process to calculate the performance considering multiple factors of a virtual machine in terms of optimizations, algorithms and other features of the VM [56]. The calculated results of benchmark suites is used to compare VMs to analyse the performance of adding a new feature or optimization. A microbenchmarking test framework is used to calculate performance, often based on a single function or small part of code run repeatedly [55]. This identifies the performance difference more accurately on sectional code development.

Bumblebench is one of the commonly used microbenchmarking test frameworks for OpenJ9 [77]. There are special built-in tests available in BumbleBench for arraycopy, which help us to see the performace of arrays, of various datatypes and lengths. In this thesis, we experiment by tweaking the existing benchmark for arraycopy, to generate arraycopy nodes various ways.

The DaCapo benchmark suite is meant for benchmarking Java with real world applications on various architectures [57]. DaCapo is known to have a very high code complexity, many class hierarchies and structures, complex memory composition and object behaviour. Table 2.2 shows the average number of arrays allocated per second (array rate) in a sub-benchmark and the average array length in the DaCapo benchmark suite. The average values are calculated by dividing the total number of arrays, objects and aray lengths by the total run time. All the DaCapo sub-

Table 2.2: Average Array length and Array rate [56]

Sub-benchmark	**Average Object Size**	**Array Rate (%)**	**Array Length**
	(bytes)		
avrora	30.5	31.8	12.1
fop	42.5	35.1	19.8
h2	38.2	38.5	10.8
jython	43.4	32.1	58.4
luindex	48.9	38.0	182.7
lusearch	103.6	44.0	553.0
lusearch-fix	103.6	44.0	553.0
pmd	36.1	40.3	25.4
sunflow	45.1	3.4	10.5
xalan	48.4	40.3	60.9

benchmarks include arrays, with lusearch having the highest average array length and highest average object size [56]. The sub-benchmark having the smallest average object size is avrora followed by pmd. The lowest average array rate and lowest average array length lies with sunflow but it consumes the largest heap size, similar to xalan [56, 59]. The sub-benchmarks fop and h2 have similar average object sizes, array rates and lengths, hence, their performances are expected to be close. Also, luindex and xalan share the same average object size and array rate, but luindex has almost three times the average array length of xalan. There is only a single line difference between lusearch and lusearch-fix codes, since lusearch allocates the highest workload, it is included in the results as well [59]. Since we do not know the direction of copying and if they are primitive copying nodes or not, we cannot determine the purpose of each sub-benchmark. Table 2.3 shows the purpose of each sub-benchmark.

In this thesis, we will see a comparision of improvements in performance due to arraycopy implementation on x86_64 architecture with AArch64.

Table 2.3: DaCapo sub-benchmarks [58].

Sub-benchmarks	**Purpose**
avrora	Creates an AVR microcontroller environment and runs tests
fop	Analyzes and processes an XSL formatting file and generates an output file such as a PDF
h2	Measures the performance of a complex online transaction processing system (like TCP-C)
jython	Runs a Java application that works as a Python interpreter
luindex	99 A program that arranges texts and documents in an order
lusearch	Searches for words in a document
lusearch-fix	Searches for words in a document
pmd	Inspects Java source code classes for possible problems
sunflow	A 3D image processing software
xalan	Parses XML documents

2.11.4 Perf

Perf is a profiling Linux tool that is used to measure performance, such as the number of instructions, cache misses and other information of sampled events [65]. It is capable of recording events, breaking them down by function to report them and annotate assembly instructions [66]. However, the standalone Perf tool provides insufficient support to profile JIT compiled code [67]. The Java Virtual Machine Tool Interface (JVMTI) creates a medium between a native agent and a VM to retrieve information relevant to the JITed (JIT compiled) code [68]. The Perf tool, along with JVMTI agent, is run to record the events from a Java program [67]. The JIT code is then injected into perf.data, a binary file with the information collected by the Perf tool. Once it is reported, the assembly annotation can be seen.

Chapter 3

Design and Implementation

This chapter discusses the problem statement and the approach to address it. The performance enhancements due to existing arraycopy optimizations on x86_64 are observed using the DaCapo Benchmark suite in this chapter.

3.1 Problem statement

JNI provides an interface to call native methods directly, instead of compiling Java code [40, 82]. This can also be considered a disadvantage as the JIT compiler cannot optimize nor can it introduce inline JNI methods, which is called 'compilation barrier' [81]. However, the TRJIT processes native methods to IL and inlines them [16]. In this thesis, we investigate if the JNI method System.arraycopy can be optimized by JIT or not.

Currently in the Power compiler, arraycopy is implemented and optimized to copy sixteen bytes at a time using vector instructions. The arraycopyEvaluator is capable of copying groups of 256 bytes in a single instruction in the Z compiler. The arraycopyEvaluator copied up to 16 bytes using a Floating Point Register (FPR) in the x86_64 compiler. These JIT optimizations are used for inlining arraycopy loop

patterns which produce speedups. Similar to the above mentioned architectures, we want to add a similar advantage to AArch64.

This research focuses on the implementation of an inlined arraycopy for AArch64 by copying up to sixteen bytes in a single instruction, making use of a vector load and store. This is possible when the copying length is a constant (known at compile time) and it is more than 16 bytes. Load-store pair usage also helps in achieving a similar goal of copying 16 bytes at a time (64-bit each), but it will consume two registers. The runtime performance improvement due to JIT optimization on copying arrays will be examined. There are two ways an array is copied, simple arraycopy and reference arraycopy. The simple copying is for primitive data types or copying that does not deal with GC checks. The reference version is for object types where reference fields are accessed, involving GC checks. We define the arraycopyEvaluator in the AArch64 compiler code, similar to the Power compiler due to its vector usage and RISC type.

Separate optimizations for 16-bit primitives and 64-bit primitives on Intel Architecture 32-bit (IA-32) are available in the x86_64 compiler code. Primitive data types larger than 32-bit exist on IA-32 and the operator carries the type of the data [90]. Similarly, we explore implementing optimizations for primitives in AArch64. The direction of copying, as mentioned in Section 2.6, is determined during an application's runtime. Different instructions are generated for byte, halfword, word, double word and quadword for with and without direction. A vectorized implementation is available for arraycopy along with constant length, direction for primitive arraycopy and reference arraycopy modes in the Z compiler code. We implement the arraycopyEvaluator method with vector support along with word and halfword for primitive arraycopy and reference arraycopy modes in the Power (P) compiler. We see an analysis of a similar implementation in AArch64, in this thesis.

Copying arrays longer than sixteen bytes can be optimized by iteratively copying

each group of sixteen bytes with a single vector instruction for a given array length greater than sixteen. We add a suitable optimization for each data type in assembly functions. Initially, we will implement the primitive arraycopy version, using loops. We will implement a reference arraycopy version, by reusing the primitive arraycopy version, which is for reference types and requires additional checks. We investigate an efficient way to move data on AArch64, like using vector instructions. Finally, I demonstrate a comparision between optimized and non-optimized loops.

Due to the reasons discussed in Section 2.7, we attempt to inline copying of arrays. Inlining is applicable only when the copying length is constant in the case of arraycopy. This is because the JIT cannot see the copying length if it is stored in a variable, as it only runs at runtime.

The optimizations will be made by taking advantage of the architecture. The primitive arraycopy version will be initially implemented with assembly functions, followed by the implementation of the reference arraycopy version. The runtime performance of the arraycopy implementation in AArch64 will be analyzed and compared to x86_64.

3.2 Methodology

We make use of vector registers to copy 128 bits at a time in a single instruction. This, combined with inlining, is expected to improve the performance of copying arrays. The copying of arrays cannot be inlined when the number of elements to be copied is stored in a variable. When inlined, the TRJIT generates a loop that copies 64 bytes using four sets of load and stores of a SIMD (vector) register when the copying length is 128 bytes or longer. It does not generate this loop when the copying length is less than 128 bytes, to avoid the overhead of operations with the loop counter. See Listings D.1-D.7 for examples of generated native instructions.

The non-inlinable copying of arrays is executed by copying one element at a time like an interpreter would do. This would generate fewer native instructions compared to inlined methods. As this is a JIT optimization, the size to be copied is not known yet, hence it cannot be inlined.

The copying is classified into two types, primitive copying and reference copying. Primitive copying is also known as simple copying because data of primitive data types is copied. It does not involve additional checks for GC, whereas in reference copying, references or non-primitive data types are copied. When accessing references, read barriers should be checked to know if the GC is working on the same memory. Read barriers are implemented only on Concurrent Scavenge policy of GC on OpenJ9 as more development is expected around GC.

Traditionally, memcpy and related functions are called for copying an array. The native arraycopy function is called, where optimizations are performed, in the case of System.arraycopy. The arraycopyEvaluator function replaces the native System.arraycopy and memcpy functions for rest of the loop patterns, in the proposed solution. As seen in Figure 3.1, the arraycopyEvaluator method depicts how copying is performed. At the time the arraycopyEvaluator was added, read barriers existed only for the Concurrent Scavenge policy. Hence, if there are no read barriers, the control flow carries on with the regular implementation. The code contribution for arraycopy on AArch64 is as mentioned in the Appendix A.

3.3 Prerequisites

There are two evaluators that belong to the Tree Evaluators, which need to be implemented before arraycopy. The TR::arraycopyBNDCHK evaluator is utilized to check if there are any objections to reading and writing the source and destination arrays respectively. The TR::arraychk evaluator is to check if the array can be

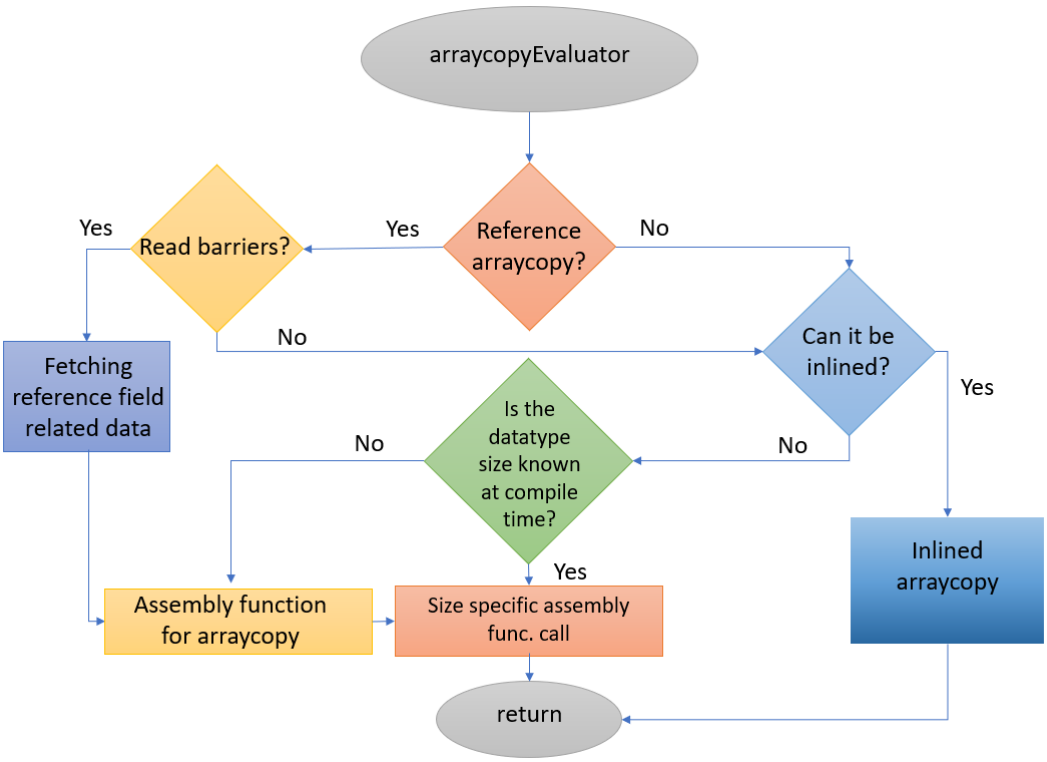

Figure 3.1: The arraycopyEvaluator method design

copied. In the JIT compiler code, the transformArrayCopyCall function has checks to perform before creating arraycopy nodes. To be able to generate arraycopy nodes, the calls are made to the above evaluators. The arrayCHKEvaluator is responsible for type checking such as if the objects are of array type, if their classes match and if they do not, they both need to be reference types. If any of the checks above is not passed, an exception is thrown. The ArrayCopyBNDCHKEvaluator performs bounds checking and throws an exception when the checks fail. It checks if the length to be copied is within the index bounds. An inlinearraycopy function will be added to perform array copying. This will be utilized by the arraycopyEvaluator function.

3.4 Checkpoints

Primitive and reference arraycopy need to be enabled for AArch64, identical to other architectures. Although building OMR alone does not create a runtime, it can be

built independently to verify that its code is stable. OMR's arraycopyEvaluator will only have a language agnostic version of arraycopyEvaluator. OpenJ9's arraycopyEvaluator will include access to GC checks and objects, that is related to Java. Hence, it is enabled in the downstream project OpenJ9 as well, similar to other architectures. The arraycopyEvaluator optimizations are disabled by hardcoding the command line option to disable it, as enabling without implementation may lead to build failures, in OpenJ9's AArch64 code base.

Not all the parts of a method can be inlined. Some checks such as if copying length is a constant are carried out to see if it can be inlined or not, in the arraycopyEvaluator method. Only in the case of constant length where direction is known and no checks are needed, arraycopy can be inlined. The assembly functions for arraycopy that copy element by element are used, for the part that cannot be inlined. There are functions for arraycopy for unknown direction, forward and backward copying functions for unknown data alignment and known data alignment. Some optimizations such as skipping the redundant length check is applied on the loops in assembly functions for arraycopy as well. Since inlined functions copy more elements in groups than assembly functions, assembly functions' performance is not expected to be the same as inlined functions.

Chapter 4

Experimental Setup

This chapter discusses different machines and their configurations, on which the arraycopy implementation is tested and benchmarked. The arraycopy sequence that is generated for each case is verified in the testing section. The difference between the performance for each sub-benchmark on x86_64 and AArch64 architectures is observed in the benchmarking section. The performance is evaluated by comparing the benchmark results of the Java version included in Appendix A, that include the arraycopy implementation.

4.1 Testing and Benchmarking

I test the arraycopyEvaluator on the AArch64 machine with specifications as in Table 4.1. The other devices available were Khadas VIM3, Cortex-A73/A53 based and ThunderX2 - Cavium CN9975. I chose a Raspberry Pi as it is the most affordable and resource constrained compared to the rest. The sub-benchmark in BumbleBench benchmark that focuses on copying of arrays is ArrayCopy. I run this sub-benchmark to compare the performances with and without arraycopyEvaluator.

Breakpoints can be set to pause when the arraycopyEvaluator is called in the GBD

Table 4.1: Specification of Raspberry Pi 4 Model B

Architecture	aarch64
Model	Raspberry Pi 4 Model B Rev 1.4 / Cortex-A72
Features	fp asimd evtstrm crc32 cpuid
Hardware	BCM2835
Cores	4
Minimum CPU Frequency	600 MHz
Maximum CPU Frequency	1500 MHz
Locked Frequency	1300 MHz

Listing 4.1: Disabling arraycopy

```
export OPENJ9_JAVA_OPTIONS='-Xnoaot -Xjit:disableArrayCopyOpts'
```

debugger. By back-tracing from that point, the generation of arraycopy nodes can be traced back from the create arraycopy method in the code base, which is called by the optimizer in the compiler. The hotness level of the JIT, as described in Chapter 3, is set to "veryHot", so that 'transformarraycopycall' is called. OMR::ValuePropagation::canRunTransformToArrayCopy() is called by the transformarraycopy method, which returns true only when the optimization level is warm or higher. The flow of method calls can be checked in gdb by setting breakpoints in a function. The arraycopy optimizations can be disabled using an environment variable as follows. Along with disabling the arraycopy optimizations, Ahead-of-Time (AOT) compilation is also disabled. This is because AOT can already optimize the array copying, even though it is turned off in the JIT.

The number of times arraycopyEvaluator is accessed can be counted using an fprintf statement. The signature of the function using the arraycopyEvaluator method, which implements arraycopy, can be seen using the following statement. OpenJ9 has

Listing 4.2: Debugging option for arraycopyEvaluator method

```
fprintf(stderr, "[arraycopy] %s\n", cg->comp()->signature());
```

Listing 4.3: Debug counters for arraycopy

```
-Xjit:debugCounters={*arraycopy*}
```

a JIT option to count the number of times a method is called during compile time. This is implemented using the debugCounters parameter, to which a regular expression is assigned [74]. The command line options prefixed with -X are nonstandard JVM options, some of them sharing the same name and functionality with other VM implementations [21]. I use such options to set the hotness level of the JIT, generate trace files as described earlier, disable SIMD and select the GC policy that the JIT should be using [91]. I set the hotness level only for testing purposes. The JIT will optimize array copying as and when required in the real-world scenario. The GC policy matters because the read barriers that are being handled by arraycopyEvaluator belong to Concurrent Scavenge alone. In our case, I provide a regular expression with "arraycopy", as it could be forward, backward or generic arraycopy. It is observed that the OpenJ9 build itself uses the arraycopyEvaluator optimizations from these options. The encounter with a debug counter can be stored in a file with the following environment variable option [74].

Listing 4.4: Saving arraycopy counters to a text file

```
export TR_DebugCounterFileName=debugcounters.txt
```

I disable inlining of methods in JIT to see the flow of instructions for testing purposes. There is an argument that can be passed to the JIT so that inlined methods are not utilised, as shown below to disable inline methods [73]. I use this option to see the flow of instructions from the assembly helper function from the assembly helper file.

All the added optimizations are applied with the 'veryHot' level of optimization. Since these optimizations happen during runtime, the optimization level is passed as a parameter to the JIT compiler when executing the Java program. To be able

Listing 4.5: Disabling inlining of arraycopy

```
-Xjit:disableInlining
```

Listing 4.6: Setting JIT optimization level

```
-Xjit:optLevel=veryHot
```

to generate reference arraycopy nodes going through read barrier checks for testing purposes, the Concurrent Scavenge garbage collection policy is used in the command line option. Read barriers exist only for the Concurrent Scavenge policy currently in OpenJ9. This option is set when running the Java application.

4.1.1 Specifications

Tables 4.1 and 4.2 show the specifications of the AArch64 and x86_64 machines that are used for testing and benchmarking. The AArch64 machine that is used is a Raspberry Pi (RPi), A72 based device, it comes at a low cost and consumes less power [65]. The CPU frequency of the x86_64 machine is set to 1360 MHz and AArch64's to 1300 MHz to make the results comparable with AArch64's. The x86_64 machine, Beelink Mini PC and AArch64 machine, Raspberry Pi are quad core with no hyperthreading. The x86_64 machine that is used is Beelink, a mini PC, that has specifications close to the RPi device, to give comparable results [18]. Raspberry Pi is a prominent System-on-chip (SoC) computer architecture with a small size, low price and minimized circuit complexity [95]. An SoC includes vital components, such as Central Processing Unit (CPU) and Graphics Processing Unit (GPU), on a single

Listing 4.7: Specifying GC policy

```
-Xgc:concurrentScavenge
```

Table 4.2: Specification of x86_64

Architecture	x86_64
Model	Intel(R) Atom(TM) x5-Z8350
Cores	4
Minimum CPU Frequency	480 MHz
Maximum CPU Frequency	1920 MHz
Locked Frequency	1360 MHz

chip [95].

DaCapo and BumbleBench benchmarks are used to evaluate the performance, at a variety of element sizes and array lengths, as mentioned in Chapter 5. DaCapo provides a real life application, with a variety of array size [57, 56, 59]. BumbleBench sub-benchmark ArrayCopy's results are focussed on copying of arrays alone. This gives a us a closer look of performance differences. The instructions used to copy elements are verified using trace files and the Perf tool. The memory is set to '-Xms2G -Xmx2G' on the JVM for all the DaCapo sub-benchmarks (described in Section 2.11.3) runs so that the same amount of memory is used across different runs and both the architectures. This means the upper limit and lower limit of memory is set to 2 GB. The results from the DaCapo Benchmark suite are an average of thirty runs with standard deviation, ten more than what Lengauer et al. suggest, after the outliers are removed [56]. More runs (such as 20 or more) are needed to be completed to ensure that the results are stabilized from JIT cache [56]. The BumbleBench Microbenchmarking test framework is run only ten times, as it includes a warm up phase. In each run of BumbleBench, the same program is run thirty-five times on average, hence I chose ten runs. Similar to the DaCapo Benchamrk suite's results, the outliers are removed, the mean of the BumbleBench score and their standard deviation are graphed. The concurrent scavenge GC policy is enabled for both DaCapo and Bumblebench runs.

Chapter 5

Results

This chapter discusses various methods of testing, their results and analyses derived from the results. The results obtained from running the Bumblebench microbenchmarking framework and the DaCapo benchmark suite on AArch64 are studied in this chapter.

5.1 OpenJ9 tests

OpenJ9 has tests that are run before merging new code with the master branch in the git repository, as a part of CI/CD. Commonly used tests include jenkins, sanity, alinux64 and jdk11 for AArch64. These tests are run to make sure the code does not run into any issues later. The procedure to run these tests is included in Appendix B. After running the OpenJ9 built-in tests, the passed tests include the following, to conclude that arraycopy works.

TestArrayCopy_0

TestArrayCopy_1

TestArrayCopy_2

TestArrayCopy_3

TestArrayCopy_openj9_none_SCC_0

TestArrayCopy_openj9_none_SCC_1

TestArrayCopy_openj9_none_SCC_2

TestArrayCopy_openj9_none_SCC_3

When class sharing is enabled for a class and the VM needs to load the same, then the VM looks for it in the Shared Class Cache (SCC) [87]. The SCC includes data related to JIT and AOT compiled code, bootstrap classes and others for improved startup time [87]. The SCC usage can be disabled by using the following command line option [88].

