computers

MDPI

Article

Accidental Choices—How JVM Choice and Associated Build Tools Affect Interpreter Performance

Jonathan Lambert [1,2,*,†], Rosemary Monahan [1,†] and Kevin Casey [1,†]

[1] Department of Computer Science, Maynooth University, Maynooth, Co. Kildare, Ireland; rosemary.monahan@mu.ie (R.M.); kevin.casey@mu.ie (K.C.)
[2] Mathematics Development and Support, National College of Ireland, Mayor Street, Dublin 1, Ireland
* Correspondence: jonathan.lambert@ncirl.ie; Tel.: +353-1-4498650
† These authors contributed equally to this work.

Abstract: Considering the large number of optimisation techniques that have been integrated into the design of the Java Virtual Machine (JVM) over the last three decades, the Java interpreter continues to persist as a significant bottleneck in the performance of bytecode execution. This paper examines the relationship between Java Runtime Environment (JRE) performance concerning the interpreted execution of Java bytecode and the effect modern compiler selection and integration within the JRE build toolchain has on that performance. We undertook this evaluation relative to a contemporary benchmark suite of application workloads, the Renaissance Benchmark Suite. Our results show that the choice of GNU GCC compiler version used within the JRE build toolchain statistically significantly affects runtime performance. More importantly, not all OpenJDK releases and JRE JVM interpreters are equal. Our results show that OpenJDK JVM interpreter performance is associated with benchmark workload. In addition, in some cases, rolling back to an earlier OpenJDK version and using a more recent GNU GCC compiler within the build toolchain of the JRE can significantly positively impact JRE performance.

Keywords: Java Runtime Environment; Java virtual machine; JVM template interpreter; JRE performance; GNU GCC compiler collection; GNU GCC effect on JRE performance; OpenJDK; compiler optimisations; Renaissance Benchmark Suite

Citation: Lambert, J.; Monahan, R.; Casey, K. Accidental Choices—How JVM Choice and Associated Build Tools Affect Interpreter Performance. *Computers* **2022**, *11*, 96. https://doi.org/10.3390/computers11060096

Academic Editor: Alberto Simões, Ricardo Queirós and Mário Pinto

Received: 6 May 2022
Accepted: 7 June 2022
Published: 14 June 2022

1. Introduction

The Java programming language has continued to cement its