Listing 5.1: Disabling Shared Class Cache

```
-Xshareclasses:none
```

As the JIT compiled code can affect the performance of arraycopyEvaluator, tests are run with SCC turned off in the above cases with 'TestArrayCopy_openj9_none_SCC_' for testing purposes. This is because the SCC may contain the arraycopy optimizations even when it turned off in the JIT.

5.2 Benchmark Results and Analysis

The benchmark suites are run with optimization level veryHot, to trigger all possible arraycopy optimizations, to see a difference in performance wherever possible. On levels less than veryHot, the arraycopy nodes are not generated and the optimization is not applied unless there are enough calls or iterations of the same block to copy arrays. The performance is gauged for builds with and without the arraycopy optimization in benchmarking using BumbleBench and DaCapo.

The arraycopy optimization was disabled for the 'No opts' case in the graph by removing the below flags from 'J9CodeGenerator.cpp' in the OpenJ9 code base. This will only disable the arraycopyEvaluator calls, but continue to optimize the

rest of the code at the 'veryHot' level in the JIT compiler. This allows us to check the change in the performance brought by the arraycopyEvaluator method alone.

Listing 5.2: Flags to disable arraycopy

```
    cg->setSupportsPrimitiveArrayCopy();
    cg->setSupportsReferenceArrayCopy();
```

5.2.1 BubmbleBench Microbenchmark Tool

BumbleBench has an option to switch the direction of array copying between forward and backward, and it includes arrays with various data types with customizable array and copying lengths. The direction, array length and copying length are provided as a commandline option as shown below.

Listing 5.3: BumbleBench command line options

```
java -DBumbleBench.backwardArrayCopy -jar BumbleBench.jar [
   Benchmark name]
java -DBumbleBench.arrayLength=<value> -jar BumbleBench.jar [
   Benchmark name]
java -DBumbleBench.batchTargetDuration=<value> -jar
   BumbleBench.jar [Benchmark name]
java -DBumbleBench.copyElements=<value> -jar BumbleBench.jar [
   Benchmark name]
```

Listing 5.4: BumbleBench Framework Core

```
attempt(float targetScore) Method:


// The targetIterations value is at least one.
float _unpausedFraction = 1F;
```

```
iterationRate = targetScore * Math.max(_unpausedFraction,
   Options.MIN_UNPAUSED_FACTOR);
targetIterations = ( iterationRate * Options.
   BATCH_TARGET_DURATION ) / 1000;
targetIterations = Math.max(targetIterations, 1);
measuredIterations = doBatch(targetIterations) // doBatch
   returns the number of Iterations
measuredRate = measuredIterations * 1.0e+9f / unpausedTime ;
return measuredRate; // Score


runAttempt(boolean lowball) Method:


float under = _estimate * (1-_uncertainty/2);
float over  = _estimate * (1+_uncertainty/2);
float target = lowball ? under : over; // Traget score is
   calculated
float result = attempt(target);
// The result is then used to decide if its a lowball or
   highball. (updates guessWasCorrect)
if (_recentPeak == _maxPeak) {
        if (under <= _maxPeak && _maxPeak <= over)
                _maxPeakUncertainty = Math.min(
                   _maxPeakUncertainty,_uncertainty);
}
if (newEstimateWasSpecified)
    float impliedUncertainty = Math.abs(oldEstimate - result)
       / target;
_uncertainty *= guessWasCorrect? CORRECT_GUESS_ADJUSTMENT :
   INCORRECT_GUESS_ADJUSTMENT; // Uncertainity
```

```
return guessWasCorrect; // Verifies if it is Highball or
   Lowball

run Method:

// Warmup phase
while (!runAttempt(HIGHBALL)) { if (currentTime > maxEndTime)
   break; }
while (!runAttempt(LOWBALL))  { if (currentTime > maxEndTime)
   break; }

// Ballpark phase is to make sure the results are stable
    for (int i = 0; i < BALLPARK_ITERATIONS; i+=2) {
                while (!runAttempt(HIGHBALL)){}
                while (!runAttempt(LOWBALL)){} }

// Final phase used to calculate the actual score
    for (int i = 0; i < FINALE_ITERATIONS; i+=2) {
        while (!runAttempt(HIGHBALL)){}
        while (!runAttempt(LOWBALL)){} }
// The variables like score, uncertainity at this point are
   the final values.
```

Table 5.1 shows the default values of options on BumbleBench. The score in the graph is related to the number of operations BumbleBench guesses to finish in a target duration [77]. However, the actual value can be higher or lower than the target score. The goal is to match the total elapsed time with BumbleBench's estimated target. The score begins from a low ball and eventually reaches highball, and the highest score ever achieved is called MaxPeak. The score that lies just above the

Table 5.1: Default values of options on BumbleBench.

Option	Default value
backwardArrayCopy	false
arrayLength	8192
batchTargetDuration	1000 ms
copyElements	4000
CORRECT_GUESS_ADJUSTMENT	0.6F
INCORRECT_GUESS_ADJUSTMENT	1.2F
MIN_UNPAUSED_FACTOR	0.2F
LOWBALL	true
HIGHBALL	false
_uncertainty	0.5F
_estimate	1F

highest low ball value is called Peak. During each run the values _uncertainty and _estimate are updated, which helps in calculating the final score. The uncertainity factor is greater than zero for a few Bumblebench sub-benchmarks (marked as error bars in the graphs), which is the difference between highest and lowest outliers. The higher score in the following graphs indicates that the performance is better.

5.2.2 Initial Results of BumbleBench

Figures 5.1, 5.2, 5.3 and 5.4 are with the default copying lengths and array lengths for 'ArrayCopy' sub-benchmarks are as shown in Table 5.3. Table 5.2 shows the modifications made to the arraycopy benchmark in BumbleBench, to gauge the performance of inlined methods and assembly functions. Figures 5.5, 5.6, 5.7 and 5.8 have arrays with same number of copying lengths in bytes. The details are shown in Table 5.4. The default notation for copying arrays in BumbleBench is as shown here, with length passed as a variable. Using a variable in BumbleBench provides the ability to change the number of elements to be copied from command line options. Since inlined methods are formed using a constant length, the values had to be hard-coded in the benchmark. The BumbleBench framework uses the System.arraycopy

Table 5.2: BumbleBench Mircobenchmark modifications

Forward arraycopy	
Inlined method	System.arraycopy(srcArray, srcpoint, dstArray, dstPoint, 4000)
Assembly function	System.arraycopy(srcArray, srcpoint, dstArray, dstPoint, COPY_ELEMENTS)
Backward arraycopy	
Inlined method	System.arraycopy(srcArray, srcpoint, dstArray, dstPoint, COPY_ELEMENTS)
Assembly function	for loop with constant length 4000

Table 5.3: Array copying with same lengths

Data type	**Array length bytes**	**Number of bytes to be copied**
Object	32768	16000
Long	65536	32000
Int	32768	16000
Char	16384	8000
Byte	8192	4000

Table 5.4: Array copying with same number of bytes

Data type	**Array length**	**Number of elements to be copied**
Object	2048	1000
Long	1024	500
Int	2048	1000
Char	4096	2000
Byte	8192	4000

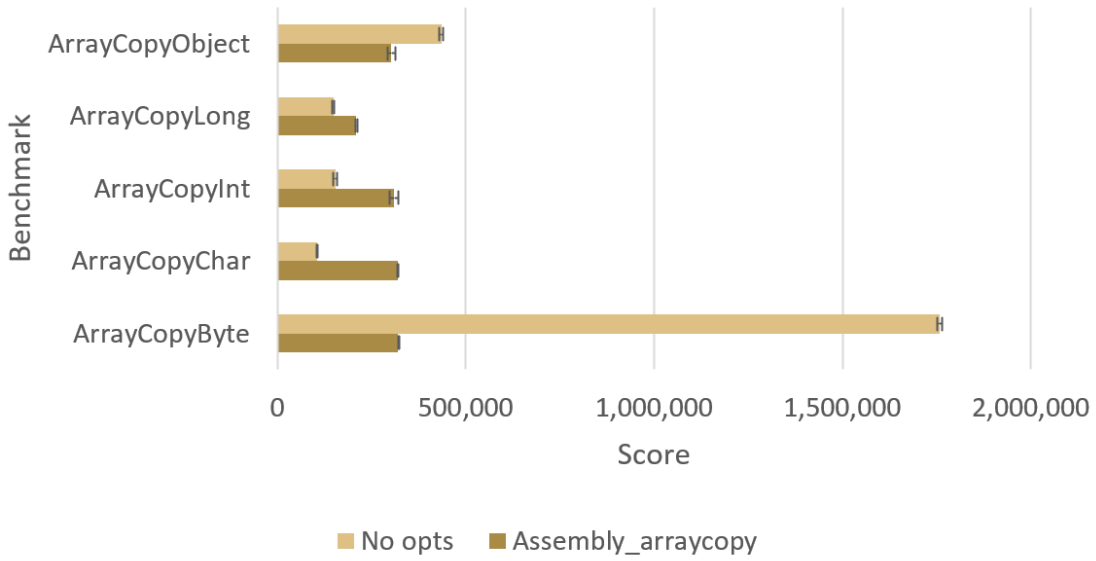

Figure 5.1: Forward arraycopy using the length variable

method with the parameters as shown below.

```
System.arraycopy(srcArray, srcpoint, dstArray, dstPoint,
   COPY_ELEMENTS)
```

When the ArrayCopyObject sub-benchmark is run, calls to 'forwardreferencearraycopy' and 'doCopyContiguousForward' methods can be seen using Perf, which involves copying elements by accessing reference fields. The size of an Object is four bytes in all the cases of BumbleBench benchmarking. The assembly function '__arraycopy' is called when optimization is enabled. In the case of forward arraycopy, passing a variable as the length argument to 'System.arraycopy' triggers an '__arraycopy' call when optimization is enabled. If instead, a value is passed as a constant, it is inlined. In Figure 5.1, the optimized performance is better than with no optimization in only ArrayCopyChar, ArrayCopyInt and ArrayCopyLong sub-benchmarks. The reason behind Byte performing better is explained later in this section. It is observed that the bar lengths in Figure 5.1, of Int and Object almost match because of their matching alignment. A slight variation is expected as there

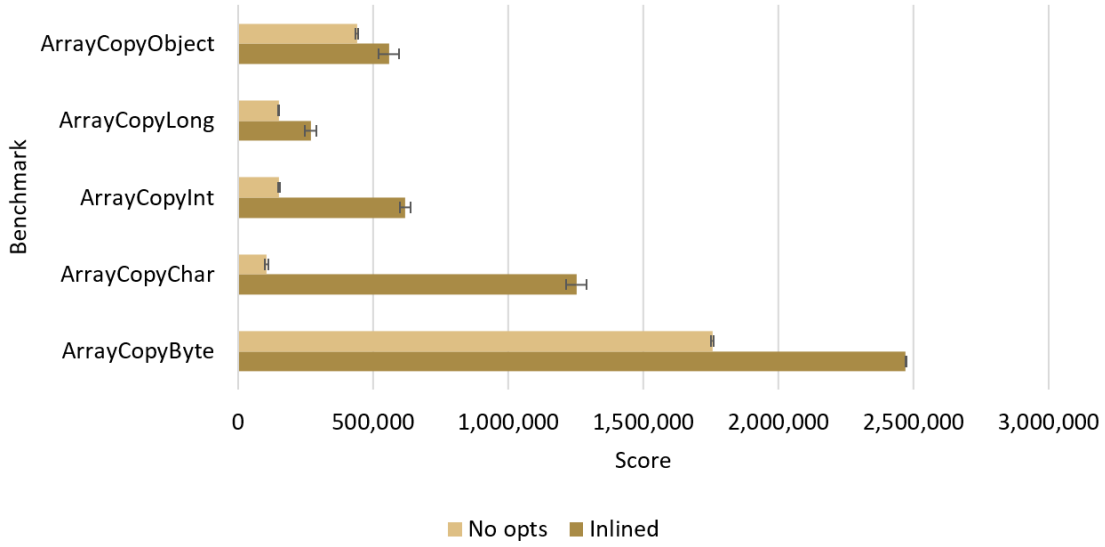

Figure 5.2: Forward arraycopy using constant length

are minor variatons of performance each time a program is run. The array length (8196) and copying length (4000) are the same across all the data types. Hence, the total number of bytes to be copied and the array length bytes differ for each data type. For this reason, the following graphs show copying the same number of bytes are compared for different data types. By analyzing the effectiveness among the dataypes, larger data type sizes function optimally as seen in Figure 5.5. This is because the array is copied element by element, hence bigger data types perform better due to their larger size in a single element. In Figure 5.2, the inlined forward copying performs better for all the data types due to the bulk of elements being copied in a single instruction. The passing of a constant to 'System.arraycopy' does not trigger the inlined method in backward copying. This could be due to a compilation barrier on backward arraycopy but not forward arraycopy. From this we can conclude that the JIT compiler sees backward arraycopy nodes differently when compared to forward arraycopy nodes. When the arraycopy nodes are created in the ValuePropagation.cpp file, due to their different natures, forward and backward copying go through validation checks differently. The dissimilarities occur in source

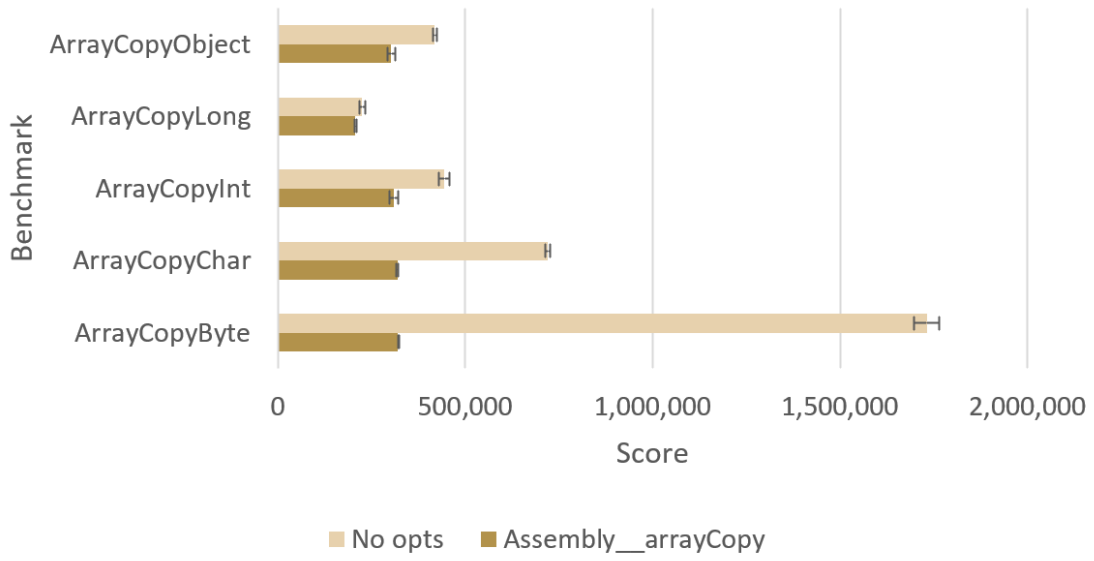

Figure 5.3: Backward arraycopy using the length variable

and destination arrays and lengths validation due to the direction of copying. One such example is discussed later in this chapter. Hence, the properties of forward and backward arraycopy nodes are different. The assembly method '__arraycopy' is triggered whether a constant or a variable is passed to 'System.arraycopy'. To trigger an inlined method for backward arraycopy, a 'for loop' with a constant is used. In the no optimization case of backward arraycopy, a generic 'memcpy' method is called in the case of byte. For the rest of the data types, 'alignedBackwardsMemCpy' method is called that is tailored for backward copying. This method calls various 'memcpy' methods that are designed for their respective data types. The 'memcpy' helper methods called in the 'no optimization, length passed using a variable' case show optimal effectiveness, as seen in Figure 5.3 due to vector usage. This performance is better than 'no optimization, length passed using a variable' case of forward copying. There are optimizations involved with 'memcpy' that give better efficiency, such as using vectors in this case. The GCC (GNU Compiler Collection) is capable of performing optimizations such as auto-vectorization [92, 93, 8]. In Figure

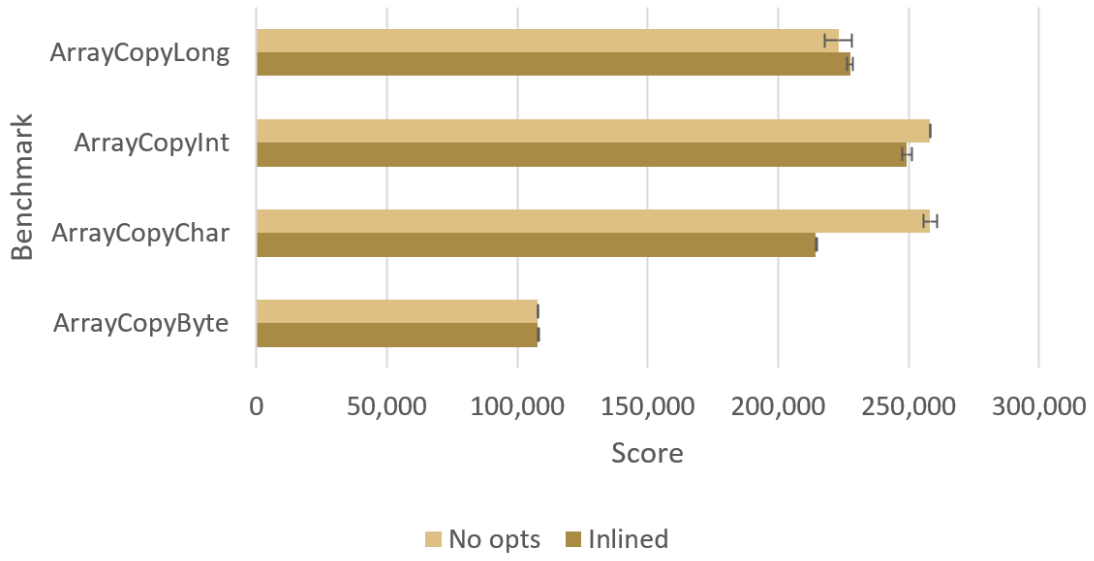

Figure 5.4: Backward arraycopy using constant length

5.4, the scale is different from the rest of the graphs for better visualization, hence the differences can be clearly seen. These differences are almost negligible when the number of bytes are the same, as seen in Figure 5.8, due to their scale. Figures 5.5, 5.6, 5.7 and 5.8 show the performances of different data types operating on the same number of data types. For a better understanding of the performance and improvements, in each sub-benchmark of ArrayCopyByte, ArrayCopyChar, ArrayCopyInt, ArrayCopyLong and ArrayCopyObject, the same number of bytes is used for arrays and copying lengths. Table 5.4 shows the array lengths and copying lengths used for these data types.

In the case of the backward arraycopy, the JIT compiler does not generate arraycopy nodes for Object type when a 'for loop' is used. Hence, in Figures 5.4 and 5.8 'ArrayCopyObject' is omitted.

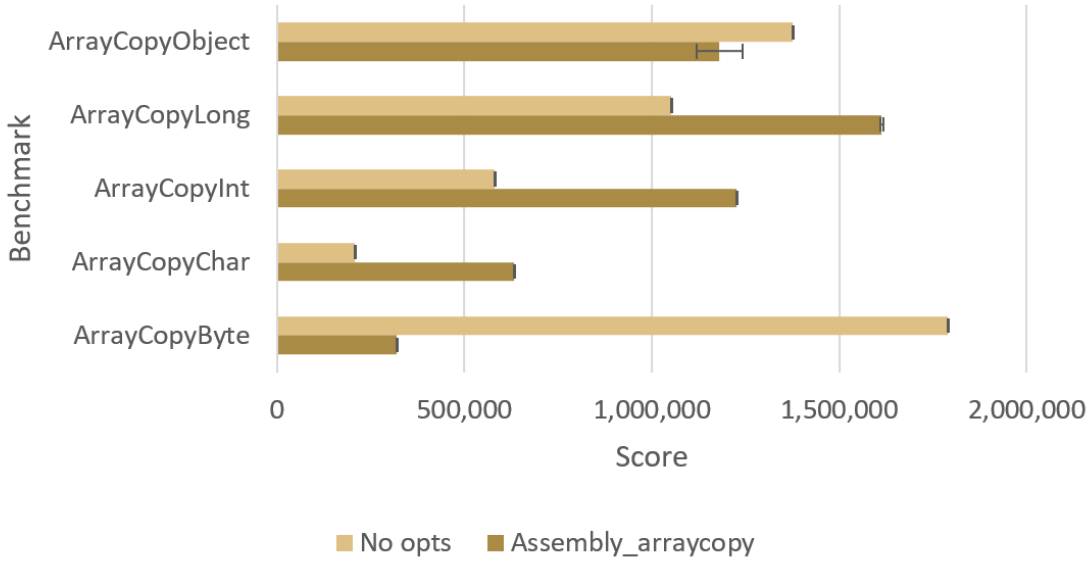

Figure 5.5: Foward arraycopy using the length variable

5.2.3 Using the Perf tool

To observe what instructions are used in the case of 'memcpy', in which byte has better performance compared to other data types in the case of forward copying, we use utility tools like javap. The javap tool comes with the JDK, allowing us to see information regarding methods, that looks similar to IL. However, this does not include information regarding the underlying native instructions used for copying. Hence, the assembly notation for each method can be spotted with command line options such as '-XX:+PrintAssembly' in HotSpot [78, 79]. However, OpenJ9 does not have support to dissemble code from command line options, hence we rely on the Perf tool. Perf is a profiling tool used to gain information on events of a program [66, 65]. Perf is capable of mapping JIT compiled method symbols to actual addresses using '-Xjit:perfTool' in the command line [89]. Using the commands mentioned in Perf (Appendix C.1.5) , we can see only the assembly code of non-JITed code, due to the limitations of JIT [89]. With the optimization level 'veryHot' methods like 'doBatch' in BumbleBench, where copying loop exists, is JIT compiled even

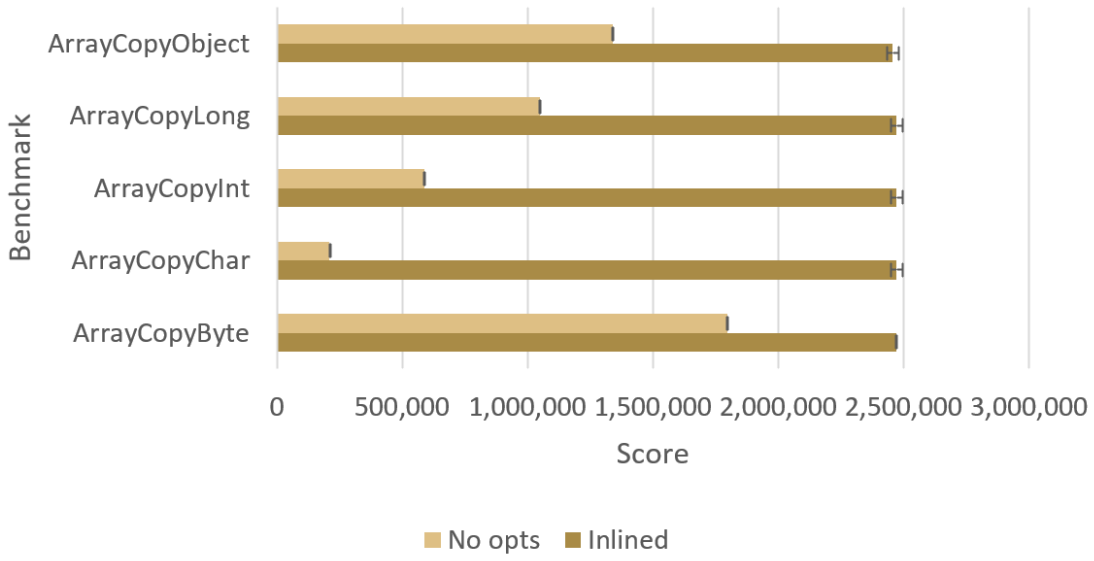

Figure 5.6: Foward arraycopy using constant length

without the arraycopy optimization added. The 'memcpy' function and its assembly instructions are hidden in the JITed code. For JITed code, the assembly instructions could not be mapped to (post JIT) the JITed code. Hence, we make use of the JVMTI agent in Section 2.11.4 to map the instructions, so the assembly of JITed methods can be seen. Hence, to see the assembly instructions of 'memcpy', the instructions in Appendix C.1.5 are used. In Figures 5.2 and 5.4 it might look like ArrayCopyByte's performance is better than the other data types. Hence, we equalize the number of bytes and run BumbleBench to see if that is true.

Based on the data obtained from Perf, it is observed that in the case of copying arrays of byte types in the forward direction, load and store pairs are used. Load and store pairs use a pair of 64-bit registers, which can load and store 128 bits at a time. However, it can be seen from Figures 5.2 and 5.4 that the performance is better with vector usage because multiple elements are copied in a single instruction. In the sub-benchmark results other than ArrayCopyByte in forward direction, elements are copied one at a time. In Figure 5.1, the optimized ArrayCopyObject result is a little

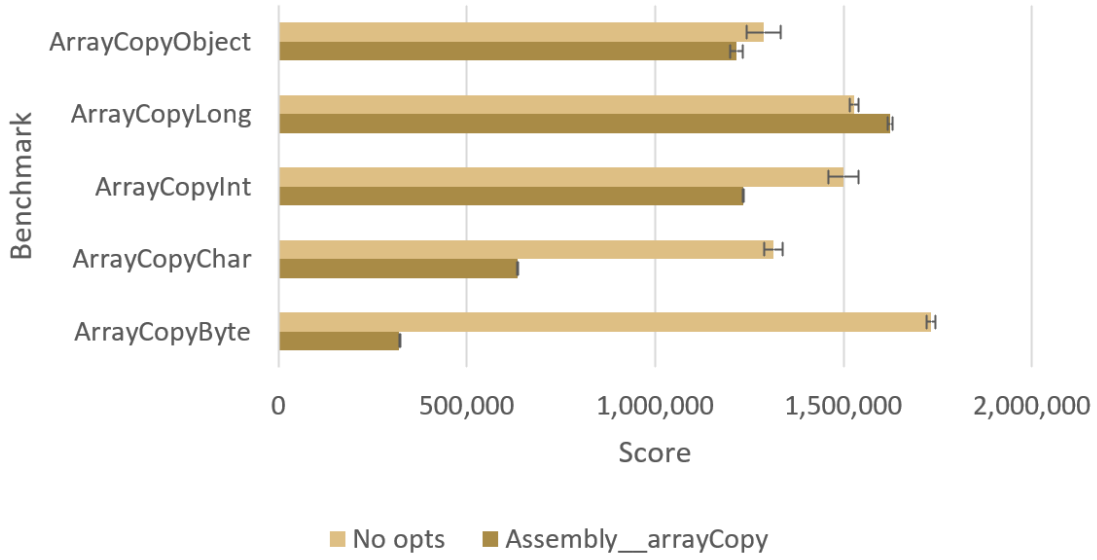

Figure 5.7: Backward arraycopy using the length variable

worse the non-optimized version. It was observered that ArrayCopyObject used vectors partially. It used 'forwardreferencearraycopy' and 'doCopyContiguousForward' methods when TR_writeBarrierBatchStore runtime helper was called. However, 'doCopyContiguousForward' was directly used in the no optimization case, causing it to have better performance over the optimized version in Figures 5.1 and 5.5. In Figures 5.3 and 5.7, the high performance of backward arraycopy with no optimization is due to use of vector loads and stores by memcpy. In '__arraycopy' function of arraycopyEvaluator, vectors cannot be used, as the size to be copied is not known. Figures 5.4 and 5.8 show similar performance with and without optimization. It is found from Perf that although vectors are used for copying data in the optimized version, they are skipped due to a store check performed by OpenJ9's optimizer. The code that is triggering these instructions is studied later in this thesis. In both the cases, vectorization is avoided and this is investigated later on in this chapter. Improving assembly functions in the future for backward copying in the assembly function by studying 'memcpy' can provide more efficiency in terms of time. In Fig-

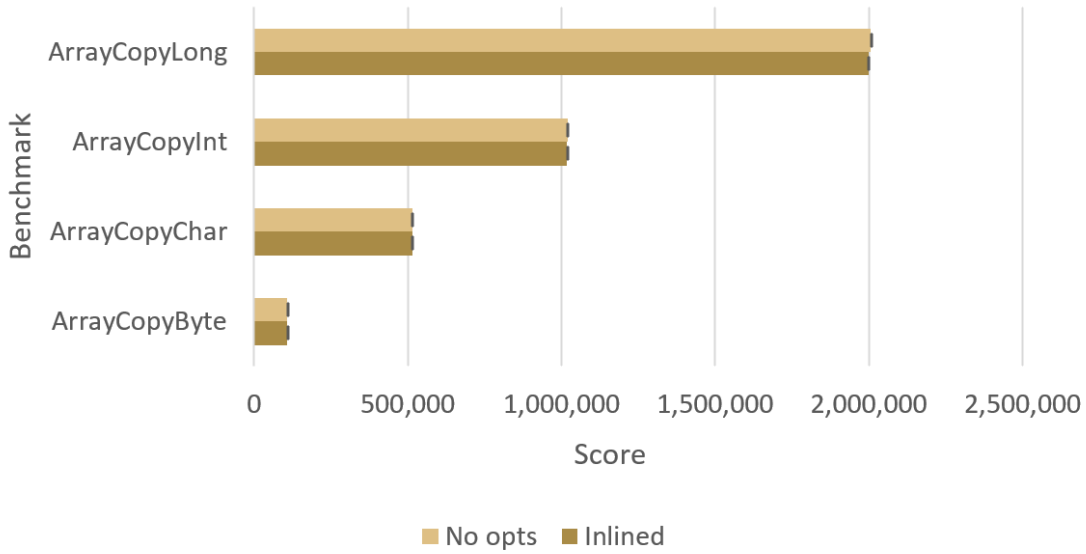

Figure 5.8: Backward arraycopy using constant length

ure 5.8, we can see that the performance of the inlined backward arraycopy is the same as the backward version of the 'memcpy' method. Table 5.5 shows the number of arraycopy calls made in each sub-benchmark of DaCapo, which detemines that jython could have a greater variety of arrays. Figure 5.9 demonstrates the performance with existing arraycopy optimizations on x86_64 relative to the performance after disabling optimizations with the same configurations. The results in Figure 5.9 show that better results are obtained after adding arraycopy optimizations, except for the cases of xalan, luindex and fop. The sub-benchmarks where a decrease in performance is seen, it is speculated that while JIT optimizations take place, it is missing memcpy optimizations. The AArch64 platform is expected to have similar results to that of x86_64. To identify the outliers in the data, the following approach is integrated. A statistician named John Tukey suggested the following definition to identify outliers [94]. To identify the outliers in the generated data, the first and third quartile are calculated. Using these two values, the lower and upper bounds are calculated. If the elapsed time of a benchmark lies outside of the values between

Table 5.5: The count of arraycopy method calls

Sub-benchmark	Number of arraycopy calls
avrora	106
fop	63
h2	303
jython	780
luindex	99
lusearch	101
lusearch-fix	49
pmd	29
sunflow	102
xalan	84

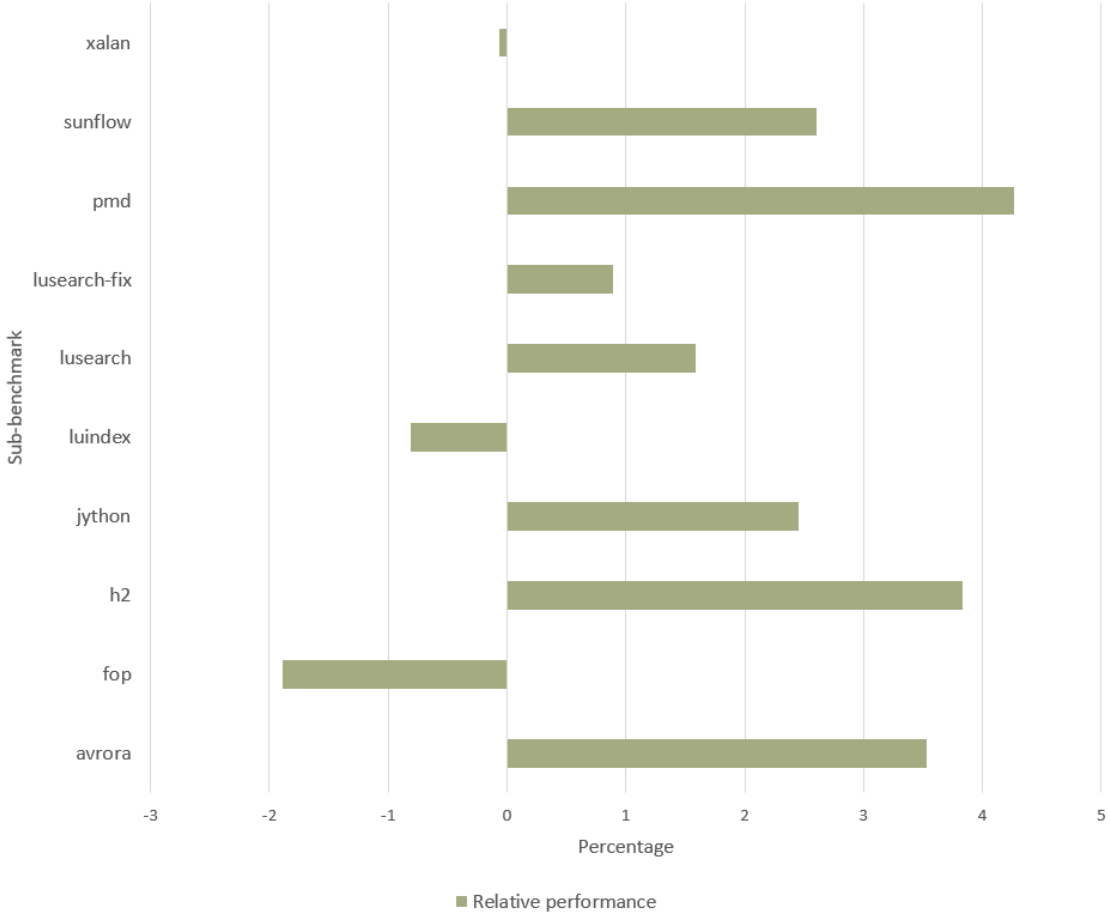

Figure 5.9: Relative performance on x86_64

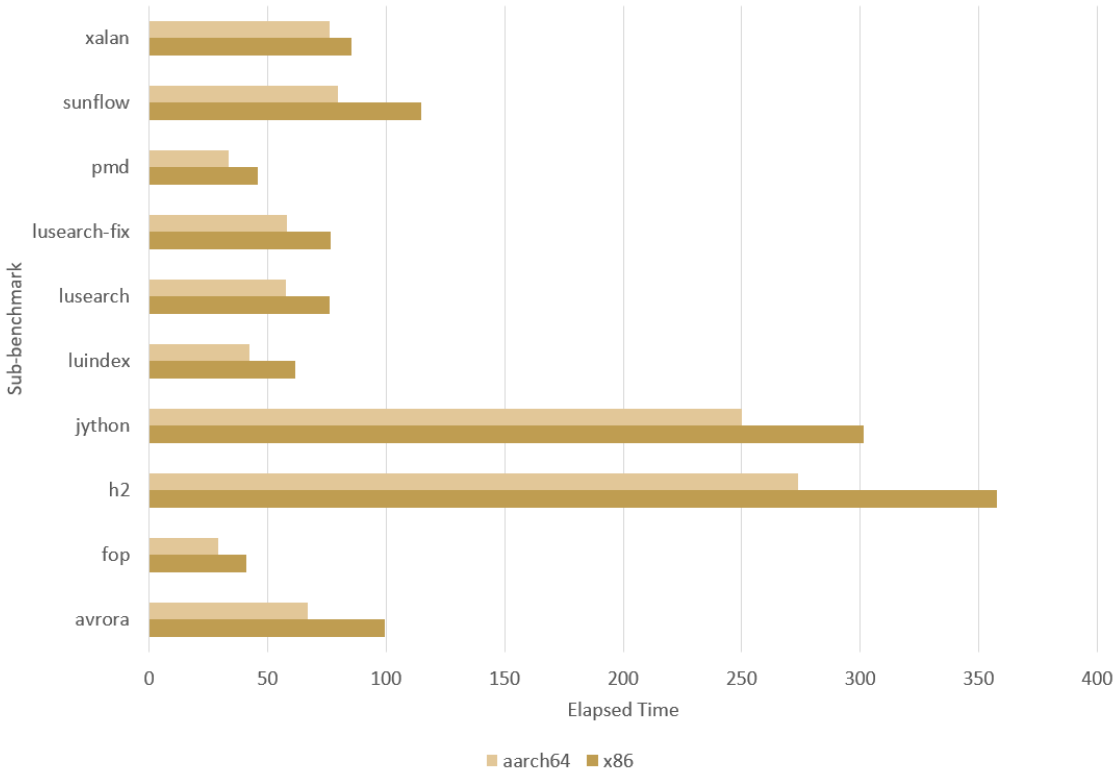

Figure 5.10: Performance on x86_64 vs. AArch64

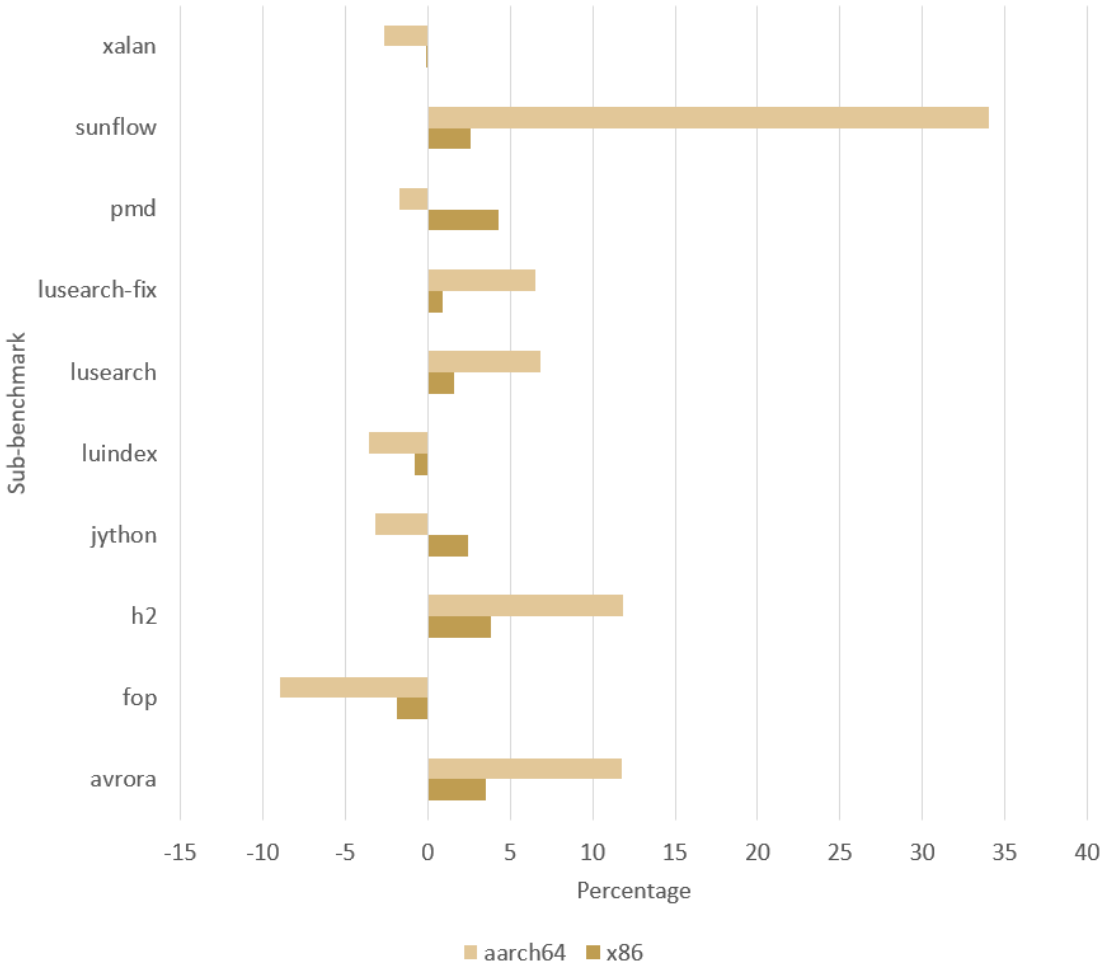

Figure 5.11: Relative performance on x86_64 vs. AArch64

lower and upper bounds, it is identified as a outlier. Since the data was entered in a Microsoft Excel Worksheet to generate the graphs, their in-built formulas are used to calculate the lower and upper bounds.

The higher standard deviation in few sub-benchmarks of DaCapo and BumbleBench is attributed to the performance bottleneck caused by the arrays stored in exactly one page frame or fitting multiple complete page sizes.

Inlining arraycopy significantly improves the performance of forward arraycopy while maintaining backward arraycopy's performance the same. To enhance the performance of non-inlinable backward copying of arrays, native methods similar to that of System.arraycopy can be added as a part of future work. This is to ensure that during the runtime, when the size to be copied is known, appropriate calls to the existing optimized memcpy functions can be made. Figure 5.10 shows a comparision of DaCapo results of x86_64 and AArch64. As seen in Figure 5.11, the performance improvement on AArch64 in xalan and sunflow is better than the improvement on x86_64. In the AArch64 results, the discrepancies in optimal efficiency in benchmarks like lusearch-fix, lusearch, h2 and fop could be due to backward copying using variables or forward copying of byte type arrays using variables.

The decrease in performance caused by the application of arraycopyEvaluator on x86_64 is speculated to be due of the better performance of 'memcpy' rather than the non inlined arraycopyEvaluator version. This is similar to the performance observations on the AArch64. Hence, to further improve the performace of arraycopyEvaluator, 'memcpy' like implementation can be utilized in the non inlinable part.

Figures 5.12 and 5.13 show similar performance, with and without SIMD. This is because inline arraycopy and the memcpy function both use vectors that use Floating Point registers. Vector Floating Point Registers (VFR) may be used for copying 128 bits that are not necessarily floating point values [80].

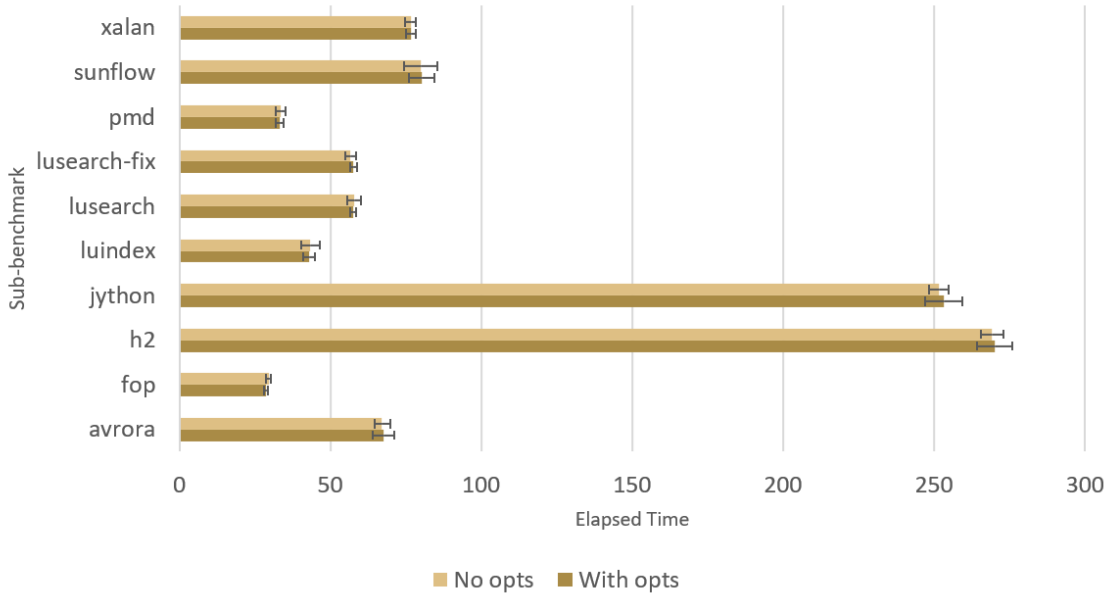

Figure 5.12: Performance on x86_64 vs. AArch64 with SIMD disabled

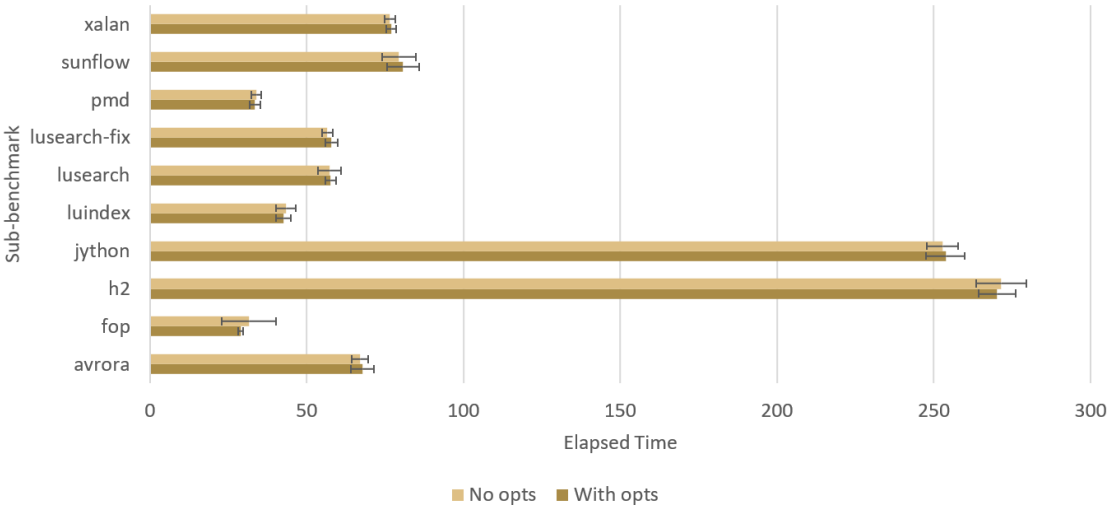

Figure 5.13: Performance on x86_64 vs. AArch64 with SIMD enabled

When optimizations like inline copying of arrays are applied during JIT compilation, it also opens up possibilities of further optimization [16]. This is possible when the inlined method is a part of code that is further optimizable by the JIT [16]. This reduces overhead factors such as memory footprint, when compared to C/C++ library functions such as 'memcpy'. Memory profiling tools like jps and jmaps that ship with OpenJDK have limited support with OpenJ9, which is insufficient for assessing memory consumption at this point. Perf can also be used for profiling memory but it does not have support on AArch64 yet.

Load/store pairs can be used on Vectors as well [80]. However, there are some restrictions such as no option to sign extend, and the address being loaded from a 64-bit register [80]. Hence, vectors with load/store pairs were avoided in this thesis. In order to improve the performance of non inlinable copying, memcpy method should be utilized, as was the case with no optimization.

5.2.4 Subrange Overlap test

After comparing the results in inlined backward arraycopy with and without optimizations, there was no observed improvement in the optimized version. Further investigating Perf with JVMTI agent showed that the vector instructions existed in the assembly code to be executed. However, that block containing inlined vector instructions was skipped with a branch instruction. With the help of Perf and the trace file, the condition and branch instructions were traced. It is found that in the IdiomTransformations.cpp file of OpenJ9 JIT optimizer, checkArrayStore function is called that creates a guard if the source and destination arrays are the same. This is preventing the JIT optimization for inlined backward arraycopy. Disabling this check boosts the performance of inlined backward arraycopy.

The Streaming SIMD Extensions (SSE) in x86_64 copies at least 4 bytes at a time.

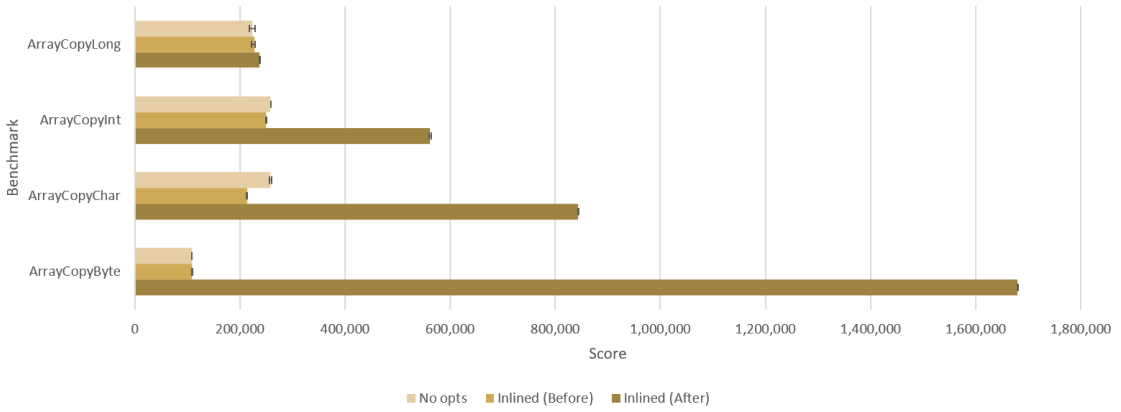

Figure 5.14: Backward arraycopy using constant length

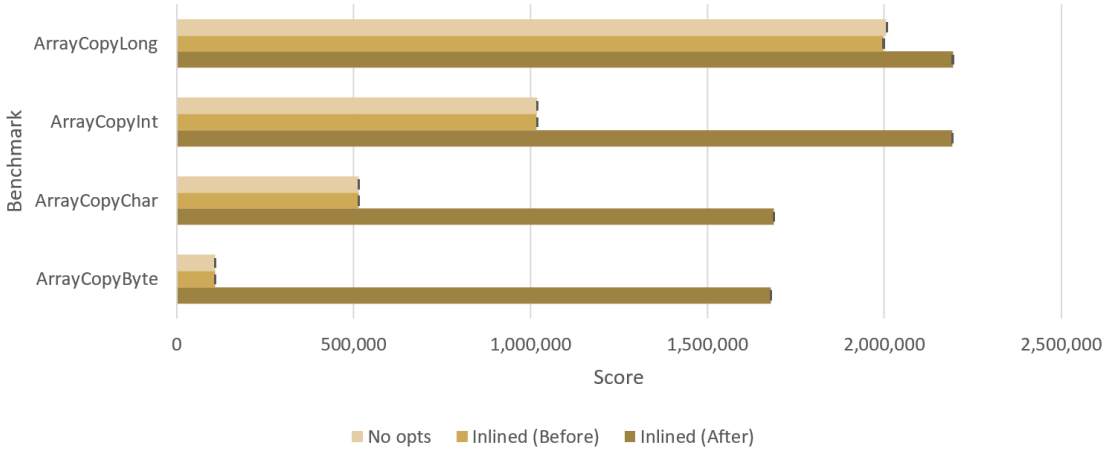

Figure 5.15: Backward arraycopy using constant length

Hence, in the optimizer, there are checks such as Subrange Overlap test that disables optimization. The Subrange Overlap test had to be turned off for AArch64. The benchmark results after turning this off can be seen in Figures 5.14, 5.15, 5.16 and 5.17. After studying the values in Figures 5.14 and 5.15, it is speculated that a check similar to Subrange Overlap test meant for x86_64 is causing a slow down for Byte and Char. This is possible because of x86_64 not being able to efficiently handle less than 4 bytes of size at a time. Identifying this and disabling it for AArch64 can give the same BumbleBench score for all the datatypes. In Figure 5.16, we can see an improved performance in sunflow, lusearch, lusearch-fix, h2 and avrora. The same is

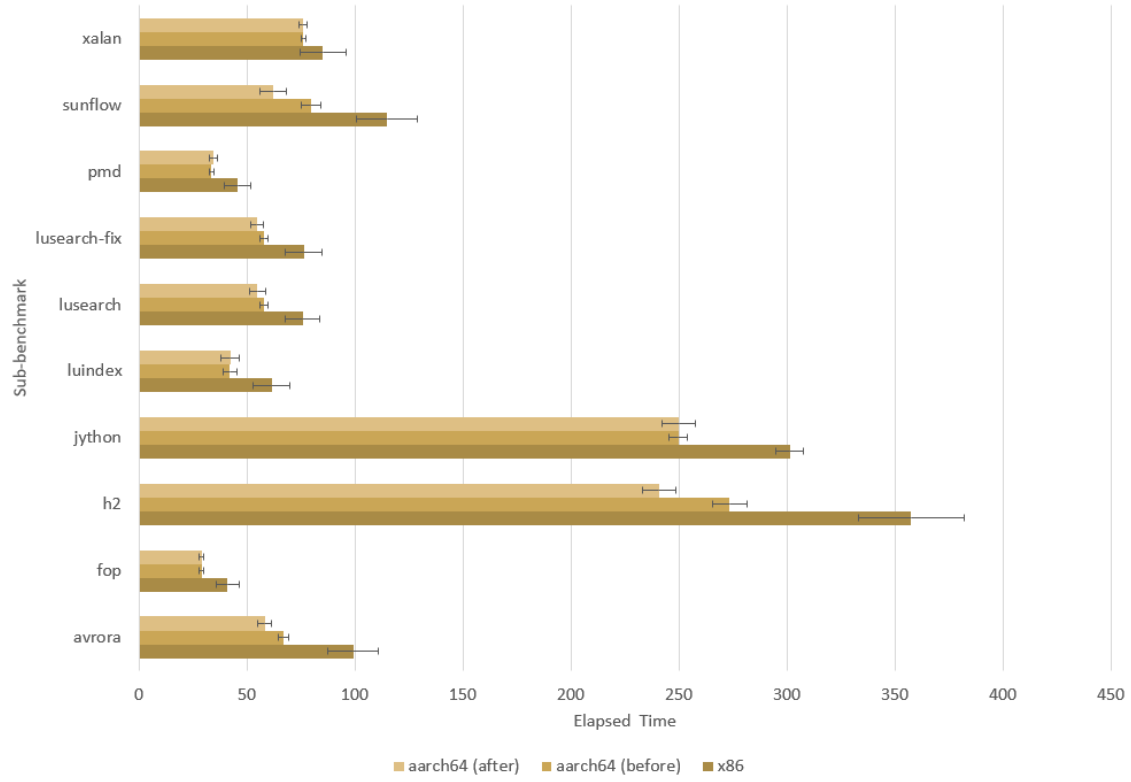

Figure 5.16: Performance on x86_64 vs. AArch64

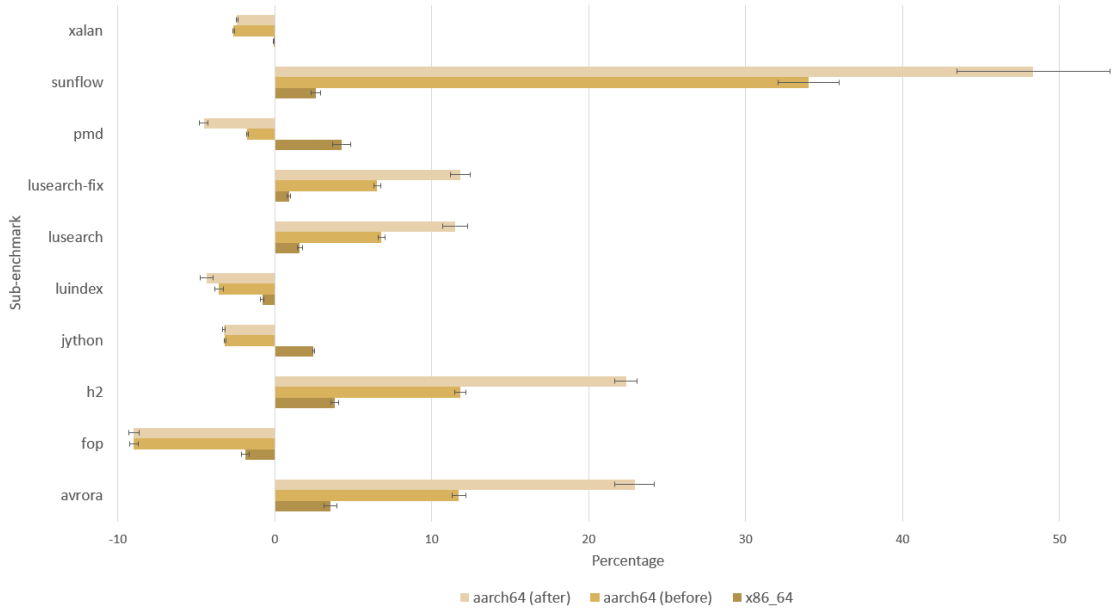

Figure 5.17: Relative performance on x86_64 vs. AArch64

reflected in Figure 5.17.

Considering all the benchmark scores of individual data types, ArrayCopyChar in Figure 5.2 shows ten times better performance. In comparision to x86_64, AArch64's performance in the sunflow sub-benchmark of DaCapo shows highest improvement of 45.7%.

The Bumblebench benchmark results helped us study the performance with various data types and array sizes. The DaCapo benchmark results aided in understanding the performance of AArch64 compared to x86_64. Thus the vector usage and inlining proved to be efficient initially on forward copying only. Later on, when a redundant test was disabled, the backward copying showed a better performance. Also, a further scope to improve the performance of OpenJ9 on AArch64 is noted.

Chapter 6

Future Work

This section explains the possible extensions to the current work to add optimizations with reference to arrays. The missing optimizations in the AArch64 code, compared to other architectures codebase that can be added to improve the performance are discussed.

The JIT compiler must be made to see the forward and backward arraycopy nodes the same way, so System.arraycopy and for loop yield similar results using both the nodes. The differences in forward and backward arraycopy nodes are in the checks performed for the same copying pattern. As we saw in Chapter 5, System.arraycopy with constant length triggers inlined copying in forward case, where as in backward copying it does not. If generation of nodes in forward and backward copying are consistently similar, it could broaden the range of inlinable patterns. The 'memcpy' method calls need to be incorporated in the arraycopyEvaluator, so it can have the advantage of the better performance of 'memcpy' instead of calling the assembly functions where one element is copied at a time. In addition to that, the following evaluators with patterns similar to loop copying need to be added on AArch64:

- arraytranslateEvaluator()
- arraycmpEvaluator()

- arraysetEvaluator()

The commonly used memory functions are memcpy, memcmp and memset. The TR-JIT solution is to recognize loops of particular patterns and map it to an evaluator method that uses platform specific code generation support. These calls are arrayset to set values, arraycmp to compare arrays, arraytranslate to type cast arrays, array-TranslateAndTest to scan arrays. The memory instructions used by arraytranslate in Z System are:

- TROT - translate one-to-two
- TRTO - translate two-to-one
- TROO - translate one-to-one
- TRTT - translate two-to-two

These methods can possibly utilize vector support to move, clear and compare elements for a better performance. In the AArch64 codebase, similar and suitable instructions need to be added to have arraytranslate support. Similar to arraycopy, some parts of these evaluators may be inlinable. Identifying and optimizing loop patterns that are discussed in this section can yield better outcome.
To summarize, more work around non-inlinable code of backward copying and other loop patterns to boost OpenJ9's potential on AArch64 are possible.

Chapter 7

Conclusions

Overall, the performance of arraycopy has shown significant advancement on inlined methods. Using VFP registers for copying arrays has been proven to be efficient. A significant improvement can be seen in the BumbleBench benchmark results. The DaCapo benchmark results show improvements of up to 45.7%.

The arraycopyEvaluator function was added because the JIT compiler was able to identify the copying pattern of arrays and inline them. While diverting the flow from calling memcpy to inline copying of arrays, first it was checked if copying arrays is inlinable. Hence, we also added assembly functions to the JIT, for the case of when copying of arrays cannot be inlined. The length to be copied was identified during compile time, that can be stored in a variable or as a constant value. Copying of an array was inlinable when the length to be copied was a constant. The assembly functions that were used when array copies could not be inlined, were slower than memcpy in few cases. Since memcpy is a function called during runtime, it knows the size even though the length to be copied was stored in a variable.

The different characteristics of forward and backward copying were studied. We provided a comparison of performance of various data types with constant and variable lengths. We used the BumbleBench microbenchmarking test framework to

calculate the efficiency of inlined arraycopy and the assembler functions versus the non-optimized version. Perf was utilized to determine the underlying functions used for copying arrays in the cases where inlining methods and assembly methods were not applied. This tool was also used to study the assembly notation of JIT-compiled methods. The differences in efficiencies in the instances of variable and constant copying lengths were distinguished. The advantage of the arraycopy implementation on AArch64 was compared to x86's improvement on existing changes. AArch64's performance improved multiple folds in sub-benchmarks of DaCapo: avrora, h2, lusearch, lusearch-fix and sunflow compared to x86_64. The decrease in elapsed time of sub-benchmarks fop, jython, luindex, pmd and xalan were not as oustanding as the previously mentioned sub-benchmarks. This can be attributed to the backward copying of non-inlined arraycopy and not making use of the already optimized 'memcpy' method.

We discussed further scope of improvements around loops in Chapter 6. In conclusion, this work set a step towards loop optimizations in the OpenJ9 JVM on AArch64.

Bibliography

[1] Beginners Book, *Introduction to Java Programming*, available at `https://beginnersbook.com/2013/05/java-introduction/`. Access date: 14th September 2020.

[2] Daryl Maier, *AArch64 Early Access in OpenJ9 0.20.0*, available at `https://blog.openj9.org/2020/04/24/aarch64-early-access-in-openj9-0-20-0/`. Access date: 14th September 2020.

[3] IBM Support, *JIT Problem Determination for IBM SDK using -Xjit*, available at `https://www.ibm.com/support/pages/jit-problem-determination-ibm-sdk-using-xjit`. Access date: 3rd February 2022.

[4] Eclipse, *Eclipse OMR*, available at `https://projects.eclipse.org/projects/technology.omr`. Access date: 14th September 2020.

[5] Peter Verhas, *Java 9 Programming By Example*, Packt Publishing; 2017. Access date: 26th January 2022.

[6] Angela Lin, *Building Your Own Language Runtime*, Proceedings of the 25th Annual International Conference on Computer Science and Software Engineering, CASCON '15, Markham, Canada, pp. 296–298.

[7] James A. Langbridge, *Professional Embedded ARM Development*, John Wiley & Sons, Incorporated, 2014. ProQuest Ebook Central,

https://ebookcentral.proquest.com/lib/unb/detail.action?docID=1576332.

[8] Nabil Hallou, Erven Rohou, and Philippe Clauss, *Runtime Vectorization Transformations of Binary Code*, Int. J. Parallel Program. 45, 6 (December 2017), pp. 1536–1565. DOI:https://doi.org/10.1007/s10766-016-0480-z.

[9] Eclipse, *Eclipse OMR*, available at https://developer.ibm.com/depmodels/cloud/projects/eclipse-omr/r. Access date: 14th September 2020.

[10] Open Source, *Eclipse OMR and OpenJ9*, available at https://github.com/eclipse/openj9-omr. Access date: 14th September 2020.

[11] Open Source, *Eclipse OpenJ9*, available at https://github.com/eclipse-openj9/openj9. Access date: 10th December 2021.

[12] Open Source, *Eclipse OpenJ9*, available at https://www.eclipse.org/community/eclipse_newsletter/2018/april/openj9.php. Access date: 14th September 2020.

[13] Open Source, *DDR*, available at https://github.com/eclipse-openj9/openj9/blob/master/runtime/ddr/ddr.readme. Access date: 19th August 2021.

[14] Jean-Philippe Legault, *Prototype Eclipse OMR Port Performance Evaluation on AArch64*, available at https://unbscholar.lib.unb.ca/islandora/object/unbscholar%3A10462, University of New Brunswick, 2020.

[15] Ricardo Nabinger Sanchez, José Nelson Amaral, Duane Szafron, Marius Pirvu and Mark Stoodley, *Using Machines to Learn Method-Specific Compilation*

Strategies, International Symposium on Code Generation and Optimization (CGO 2011), 2011, pp. 257 - 266.

[16] Levon Stepanian, Angela Demke Brown, Allan Kielstra, Gita Koblents, Kevin Stoodley, *Inlining java native calls at runtime*, VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments, June 2005, pp. 121 - 131.

[17] Eric Coffin, Scott Young, Kenneth B. Kent, Marius Pirvu, *A Roadmap for Extending MicroJIT: A Lightweight Just-in-Time Compiler for Decreasing Startup Time*,Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering, CASCON '19, Toronto, 2019, pp. 293–298.

[18] Aaron G. Graham, *Just-In-Time Compiler Performance Evaluation on the AArch64 Platform*, available at `https://unbscholar.lib.unb.ca/islandora/object/unbscholar%3A10594`, University of New Brunswick, 2021.

[19] Christian Wimmer, Codrut Stancu, Peter Hofer, Vojin Jovanovic, Paul Wögerer, Peter B. Kessler, Oleg Pliss, and Thomas Würthinger, *Initialize Once, Start Fast: Application Initialization at Build Time*, Proc. ACM Program. Lang. 3, OOPSLA, Article 184 (October 2019), 29 pages. https://doi.org/10.1145/3360610

[20] Christian Haubl, Christian Wimmer, Hanspeter Mossenbock, *Context-sensitive trace inlining for Java*, Computer Languages, Systems & Structures, Volume 39, Issue 4, 2013, Pages 123-141, ISSN 1477-8424, https://doi.org/10.1016/j.cl.2013.04.002.

[21] Eclipse OpenJ9, *Using -X command-line options*, available at https://www.eclipse.org/openj9/docs/x_jvm_commands/. Access date: 26th August 2021.

[22] IBM Documentation, *How the JIT compiler optimizes code*, available at https://www.ibm.com/docs/en/sdk-java-technology/8?topic=compiler-how-jit-optimizes-code. Access date: 10th December 2021.

[23] Mark Stoodley, *Under the Hood of the Testarossa JIT Compiler*, available at https://www.slideshare.net/MarkStoodley/under-the-hood-of-the-testarossa-jit-compiler. Access date: 24th February 2022.

[24] Arm Developer, *Neon*, available at https://developer.arm.com/architectures/instruction-sets/simd-isas/neon. Access date: 26th August 2021.

[25] Arm Developer, *Neon*, available at https://developer.arm.com/documentation/den0024/a/AArch64-Floating-point-and-NEON. Access date: 12th December 2021.

[26] Eclipse OpenJ9, *OpenJ9 command-line options*, available at https://www.eclipse.org/openj9/docs/cmdline_specifying/. Access date: 19th August 2021.

[27] Manfred Jendrosch, *Runtime escape analysis in a Java virtual machine*, Masters thesis, University of New Brunswick, May, 2013.

[28] Anthony Ira Wasserman, *Java development environments (panel)*, ICSE '97: Proceedings of the 19th International Conference on Software Engineering, May 1997, pp. 582–583.

[29] Nick Samoylov, *Introduction to Programming : Learn to Program in Java with Data Structures, Algorithms, and Logic*, Packt Publishing,2018. Access date: 26th January 2022.

[30] Bill Venners, *Inside the Java Virtual Machine*, available at `artima.com/insidejvm/ed2/jvm9.html`. Access date: 21st October 2020.

[31] Oracle, *Class System*, available at `https://docs.oracle.com/javase/7/docs/api/java/lang/System.html`. Access date: 22nd October 2020.

[32] Arm®Architecture Reference Manual, *Armv8, for A-profile architecture*, 22 July 2021, Updated Armv8.7 EAC release.

[33] Mathias Longo, Ana Rodriguez, Cristian Mateos, Alejandro Zunino, *Reducing energy usage in resource-intensive Java-based scientific applications via micro-benchmark based code refactorings*, Computer Science and Information Systems 16(2), June, 2019, pp. 41–564.

[34] Hayri Acar, Gülfem Alptekin, Jean-Patrick Gelas, Parisa Ghodous, *The Impact of Source Code in Software on Power Consumption*, International Journal of Electronic Business Management, Electronic Business Management Society, Taiwan, 2016, 14, pp. 42-52.

[35] David Patterson, *Reduced instruction set computer then and now*, IEEE, Computer (Volume: 50, Issue: 12, December 2017), pp. 10 -12.

[36] Oracle, *Java Native Interface*, available at `https://community.ibm.com/community/user/ibmz-and-linuxone/blogs/eduardo-ortega1/2021/03/04/ibm-z-ibm-power-systems-linuxone-and-other-wonders`. Access date: 13th September 2021.

[37] IBM Z and LinuxONE Community, *JRE (IBM Z, IBM Power Systems, LinuxONE and other wonders*, available at `https://www.ibm.com/cloud/learn/jre`. Access date: 14th September 2020.

[38] Eclipse, *Eclipse OMR*, available at `https://www.eclipse.org/omr/starter/whatisomr.html`. Access date: 14th September 2020.

[39] ARM Developer, *Arm CPU Architecture*, available at `https://developer.arm.com/architectures/cpu-architecture`. Access date: 14th September 2020.

[40] S. Fouzi Husaini, *Using The Java Native Interface*, XRDS: Crossroads, The ACM Magazine for Students, November 1997, Volume 4, Issue 2.

[41] Eclipse, *Overview of the JNI*, available at `https://www.iitk.ac.in/esc101/05Aug/tutorial/native1.1/concepts/index.html`. Access date: 1st September 2021.

[42] Eclipse, *Java Class Library*, available at `https://en.wikipedia.org/wiki/Java_Class_Library`. Access date: 1st September 2021.

[43] Oracle, *Java Native Interface*, available at `https://docs.oracle.com/javase/8/docs/technotes/guides/jni/`. Access date: 22nd October 2020.

[44] Oracle, *Class System JDK 11*, available at `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/System.html`. Access date: 23rd October 2020.

[45] Pathania Nikhilős, *Learning Continuous Integration with Jenkins, and SPECjvm2008*, MLA 8th Edition (Modern Language Assoc.), APA 7th Edition (American Psychological Assoc.), Packt Publishing, 2016, Birmingham, ISBN 978-1-78528-483-0.

[46] Scott Young, *Persistent Memory Storage of Cold Regions in the OpenJ9 Java Virtual Machine*, University of New Brunswick, 2018.

[47] Eclipse OpenJ9, *Garbage collection policies*, available at https://www.eclipse.org/openj9/docs/gc/. Access date: 10th December 2021.

[48] IBM SDK, Java Technology Edition, *Garbage collection policies*, available at https://www.ibm.com/docs/en/sdk-java-technology/8?topic=management-gc-policies. Access date: 26th January 2022.

[49] Daryl Maier, Kenneth Kent, *Advances in Open Runtime Technologies for the Cloud*, Proceedings of the 27th Annual International Conference on Computer Science and Software Engineering, CASCON '17, pp. 303 - 306.

[50] Greg Bollella, Kirk Reinholtz, *Scoped Memory*, Proceedings Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, ISIRC 2002, pp. 23-25.

[51] GeeksforGeeks, *Deep, Shallow and Lazy Copy with Java Examples*, available at https://www.geeksforgeeks.org/deep-shallow-lazy-copy-java-examples/. Access date: 3rd January 2022.

[52] Eclipse OpenJ9 (Open Source), *JIT Write Barriers*, available at https://github.com/eclipse-openj9/openj9/blob/master/doc/compiler/JitWriteBarriers.md. Access date: 23rd February 2022.

[53] Eclipse OpenJ9 Blog - Yan Luo, *Concurrent Scavenge Garbage Collection Policy*, available at https://blog.openj9.org/2019/03/25/concurrent-scavenge-garbage-collection-policy/. Access date: 3rd January 2022.

[54] The GDB developers, *GDB: The GNU Project Debugger*, available at https://www.sourceware.org/gdb/. Access date: 3rd January 2022.

[55] Mayur Ramgir, Nick Samoylov, *Java 9 High Performance*, Packt Publishing; 2017. Accessed January 4, 2022.

[56] Philipp Lengauer, Verena Bitto, Hanspeter Mössenböck, Markus Weningerős, *A Comprehensive Java Benchmark Study on Memory and Garbage Collection Behavior of DaCapo, DaCapo Scala, and SPECjvm2008*, ICPE'17, April 22-26, 2017, L'Aquila, Italy, pp. 3–14.

[57] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg , Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, Ben Wiedermannős, *The DaCapo Benchmarks: Java Benchmarking Development and Analysis*, ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications, 2006, pp. 169-190.

[58] GitHub, *Dacapobench*, available at `https://github.com/dacapobench/dacapobench/tree/master/benchmarks/bms`. Access date: 3rd February 2022.

[59] Xi Yang, Stephen M. Blackburn, Daniel Frampton, Jennifer B. Sartor, Kathryn S. McKinley, *Why nothing matters: the impact of zeroing*,ACM SIGPLAN Notices, Volume 46, Issue 10, October 2011 pp 307–324.

[60] ARM Developer, *ARMv8 Registers*, available at `https://developer.arm.com/documentation/den0024/a/armv8-registers`. Access date: 15th September 2020.

[61] SPECjvm® 2008, *SPECjvm® 2008*, available at `https://www.spec.org/jvm2008/`. Access date: 5th October 2021.

[62] ARM Developer, *An Introduction to the ARMv8 Instruction Sets*, available at https://developer.arm.com/documentation/den0024/a/an-introduction-to-the-armv8-instruction-sets. Access date: 15th September 2020.

[63] Intel, *Intel® 64 and IA-32 Architectures Software Developer's Manual*, June 2021.

[64] Erick Ochoa, Cijie Xia, Karim Ali, Andrew Craik, and José Nelson Amaral. *U Can't Inline This!*, In Proceedings of (CASCON'21), November 22 - 26 2021, Toronto, Canada.

[65] Chad Paradis and Vincent M. Weaver, *Enabling Raspberry Pi Performance Counter Support on Linux perf_event*, UMaine ECE Tech Report 2014-2.

[66] Perf, *perf: Linux profiling with performance counters*, available at https://perf.wiki.kernel.org/index.php/Main_Page, Access date: 9th December 2021.

[67] LWN, *perf: add support for profiling jitted code*, available at https://lwn.net/Articles/633846/, Access date: 9th December 2021.

[68] Eclipse OpenJ9, *Java Virtual Machine Tool Interface*, available at https://www.eclipse.org/openj9/docs/interface_jvmti/, Access date: 9th December 2021.

[69] Intel, *White Paper: Introduction to Intel® Architecture*, available at https://developer.arm.com/documentation/den0024/a/an-introduction-to-the-armv8-instruction-sets/the-armv8-instruction-sets/distinguishing-between-32-bit-and-64-bit-a64-instructions. Access date: 15th September 2020.

[70] Zhuoran Li, *Stall-Focused Benchmarks for JVMs on the x86 Architecture*, Masters thesis, University of New Brunswick, October, 2020.

[71] IBM Servers and Storage, *Top IBM Power Systems myths: x86 is the industry standard and Power is becoming obsolete*, available at https://www.ibm.com/blogs/systems/top-ibm-power-systems-myths-x86-is-the-industry-standard/. Access date: 13th September 2021.

[72] ARM Developer, *Distinguishing between 32-bit and 64-bit A64 instructions*, available at https://developer.arm.com/documentation/den0024/a/an-introduction-to-the-armv8-instruction-sets/the-armv8-instruction-sets/distinguishing-between-32-bit-and-64-bit-a64-instructions. Access date: 15th September 2020.

[73] Eclipse, *The JIT compiler*, available at https://www.eclipse.org/openj9/docs/jit/. Access date: 22nd September 2020.

[74] Eclipse, *OpenJ9-OMR*, available at https://github.com/eclipse-openj9/openj9-omr/blob/openj9/compiler/ras/DebugCounter.md. Access date: 15th September 2021.

[75] Martin Cracauer, *Generational Garbage Collection, Write Barriers/Write Protection and userfaultfd(2)*, available at https://medium.com/@MartinCracauer/generational-garbage-collection-write-barriers-write-protection-and-userfaultfd-2-8b0e796b8f7fhttps://medium.com/@MartinCracauer/generational-garbage-collection-write-barriers-write-protection-and-userfaultfd-2-8b0e796b8f7f. Access date: 23rd September 2020.

[76] Richard S. Piepho, William S. Wuős, *A comparison of RISC architectures*, IEEE Micro, Volume: 9, Issue: 4, 1989, pp. 51-62.

[77] Shelley Lambert, *BumbleBench and HumbleBench*, available at `https://github.com/adoptium/bumblebench/wiki/BumbleBench-and-HumbleBench#quick-start`. Access date: 21st October 2021.

[78] OpenJDK Wiki, *FAQ*, available at `https://wiki.openjdk.java.net/display/CodeTools/JCov+FAQ`. Access date: 12th December 2020.

[79] OpenJDK Wiki, *FAQ*, available at `https://wiki.openjdk.java.net/display/HotSpot/PrintAssembly`. Access date: 12th December 2020.

[80] ARM Developer, *Using floating point registers*, available at `https://developer.arm.com/documentation/102374/0101/Loads-and-stores---using-floating-point-registers`. Access date: 13th December 2020.

[81] Grimmer, Matthias and Rigger, Manuel and Stadler, Lukas and Schatz, Roland and Mössenböck, Hanspeter *An Efficient Native Function Interface for Java*, Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, PPPJ '13, Stuttgart, Germany, pp. 35–44.

[82] Grichi, Manel and Abidi, Mouna and Guéhéneuc, Yann-Gaël and Khomh, Foutse *State of Practices of Java Native Interface*, Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering, CASCON '19, November 22 - 26 2021, Toronto, Canada, pp. 274–283.

[83] Monica Beckwith, *Eclipse Open J9 – an Open Source Java Virtual Machine Based on the Eclipse OMR Project*, available at `https://www.infoq.com/news/2018/03/OMR-OpenJ9/`. Access date: 17th December 2020.

[84] IBM SDK, Java Technology Edition, *How the JIT compiler optimizes code*, available at

https://www.ibm.com/docs/en/sdk-java-technology/8?topic=compiler-how-jit-optimizes-code. Access date: 19th December 2020.

[85] Owen Kaser, C.R. Ramakrishnan, *Evaluating inlining techniques*, Computer Languages,Volume 24, Issue 2, 1998, pp. 55-72.

[86] McFarling Scott, *Procedure Merging with Instruction Caches*, Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation, PLDI '91, pp. 71–79.

[87] Eclipse OpenJ9, *Introduction to class data sharing*, available at https://www.eclipse.org/openj9/docs/shrc/. Access date: 22nd December 2020.

[88] IBM SDK, Java Technology Edition, *-Xshareclasses*, available at https://www.ibm.com/docs/en/sdk-java-technology/8?topic=options-xshareclasses. Access date: 22nd December 2020.

[89] Filip Jeremic, *Inspecting OpenJ9 performance with perf on Linux – JIT Compiled Methods*, available at https://blog.openj9.org/2019/07/18/inspecting-openj9-performance-with-perf-on-linux-jit-compiled-methods/. Access date: 22nd December 2020.

[90] Sun Microsystems, *IA-32 Assembly Language Reference Manual*, available at https://docs.oracle.com/cd/E19455-01/806-3773/806-3773.pdf. Access date: 24th December 2020.

[91] Md Alvee Noor, *Adding SIMD Support to Improve Performance of Eclipse OpenJ9 on the AArch64 Platform*, University of New Brunswick, 2021.

[92] GCC GNU, *Optimization Levels*, available at https://gcc.gnu.org/onlinedocs/gnat_ugn/Optimization-Levels.html. Access date: 1st January 2021.

[93] GCC GNU, *Auto-vectorization in GCC*, available at https://gcc.gnu.org/projects/tree-ssa/vectorization.html. Access date: 1st January 2021.

[94] Geert M. N. Verschuuren, *Excel 2013 for Scientists* Vol Rev. & expanded 3rd ed. Holy Macro! Books; 2014. Accessed January 2, 2022.

[95] Eben Upton, Jeffrey Duntemann, Ralph Roberts, Tim Mamtora, and Ben Everard, *Learning Computer Architecture with Raspberry Pi* https://ebookcentral.proquest.com/lib/unb/detail.action?docID=4653541, John Wiley & Sons, Incorporated, 2016. ProQuest Ebook Central.

Appendix A

Code Contributions for arraycopy

- Siri Sahithi Ponangi. *AArch64: Adding ArrayCopyBNDCHKEvaluator()*. Merged on: 2021-06-06 `https://github.com/eclipse/omr/pull/6041`.
- Siri Sahithi Ponangi. *AArch64: Implementing ArrayCHKEvaluator()*. Merged on: 2021-06-22 `https://github.com/eclipse/openj9/pull/12970`.
- Siri Sahithi Ponangi. *AArch64: Assembly helper for arraycopy*. Merged on: 2021-06-22 `https://github.com/eclipse/omr/pull/6073`.
- Siri Sahithi Ponangi. *AArch64: Adding backward arraycopy implementation*. Merged on: 2021-09-27 `https://github.com/eclipse/omr/pull/6174`.
- Siri Sahithi Ponangi. *AArch64: Adding additional arraycopy helper entries*. Merged on: 2021-10-05 `https://github.com/eclipse/omr/pull/6188`.
- Siri Sahithi Ponangi. *AArch64: Adding additional arraycopy helper entries*. Merged on: 2021-10-05 `https://github.com/eclipse/openj9/pull/13598`.
- Siri Sahithi Ponangi. *AArch64: Disabling Subrange Overlap test*. Merged on: 2022-01-11 `https://github.com/eclipse/openj9/pull/14247`.

- Konno Kazuhiro. *AArch64: Implement reference arraycopy.* Merged on: 2021-08-08 https://github.com/eclipse/omr/pull/6129.
- Konno Kazuhiro. *AArch64: Implement arraycopyEvaluator() for primitive arraycopy.* Merged on: 2021-08-04 https://github.com/eclipse/omr/pull/6124.
- Konno Kazuhiro. *AArch64: Add arraycopy helpers.* Merged on: 2021-07-22 https://github.com/eclipse/omr/pull/6076.
- Konno Kazuhiro. *(0.29.0) AArch64: Add x18 to register dependencies for calling arraycopy helper.* Merged on: 2021-10-13 https://github.com/eclipse-o\\penj9/openj9/pull/13676.
- Konno Kazuhiro. *AArch64: Add x18 to register dependencies for calling arraycopy helper.* Merged on: 2021-10-12 https://github.com/eclipse-openj9/openj9\\/pull/13675.
- Konno Kazuhiro. *AArch64: Implement reference arraycopy.* Merged on: 2021-08-17 https://github.com/eclipse-openj9/openj9/pull/13279.
- Konno Kazuhiro. *AArch64: Enable ArrayCopyOpts and primitive arraycopy.* Merged on: 2021-08-05 https://github.com/eclipse-openj9/openj9/pull/13263.
- Konno Kazuhiro. *AArch64: Add arraycopy helpers.* Merged on: 2021-06-25 https://github.com/eclipse-openj9/openj9/pull/12987.
- Akira Saito. *AArch64: Stop using placeholder register for arraycopy.* Merged on: 2021-11-04 https://github.com/eclipse/omr/pull/6226.
- Akira Saito. *AArch64: Kill placeholder registers for arraycopy.* Merged on: 2021-11-09 https://github.com/eclipse-openj9/openj9/pull/13873.

Appendix B

Benchmarks

In this section, supplementary results are presented, that were not discussed in the Results Chapter. The commit details used throughout the testing, comparing results from DaCapo and SPECjvm2008 Benchmark suites on different architectures are mentioned in this section. The following version is used through out testing.

```
openjdk 11.0.13-internal 2021-10-19
OpenJDK Runtime Environment (build 11.0.13-internal+0-adhoc.spona
ngi.openj9-openjdk-jdk11)
Eclipse OpenJ9 VM (build master-b466f9fa0, JRE 11 Linux aarch64-6
4-Bit Compressed References 20211110\_000000 (JIT enabled, AOT en
abled)
OpenJ9   - b466f9fa0
OMR      - d6489755f
JCL      - 4148fad367 based on jdk-11.0.13+8)
```

The gcc/g++ versions used are:

```
(Ubuntu/Linaro 7.5.0-6ubuntu2) 7.5.0
```

The SPECjvm2008[1] benchmark suite centers on JRE performance by running one

[1]https://www.spec.org/jvm2008/

Table B.1: Specification of x86.

Architecture	x86_64
Model	Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz
CPUs	12

application at a time[56]. The SPECjvm2008 benchmark has higher average array length, average object size and array rate making it suitable to work with large arrays. In this section, we will see a comparision of improvements in performance due to arraycopy implementation on various architecture like x86, ppcle, s390x with AArch64. Although we know the average array lengths and average array sizes, we do not have much information on the copying such as inlinable forward, inlinable backward in DaCapo and SPECjvm2008 benchmark suites. A Microbenchmarking framework such as BumbleBench helps us in understanding the performance of each dataype and datasize in different ways of copying. Hence, benchmark suites such as DaCapo and SPECjvm2008 do not give much insight into arraycopy performance in particular. DaCapo is centered on "real-world" applications and SPECjvm is a synthetic benchmark that is not generally considered as a real-world protoype [18]. Hence, we have only included Dacapo in the Results chapter. The SPECjvm2008 benchmark results are graphed to estimate the performance differences across various architectures. The systems that are used are not deemed to be comparable, as they have differences such as dissimilar frequencies, CPUs, hyperthreading and memory. Changing these settings on the individual machines is out of the scope of this research.

The specificications of the machines on which arraycopyEvaluator's performance is benchmarked is as in the following tables. Tables B.1, B.2, B.3 show the specifications of x86, s390 and ppc machines respectively.

Table B.2: Specification of s390x.

Architecture	s390x
Model	IBM/S390
CPUs	2

Table B.3: Specification of ppc.

Architecture	ppc64le
Model	POWER8E (raw), altivec supported
CPUs	24

B.1 A comparison of benchmark results

The following benchmark results were obtained before disabling the Subrange Overlap test for AArch64. Hence the optimized arraycopy results do not include improved performance on backward arraycopy. The AArch64 machine used was RPi.

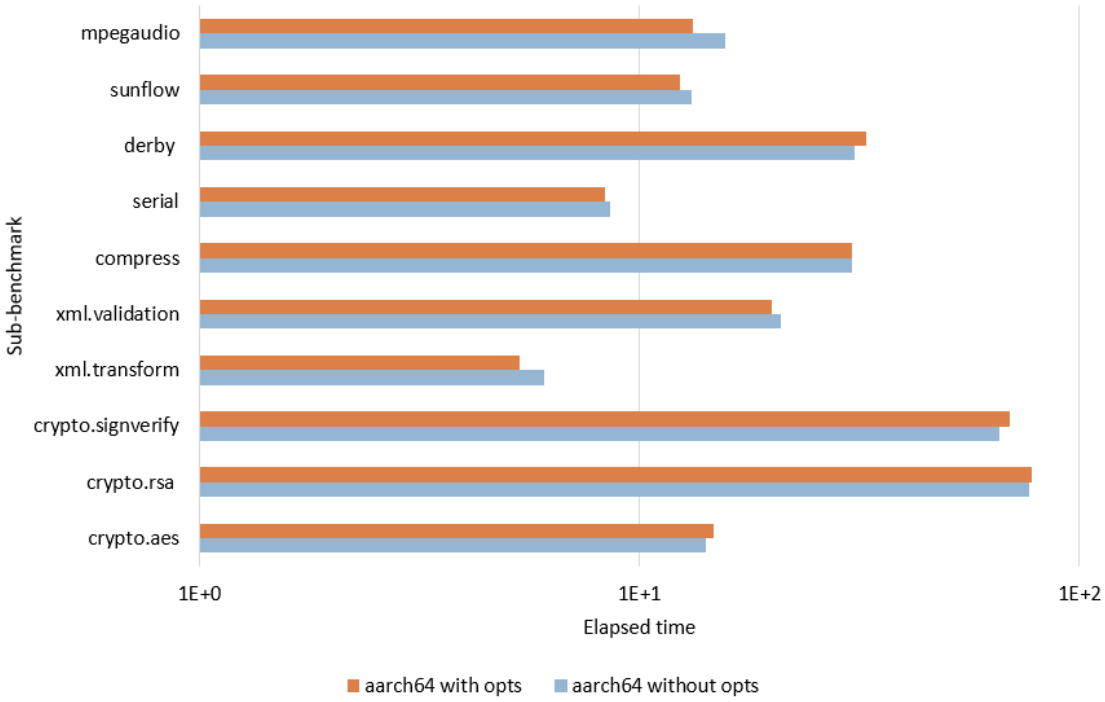

Figure B.1: SPECjvm2008 on AArch64

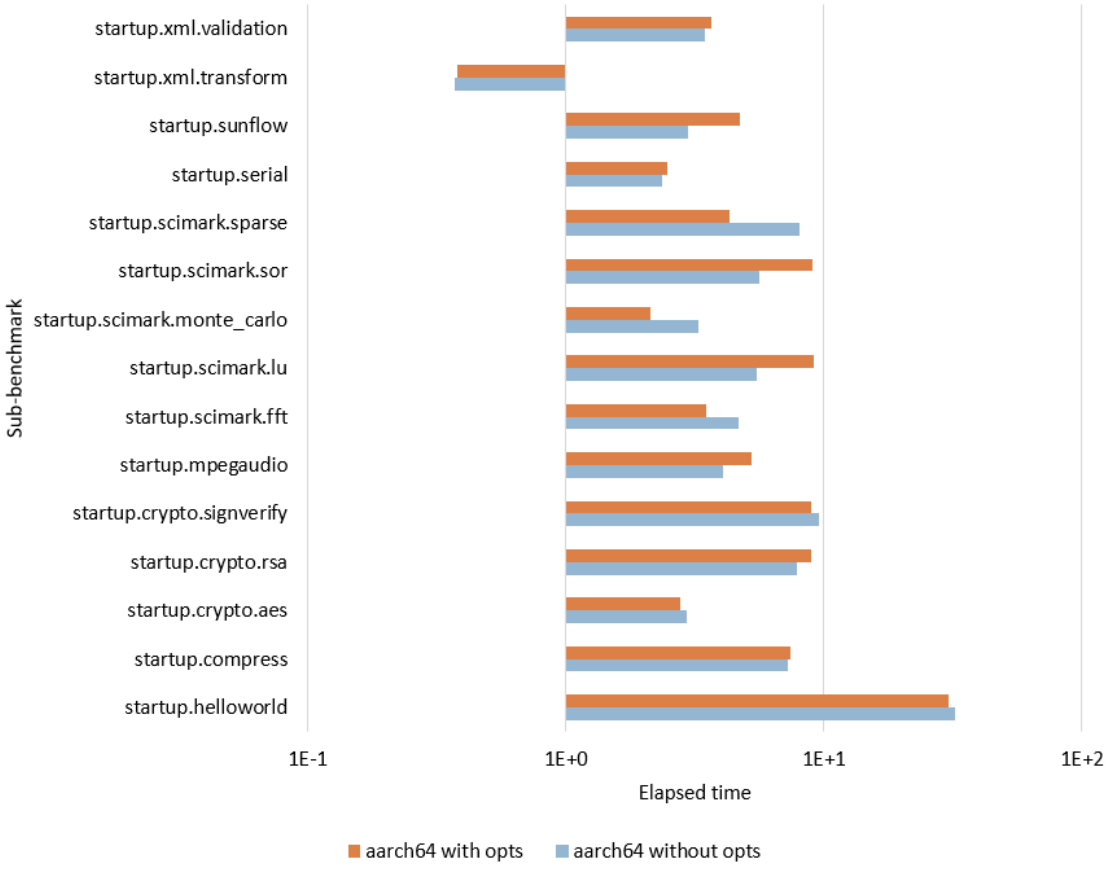

Figure B.2: SPECjvm2008 Startup on AArch64

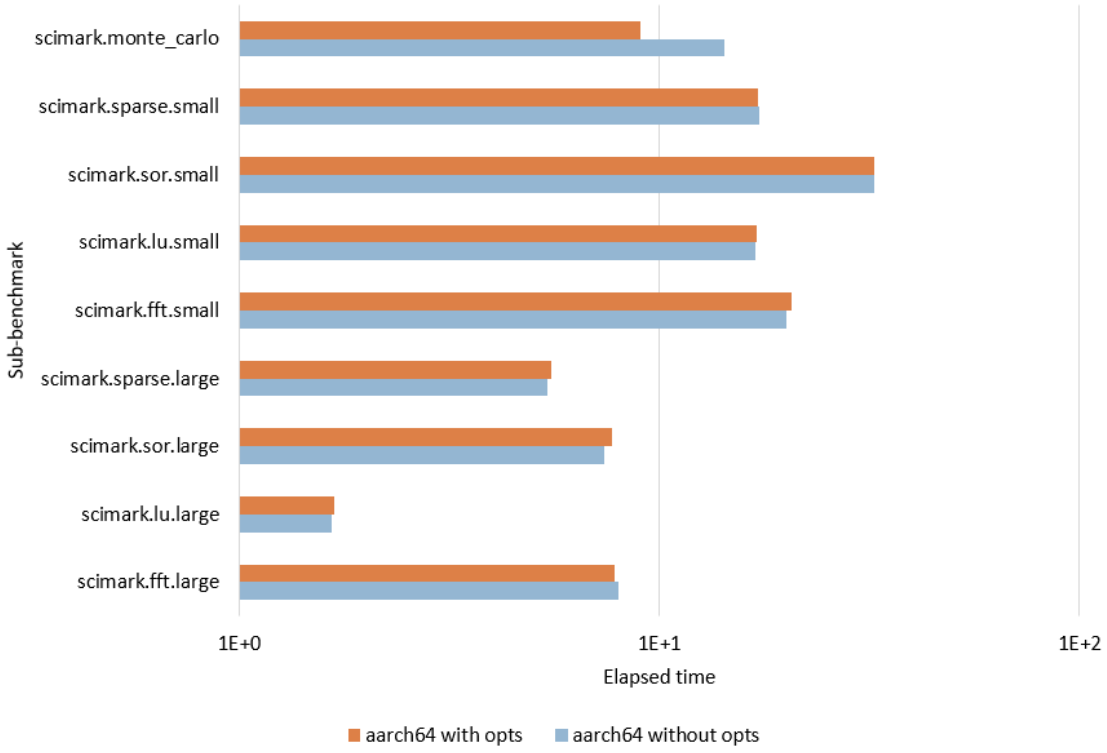

Figure B.3: SPECjvm2008 Scimark on AArch64

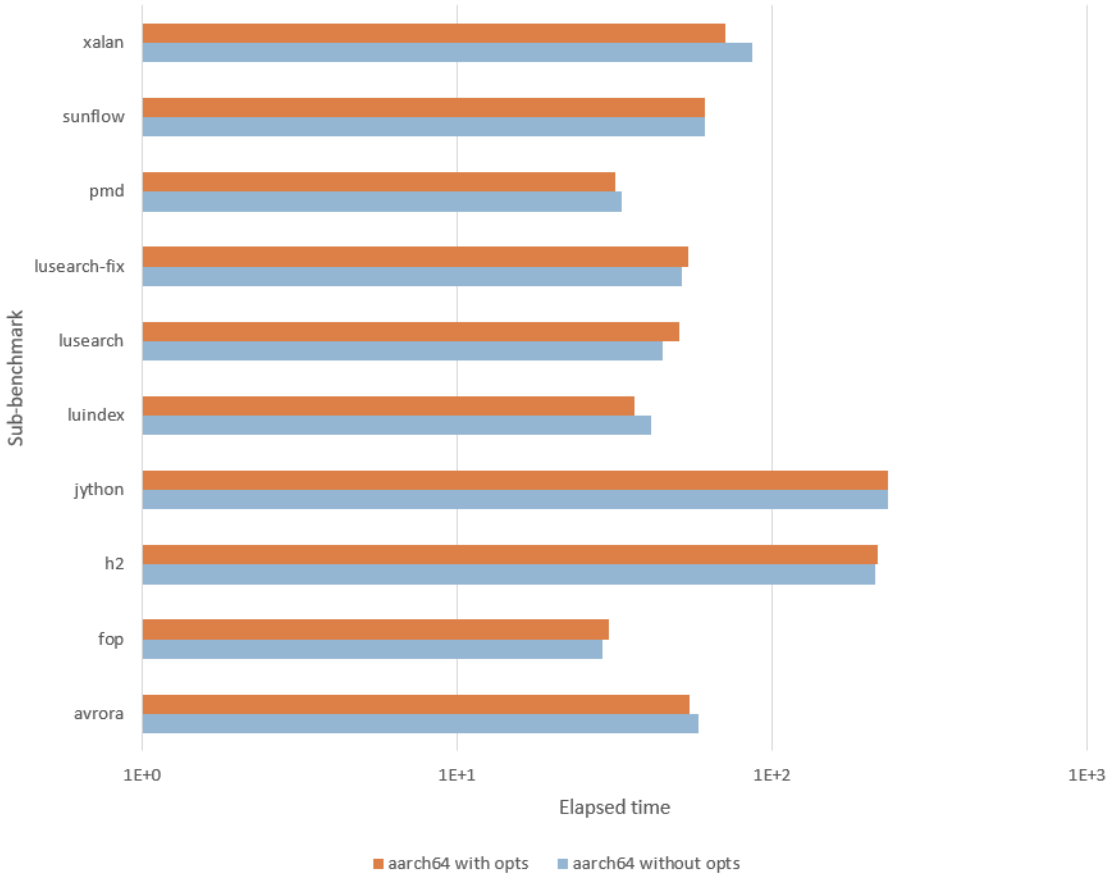

Figure B.4: DaCapo on AArch64 (Logarithmic Values)

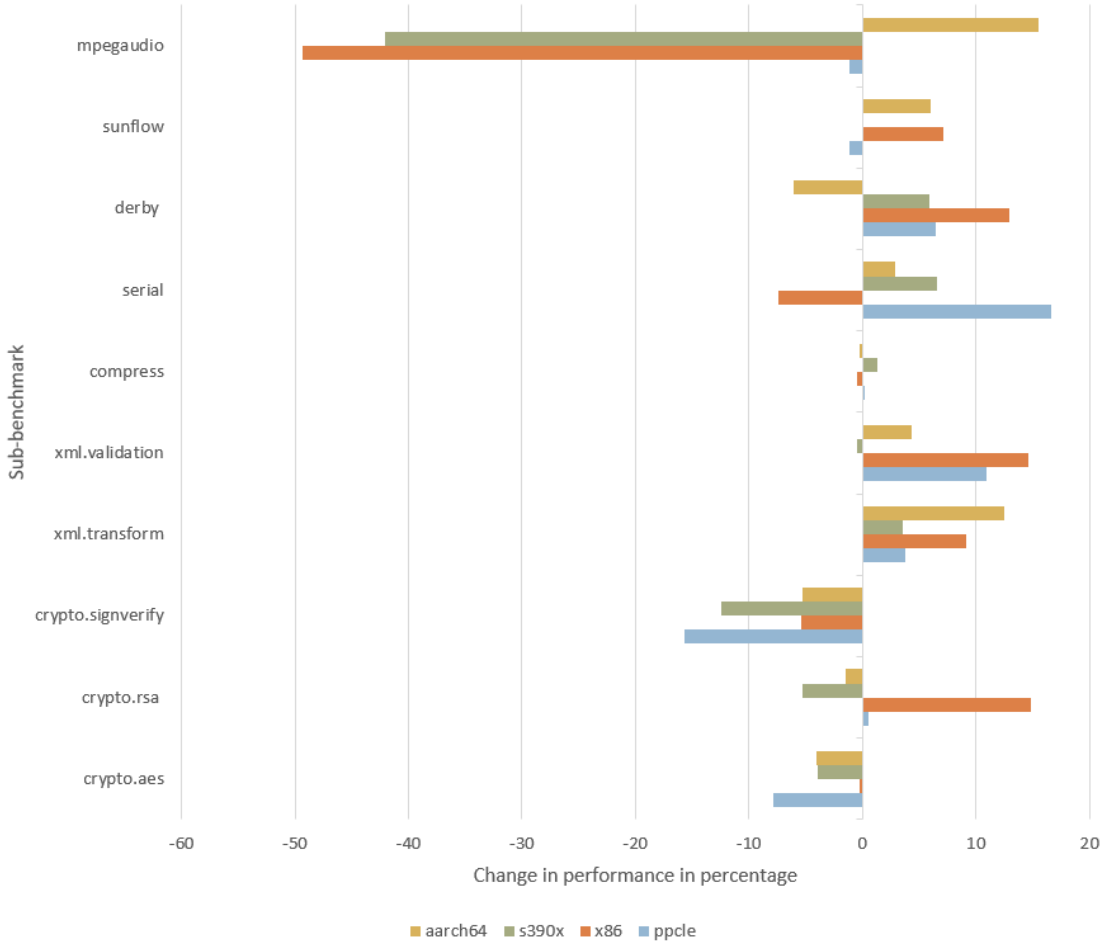

Figure B.5: SPECjvm2008 (Performance changes in percentage)

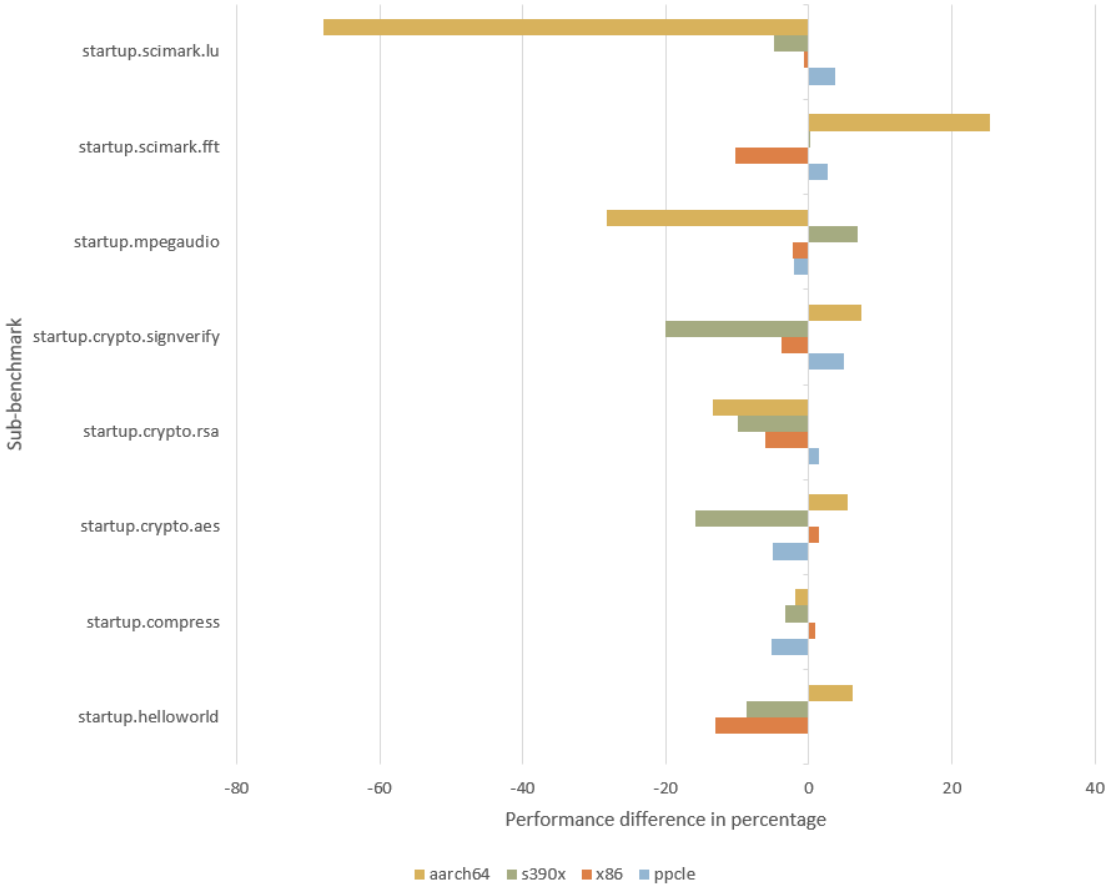

Figure B.6: SPECjvm2008 Startup (Part 1 - Performance changes in percentage)

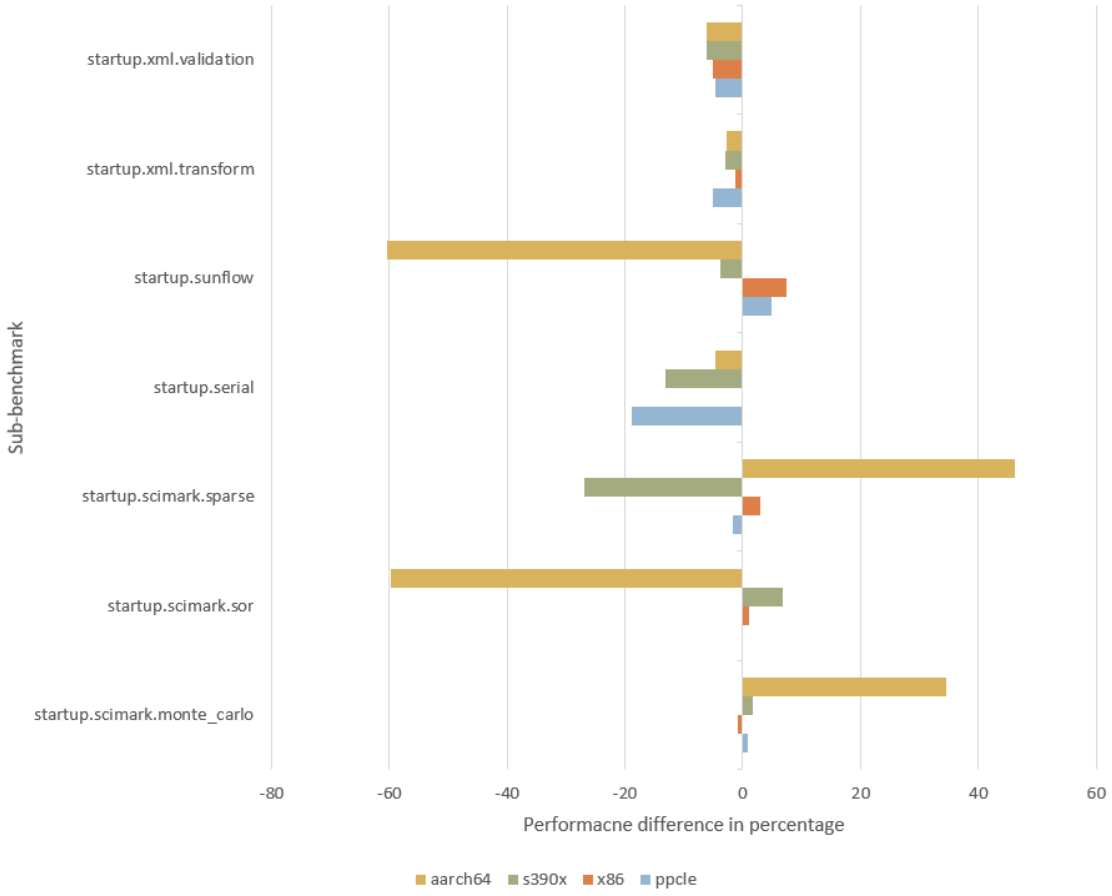

Figure B.7: SPECjvm2008 Startup (Part 2 - Performance changes in percentage)

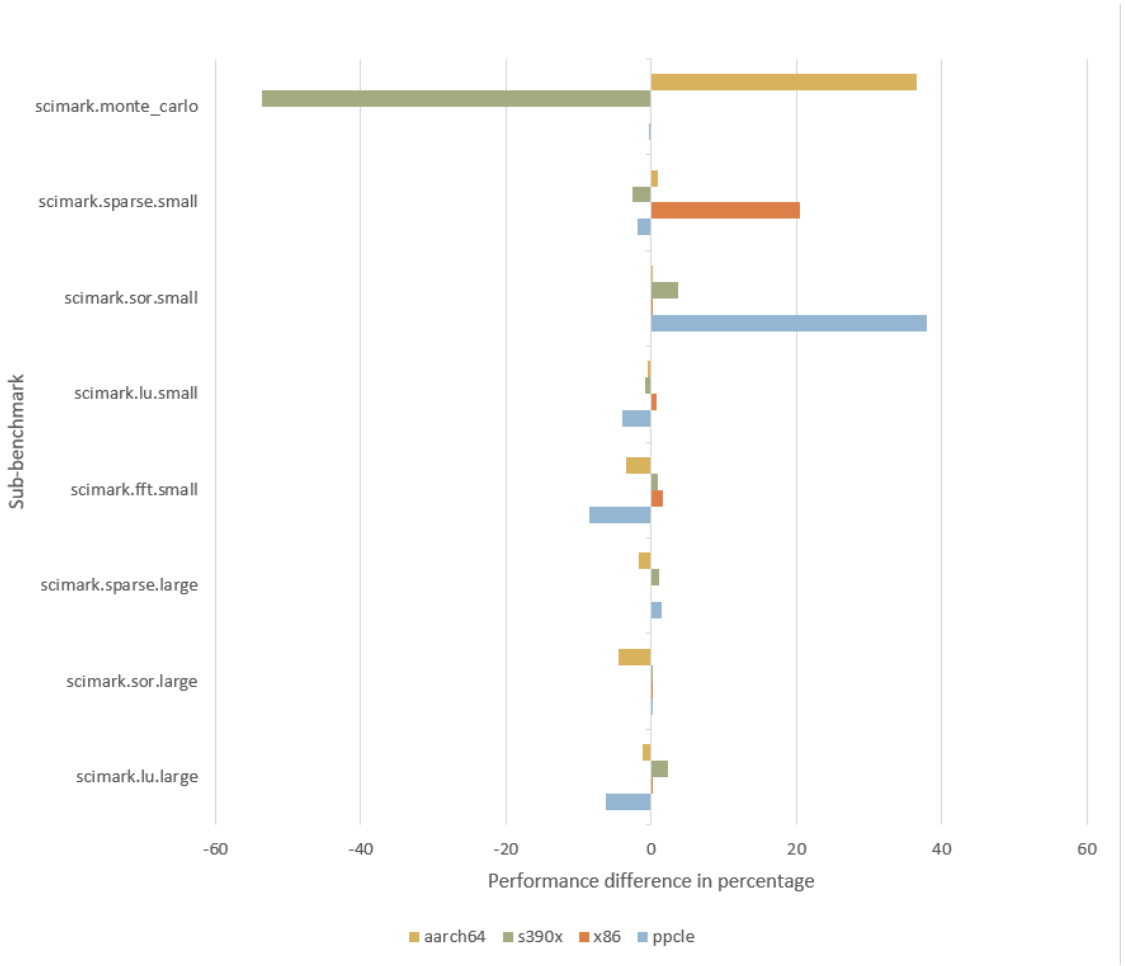

Figure B.8: SPECjvm2008 Scimark (Performance changes in percentage)

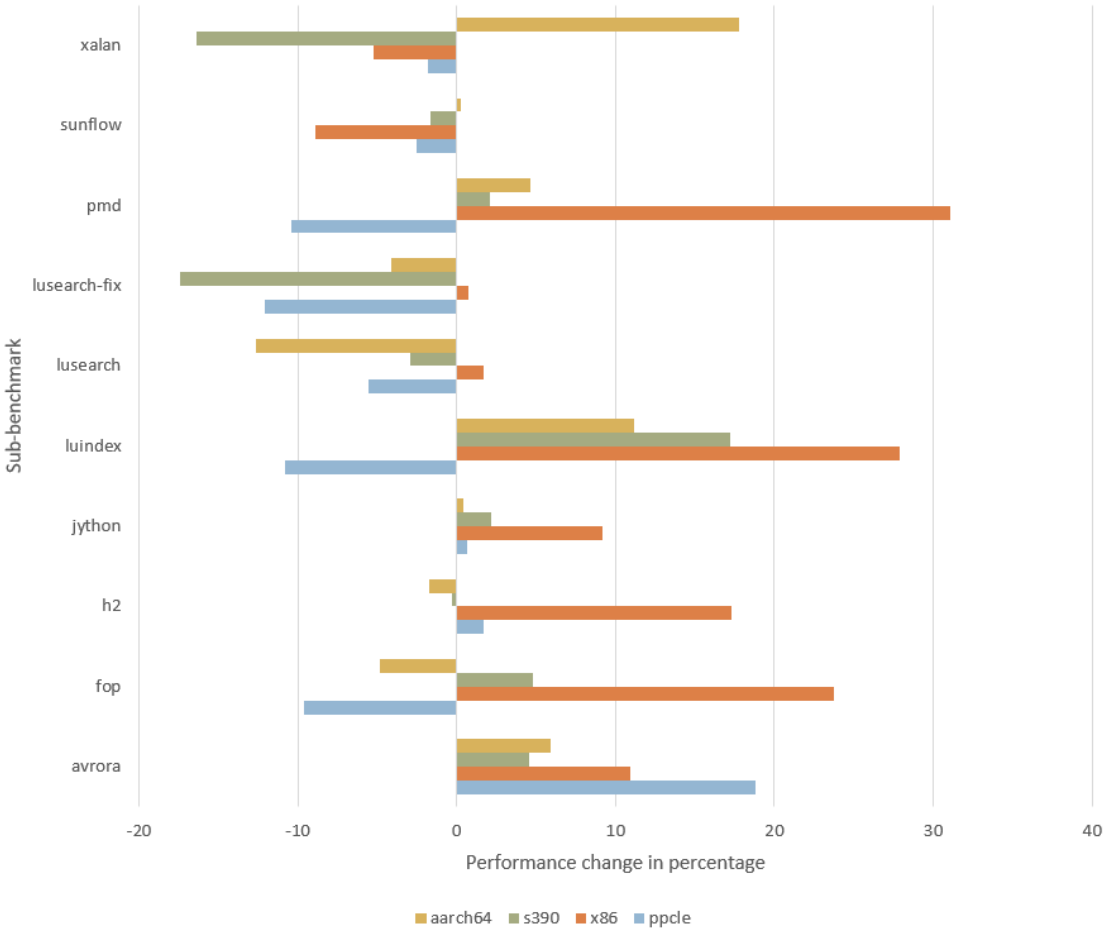

Figure B.9: DaCapo (Performance changes in percentage)

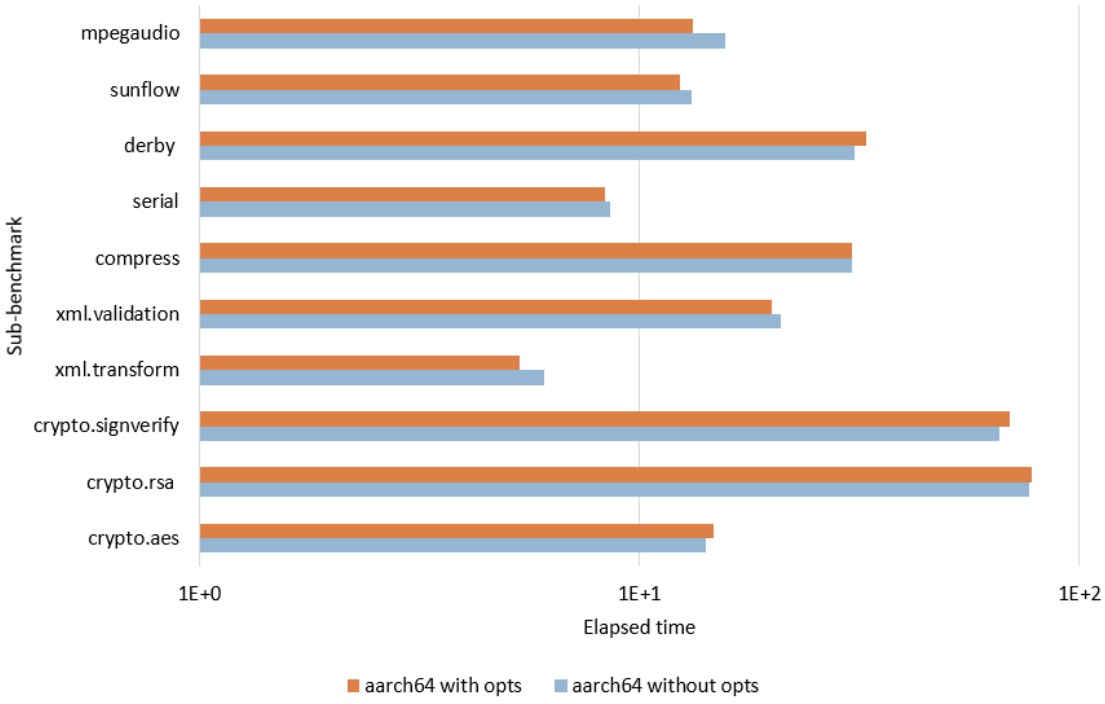

Figure B.10: SPECjvm2008

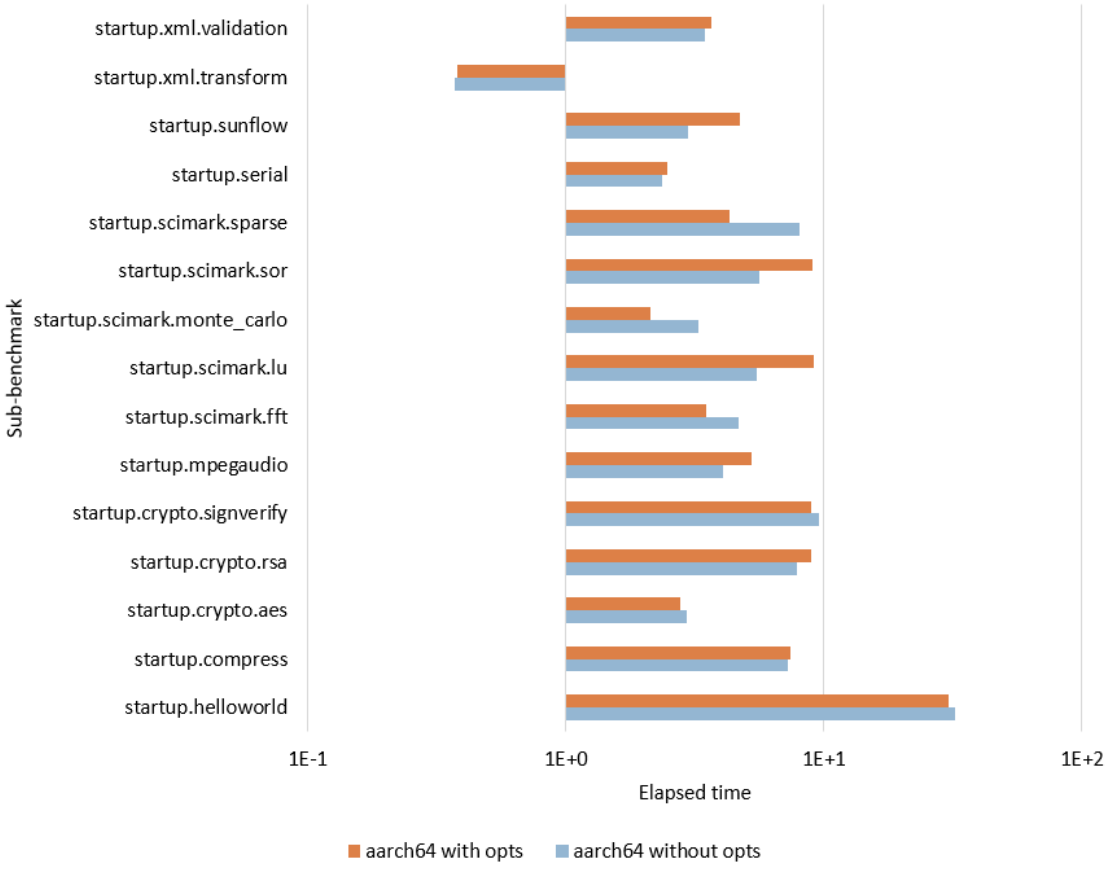

Figure B.11: SPECjvm2008 Startup on AArch64

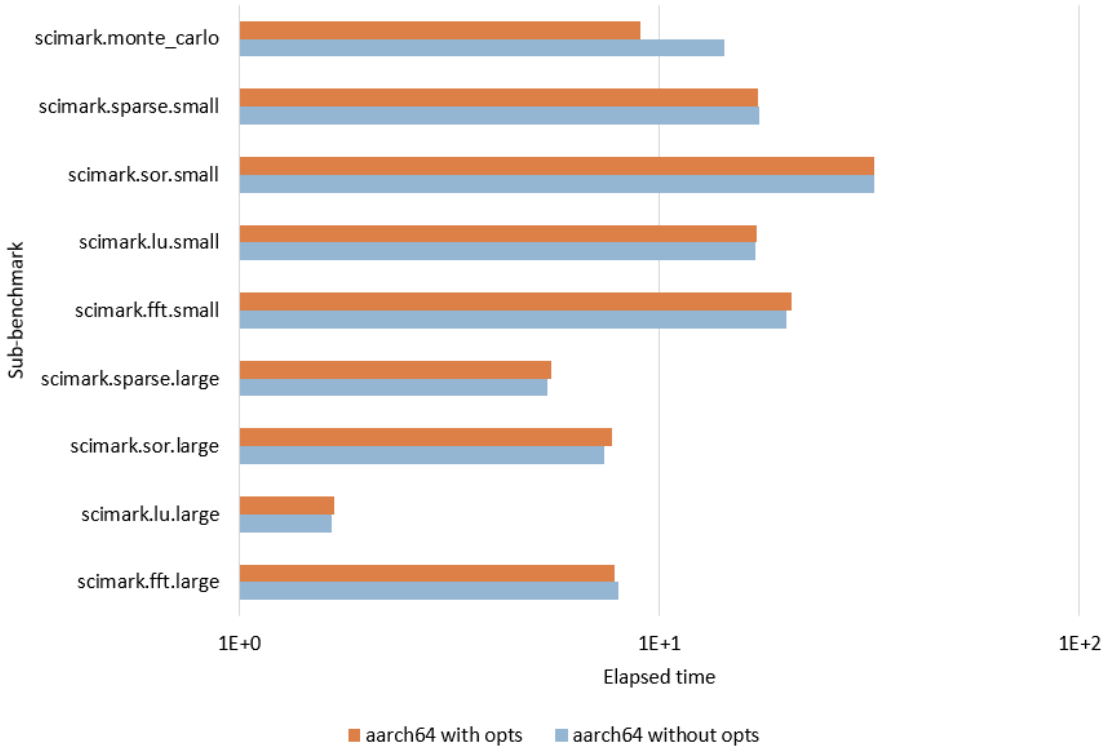

Figure B.12: SPECjvm2008 Scimark on AArch64

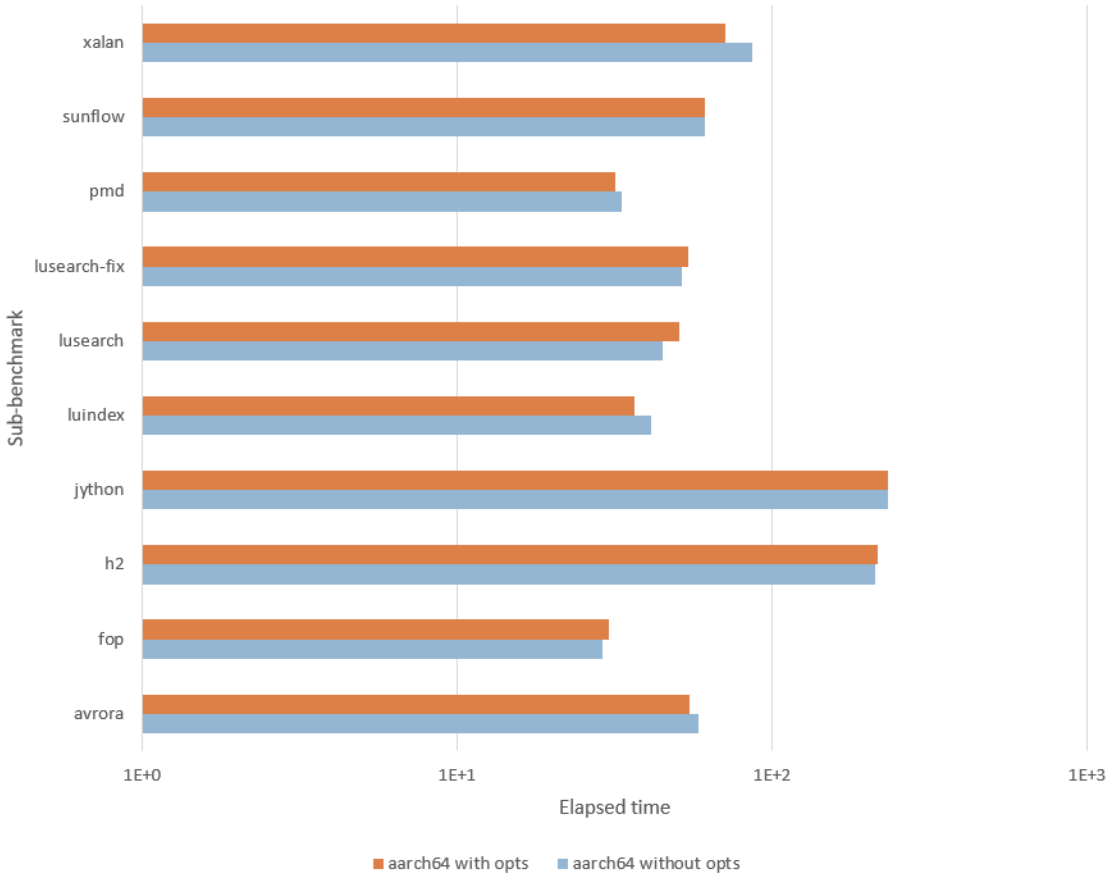

Figure B.13: DaCapo on AArch64 (Logarithmic Values)

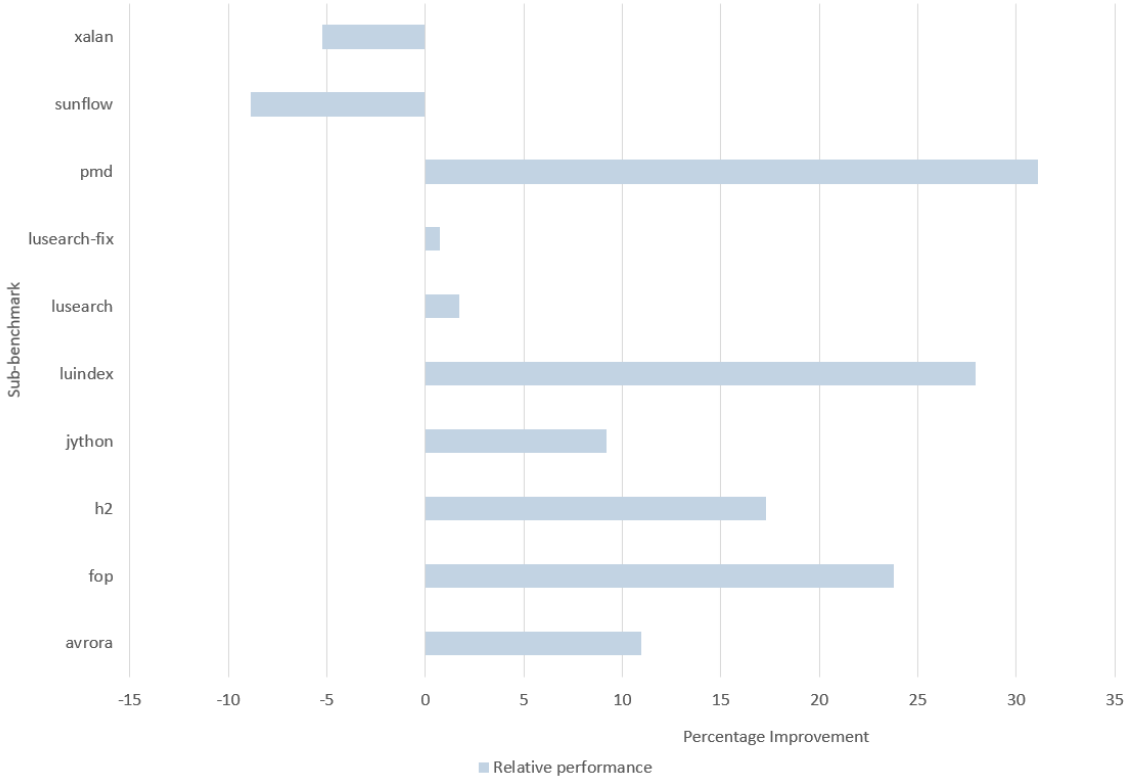

Figure B.14: Relative performance on x86_64 - DaCapo

Appendix C

Instructions for Setup and Testing

C.1 Building Eclipse OMR - OpenJ9 pair

This section includes the instructions for building OpenJ9, running OpenJ9 tests and installing ant for BumbleBench benchmark suite. This section also contains steps to use the Perf tool to see the assembly instructions of JITed code. In addition, this section comprises test programs that generate arraycopy nodes.

C.1.1 Initial set up

Follow the steps below from your working directory.

- git clone `https://github.com/ibmruntimes/openj9-openjdk-jdk11.git`
- cd openj9-openjdk-jdk11/
- git checkout origin/openj9 -b openj9
- cd ../
- wget 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 -O bootjdk11.tar.gz

- tar -xzf bootjdk11.tar.gz
- mv jdk-11 bootjdk11/
- cd openj9-openjdk-jdk11/
- git clone `https://github.com/eclipse/omr.git`
- git clone `https://github.com/eclipse-openj9/openj9.git`

C.1.2 Making a build

- bash configure –with-cmake –with-boot-jdk=Absolute Path to OpenJ9 Workspace Folder /bootjdk11 –disable-warnings-as-errors –disable-warnings-as-errors-openj9 –disable-warnings-as-errors-omr
- `makeall`

C.1.3 Running OpenJ9 tests

To run the OpenJ9 tests:

- Tools required for running for running the tests: https://github.com/eclipse-openj9/openj9/blob/master/test/docs/Prerequisites.md
- Follow the instructions in the link: `https://github.com/eclipse-openj9/openj9/tree/master/test/README.md`.

The difference between the tests lies in their mode variation.

Listing C.1: Tests and their modes

```
TestArrayCopy_openj9_none_SCC_0 Mode100
TestArrayCopy_openj9_none_SCC_1 Mode101
TestArrayCopy_openj9_none_SCC_2 Mode600
TestArrayCopy_openj9_none_SCC_3 Mode619
```

Command line options for different modes can be found here: `https://github.com/adoptium/TKG/blob/master/resources/modes.xml`

C.1.4 Install ant for BumbleBench

To install ant:

- Follow the instructions in the link : `https://ant.apache.org/manual/index.html`
- cd apache-ant-1.10.12
- wget https://sourceforge.net/projects/ant-contrib/files/ant-contrib/1.0b3/ant-contrib-1.0b3-bin.tar.gz
- tar xf ant-contrib-1.0b3-bin.tar.gz
- mv ant-contrib /lib/
- Set JAVA_HOME to the jdk you want to build the jar file with.
- Set ANT_HOME to the path where ant /bin is in.
- Add the above variables to PATH, example: export PATH=$JAVA\_HOME/bin$:PATH

C.1.5 Running Perf to see instructions of the JITed code

Using Perf with a JVMTI agent helps us to see the assembly notation. JVMTI agent needs to be installed separately, as shown below.

- apt-get -y install build-essential make gcc g++ flex bison libelf-dev libdw-dev libslang2-dev libssl-dev libiberty-dev default-jdk

- git clone –depth 1 https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
- cd linux/tools/perf
- make

These instructions build the JVMTI agent using the Java on $PATH resulting in libperf-jvmti.so object file being created where make is run. The next steps include running perf tool with the JVMTI agent, which will help us see binary information of JIT methods.

- ./linux/tools/perf/perf record -k 1 -e cycles path to java /java -agentpath:path to libperf-jvmti.so -jar Benchmark .jar
- ./linux/tools/perf/perf inject -i perf.data –jit -o perf.data.jitted
- ./linux/tools/perf/perf report -i perf.data.jitted

From any listed method in the perf generated report, clicking on 'a' key will display all the instructions used in that method.

C.1.6 Test Programs

Listing C.2: The primitive arraycopy code (in forward direction)

```
class test2 {
    static void forwardCopy_int(int[] a, int[] b)
    {
        for (int i=0; i<=b.length-1; i++)
            a[i] = b[i];
    }
    static void forwardCopy_byte(byte[] a, byte[] b)
            {
```

```
    for (int i=0; i<=b.length-1; i++)
        a[i] = b[i];
}
public static void main(String[] args) {
int[] src = new int[16];
    for (int i=0; i<src.length; i++)
        src[i] = i+100;

int[] dst = new int[src.length];
    for (int i=0; i<=15;  i++) //  limit=main 64 bytes
        dst[i] = src[i];

int[] x = new int[40];
for (int i=0; i<x.length; i++)
    x[i] = i+100;
int[] y = new int[x.length];

// forwardArrayCopy wordElementArrayCopy limit=main 160 bytes
forwardCopy_int(y, x);
int[] c = new int[25];
for (int i=0; i<c.length; i++)
    c[i] = i+100;
int[] d = new int[c.length];

// forwardArrayCopy wordElementArrayCopy
// limit=main 100 bytes
forwardCopy_int(d, c);

// forwardArrayCopy wordElementArrayCopy
// limit=main 100 bytes
for (int i=0; i<=24; i++)
    d[i] = c[i];
```

```
// forwardArrayCopy wordElementArrayCopy
// limit=main 160 bytes
for (int i=0; i<=39; i++)
    y[i] =x[i];
byte[] m =new byte[43];
for (int i=0; i <m.length; i++)
    m[i] = (byte)(i+1);
byte[] n = new byte[43];

// forwardArrayCopy limit=main 43 bytes
forwardCopy_byte(n, m);

 // forwardArrayCopy limit=main 43 bytes
for (int i=0; i<=42; i++)
    n[i] = m[i];
int[] l = {0, 1};
int[] k = new int[2];
forwardCopy_int(k, l);           // 8 bytes
for (int i=0; i<=1; i++)
    k[i] = l[i];           // 8 bytes

// __arrayCopy call limit=main 56 bytes
System.arraycopy(src, 1, src, 2, src.length - 2);

// forwardArrayCopy wordElementArrayCopy
// limit=main  64 bytes
for (int i=0; i<=15; i++)
    dst[i] = src[i];
// forwardArrayCopy wordElementArrayCopy
// limit=main 64 bytes
for (int i=0; i<= src.length-1; i++)
    dst[i] = src[i];
// forwardArrayCopy wordElementArrayCopy
```

```
    // limit=main 64 bytes
    for (int i=0; i<= 15; i++)
        dst[i] = src[i];
    // forwardArrayCopy wordElementArrayCopy
    // limit=main 64 bytes
    forwardCopy_int(dst, src);
            byte[] a = {0, 1, 2, 3, 4, 5};
    byte[] b = {5, 2, 3, 6, 7, 8};
    // forwardArrayCopy limit=main 3 bytes
    System.arraycopy(a, 1, b, 1, 3);

    }
}
```

Listing C.3: The reference arraycopy code (in forward direction)

```
class test3 {
    static void forwardCopy_int(int[] a, int[] b)
    {
        for (int i=0; i<=b.length-1; i++)
            a[i] = b[i];
    }
    static void forwardCopy_example(example[] a, example[] b)
            {
        for (int i=0; i<10; i++)
            a[i] = b[i];
    }
    public static void main(String[] args) {
    example[] src = new example[100000];
        for (int i=0; i<src.length; i++)
            src[i] = new example(i, "name"+Integer.toString(i));

    example[] dst = new example[src.length];
        for (int i=0; i<100000;  i++)
```

```
            dst[i] = src[i];
    forwardCopy_example(src, dst);

    example[] src2 = new example[8];
    for (int i=0; i<src2.length; i++)
        src2[i] = new example(i, "name"+Integer.toString(i));

    example[] dst2 = new example[src2.length];
        for (int i=0; i<=7;  i++)
            dst2[i] = src2[i];

    for(int x=0; x<100; x++)
    System.arraycopy(src, 1, dst, 1, 999);

    }
}
class example {
    public int id;
    public String name;

    example(int id, String name)
    {
        this.id = id;
        this.name = name;
    }
    public void display()
    {
        System.out.println("example␣id␣is:␣" + id + "␣"
                           + "and␣example␣name␣is:␣"
                           + name);
        System.out.println();
    }
}
```

Listing C.4: The primitive arraycopy code in backward direction

```
class test {
  static void backwardCopy_int(int[] a, int[] b)
  {
   for (int i=b.length-1; i>=0; i--)
    a[i] = b[i];
  }
  static void backwardCopy_byte(byte[] a, byte[] b)
  {
   for (int i=b.length-1; i>=0; i--)
    a[i] = b[i];
  }
  public static void main(String[] args) {
   int[] src = new int[16];
   for (int i=0; i<src.length; i++)
    src[i] = i+100;
   int[] dst = new int[src.length];

   // backwardArrayCopy wordElementArrayCopy limit=main
   // 64 bytes
   backwardCopy_int(dst, src);

   int[] x = new int[40];
   for (int i=0; i<x.length; i++)
    x[i] = i+100;
   int[] y = new int[x.length];

   // backwardArrayCopy wordElementArrayCopy limit=main
   // 160 bytes
   backwardCopy_int(y, x);

   int[] c = new int[25];
```

```
    for (int i=0; i<c.length; i++)
     c[i] = i+100;
    int[] d = new int[c.length];

    // backwardArrayCopy wordElementArrayCopy limit=main
    // 100 bytes
    backwardCopy_int(d, c);

    byte[] m =new byte[43];
    for (int i=0; i <m.length; i++)
     m[i] = (byte)(i+1);
    byte[] n = new byte[43];

    // backwardArrayCopy limit=main 43 bytes
    backwardCopy_byte(n, m);

    int[] l = {0, 1};
    int[] k = new int[2];
    backwardCopy_int(k, l); // 8 bytes
  }
}
```

Appendix D

IL code from trace files

This section includes snippets from the trace files generated using the tests mentioned in Appendix C. The listings D.1 to D.7 demonstrate arraycopy nodes of various array length, types and methods.

Listing D.1: System.arraycopy(a 1 b 1 3)

```
   n807n ( 0) treetop () [ 0xffff78977bf0] bci=[-1,492,5
8] rc=0 vc=9 vn=- li=196 udi=- nc=1 flg=0x8
   n806n ( 0) arraycopy java/lang/System.arraycopy(L
java/lang/Object;ILjava/lang/Object;II)V[#352 unresol
ved notAccessed static Method] [flags 0x400 0x0 ] (Un
signed forwardArrayCopy ) [ 0xffff78977ba0] bci=[-1,4
92,58] rc=0 vc=9 vn=- li=196 udi=- nc=3 flg=0x4028
   n682n ( 0) ==>aladd (in GPR_2100) (X>=0 internalP
tr sharedMemory )
  n754n ( 0) ==>aladd (in GPR_2109) (X>=0 internalP
tr sharedMemory )
  n2812n ( 0) lconst 3 (highWordZero X!=0 X>=0 ) [ 0x
ffff78c2eeb0] bci=[-1,491,58] rc=0 vc=9 vn=- li=196 u
di=- nc=0 flg=0x4104
 ------------------------------
    [ 0xffff5216b2c0] 492 ldrhimm GPR_2113, [GPR_2100, 0]
```

```
[ 0xffff5216b3e0] 492 strhimm GPR_2113, [GPR_2109, 0]
[ 0xffff5216b500] 492 ldrbimm GPR_2113, [GPR_2100, 2]
[ 0xffff5216b620] 492 strbimm GPR_2113, [GPR_2109, 2]
```

Listing D.2: The length between 4 and 127 bytes (with 100 bytes length)

```
[0xffff2e88f480] 13 addimmx GPR_0544, &GPR_0512, 16
[0xffff2e88f580] 14 addimmx GPR_0545, &GPR_0470, 16
[0xffff2e88f6f0] 13 addimmx GPR_0544, GPR_0544, 100
[0xffff2e88f780] 13 addimmx GPR_0545, GPR_0545, 100
[0xffff2e88f8a0] 13 vldrpreq VRF_0546, [GPR_0544, -16]
[0xffff2e88f9c0] 13 vstrpreq VRF_0546, [GPR_0545, -16]
[0xffff2e88fae0] 13 vldrpreq VRF_0546, [GPR_0544, -16]
[0xffff2e88fc00] 13 vstrpreq VRF_0546, [GPR_0545, -16]
[0xffff2e88fd20] 13 vldrpreq VRF_0546, [GPR_0544, -16]
[0xffff2e88fe40] 13 vstrpreq VRF_0546, [GPR_0545, -16]
[0xffff2e88ff60] 13 vldrpreq VRF_0546, [GPR_0544, -16]
[0xffff2e890090] 13 vstrpreq VRF_0546, [GPR_0545, -16]
[0xffff2e8901b0] 13 vldrpreq VRF_0546, [GPR_0544, -16]
[0xffff2e8902d0] 13 vstrpreq VRF_0546, [GPR_0545, -16]
[0xffff2e8903f0] 13 vldrpreq VRF_0546, [GPR_0544, -16]
[0xffff2e890510] 13 vstrpreq VRF_0546, [GPR_0545, -16]
[0xffff2e890630] 13 ldrprew GPR_0547, [GPR_0544, -4]
[0xffff2e890750] 13 strprew GPR_0547, [GPR_0545, -4]
```

Listing D.3: No loops (with 64 bytes length)

```
[0xffff791b0aa0] 53 addimmx GPR_0240, &GPR_0177, 16
[0xffff791b0ba0] 54 addimmx GPR_0241, &GPR_0198, 16
[0xffff791b0ca0] 53 addimmx GPR_0240, GPR_0240, 64
[0xffff791b0d30] 53 addimmx GPR_0241, GPR_0241, 64
[0xffff791b0e50] 53 vldrpreq VRF_0242, [GPR_0240, -16]
[0xffff791b0f70] 53 vstrpreq VRF_0242, [GPR_0241, -16]
[0xffff791b1090] 53 vldrpreq VRF_0242, [GPR_0240, -16]
```

```
 [0xffff791b11b0] 53 vstrpreq VRF_0242, [GPR_0241, -16]
 [0xffff791b12d0] 53 vldrpreq VRF_0242, [GPR_0240, -16]
 [0xffff791b13f0] 53 vstrpreq VRF_0242, [GPR_0241, -16]
 [0xffff791b1510] 53 vldrpreq VRF_0242, [GPR_0240, -16]
 [0xffff791b1630] 53 vstrpreq VRF_0242, [GPR_0241, -16]
```

Listing D.4: The case of 43 bytes long - for example: copying by 16+16+8+2+1 bytes

```
 [0xffff791ef770] 13 addimmx *GPR_0720, &GPR_0688, 16
 [0xffff791ef870] 14 addimmx GPR_0721, &GPR_0646, 16
 [0xffff791ef9e0] 13 addimmx *GPR_0720, *GPR_0720, 43
 [0xffff791efa70] 13 addimmx GPR_0721, GPR_0721, 43
 [0xffff791efb90] 13 vldrpreq VRF_0722, [*GPR_0720, -16]
 [0xffff791efcb0] 13 vstrpreq VRF_0722, [GPR_0721, -16]
 [0xffff791efdd0] 13 vldrpreq VRF_0722, [*GPR_0720, -16]
 [0xffff791efef0] 13 vstrpreq VRF_0722, [GPR_0721, -16]
 [0xffff791f0010] 13 ldrprex GPR_0723, [*GPR_0720, -8]
 [0xffff791f0130] 13 strprex GPR_0723, [GPR_0721, -8]
 [0xffff791f0250] 13 ldrhpre GPR_0723, [*GPR_0720, -2]
 [0xffff791f0370] 13 strhpre GPR_0723, [GPR_0721, -2]
 [0xffff791f0490] 13 ldrbpre GPR_0723, [*GPR_0720, -1]
 [0xffff791f05b0] 13 strbpre GPR_0723, [GPR_0721, -1]
```

Listing D.5: __bwWordArrayCopy

```
[ 0xffff7a894ca0] addx  w1, w1, w0
[ 0xffff7a894d30] addx  w2, w2, w0
[ 0xffff7a8954b0] assocreg[GPR_0226 : x14]
[ 0xffff7a894dc0] 0000FFFF7B78CFC0              ;
Direct Call "__bwWordArrayCopy"
PRE: [GPR_0224 : x0] [GPR_0256 : x1] [GPR_0257 : x2]
 [D_GPR_0258 : x3] [D_GPR_0259 : x4]
POST: [GPR_0224 : x0] [GPR_0256 : x1] [GPR_0257 : x2]
```

```
 [D_GPR_0258 : x3] [D_GPR_0259 : x4]
[ 0xffff7a8958e0] 17ffffc2 6    b       Label L0016
```

Listing D.6: The iteration64 greater than 1 (length of 160 bytes)

```
 [0xffff791cd5d0] 13 addimmx GPR_0464, &GPR_0432, 16
 [0xffff791cd6d0] 14 addimmx GPR_0465, &GPR_0390, 16
 [0xffff791cd7d0] 13 addimmx GPR_0464, GPR_0464, 160
 [0xffff791cd860] 13 addimmx GPR_0465, GPR_0465, 160
 [0xffff791cd960] 13 movzx GPR_0467, 0x0002
 [0xffff791cda50] 13 Label L0705:
 [0xffff791cdb60] 13 vldrpreq VRF_0466, [GPR_0464, -16]
 [0xffff791cdc80] 13 vstrpreq VRF_0466, [GPR_0465, -16]
 [0xffff791cdda0] 13 vldrpreq VRF_0466, [GPR_0464, -16]
 [0xffff791cdec0] 13 vstrpreq VRF_0466, [GPR_0465, -16]
 [0xffff791cdfe0] 13 vldrpreq VRF_0466, [GPR_0464, -16]
 [0xffff791ce100] 13 vstrpreq VRF_0466, [GPR_0465, -16]
 [0xffff791ce220] 13 vldrpreq VRF_0466, [GPR_0464, -16]
 [0xffff791ce340] 13 vstrpreq VRF_0466, [GPR_0465, -16]
 [0xffff791ce3d0] 13 subimmx GPR_0467, GPR_0467, 1
 [0xffff791ce460] 13 cbnzx GPR_0467, Label L0705
 [0xffff791ce580] 13 vldrpreq VRF_0466, [GPR_0464, -16]
 [0xffff791ce6a0] 13 vstrpreq VRF_0466, [GPR_0465, -16]
 [0xffff791ce7c0] 13 vldrpreq VRF_0466, [GPR_0464, -16]
 [0xffff791ce8e0] 13 vstrpreq VRF_0466, [GPR_0465, -16]
```

Listing D.7: Reference arraycopy

```
n2592n   (  0)  treetop ()  [    0xffff64f939f0] bci=[-1,1
65,31] rc=0 vc=9 vn=- li=167 udi=- nc=1 flg=0x8
n2593n   (  0)    arraycopy  java/lang/System.arraycop
y(Ljava/lang/Object;ILjava/lang/Object;II)V[#357
 unresolved notAccessed static Method] [flags 0x400 0
x0 ] (Unsigned forwardArrayCopy referenceArrayCopy )
```

```
 [    0xffff64f93a40] bci=[-1,165,31] rc=0 vc=9 vn=- li
=167 udi=- nc=5 flg=0x4028
n3685n   (  2)      ==>aRegLoad (in &GPR_3524) (X!=0
 SeenRealReference sharedMemory )
n3684n   (  2)      ==>aRegLoad (in &GPR_3523) (X!=0
 SeenRealReference sharedMemory )
n3682n   (  2)      ==>aRegLoad (in *GPR_3521) (X!=0
 SeenRealReference sharedMemory )
n3681n   (  2)      ==>aRegLoad (in *GPR_3520) (X!=0
 SeenRealReference sharedMemory )
n2561n   (  1)      ==>lconst 3996 (in GPR_3529)
 (highWordZero X!=0 X>=0 )
```

Vita

Candidate's full name: Siri Sahithi Ponangi
University attended:

- Bachelor of Technology in Computer Science Engineering, Keshav Memorial Institute of Technology, Hyderabad, India, 2017
- Master of Computer Science, University of New Brunswick, Canada, 2022

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`).

Accepted subject to minor revisions:

- S. S. Ponangi, G. W. Dueck, K. B. Kent, D. Maier, and K. Konno. "Java Runtime Optimization for Copying Arrays on AArch64." 11th Mediterranean Conference on Embedded Computing, Budva, Montenegro (virtual conference), (6 pages), 7-10 June 2022.

Publibation in progress:

- Non-Refereed Poster: S.S. Ponangi, A.G. Graham, K.B. Kent, G.W. Dueck, K. Konno and D. Maier, "Eclipse OpenJ9 and Eclipse OMR AArch64 Just-In-Time Compiler Optimizations," Poster & Video, IBM CASTLE (IBM Centre for Advanced Studies Technical Link Event) 2021, May 10-14, 2022.

Publications:

- Refereed Poster: S.S. Ponangi, G. Dueck, K.B. Kent, K. Konno, and D. Maier. "Optimized arrayCopy Implementation in Eclipse OMR-OpenJ9 for Improved Runtime Performance on AArch64." 31st Annual International Conference on Computer Science and Software Engineering (CASCON x EVOKE 2021), Markham, Canada, November 22-26, 2021.

- Refereed Poster: S. S. Ponangi, G. W. Dueck, K. B. Kent, K. Konno and D. Maier. "Optimized arrayCopy Implementation for Improved Runtime Performance for Java and OpenJ9 on AArch64." 30th Annual International Conference on Computer Science and Software Engineering (CASCON x EVOKE 2020), Markham, Canada, November 10-13, 2020.

- Non-Refereed Poster: 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," Poster & Video, IBM CASTLE (IBM Centre for Advanced Studies Technical Link Event) 2021, May 10-14, 2021.

- Non-Refereed Poster: Md.A. Noor, Md.A. Haque, S.S. Ponangi, A.G. Graham, J.-P. Legault, K.B. Kent, G.W. Dueck and D. Maier, "AArch64 Support for the Eclipse OMR Language Runtime Toolkit & Eclipse OpenJ9," Poster, IBM CASTLE (IBM Centre for Advanced Studies Technical Link Event) 2020, May 11-15, 2020.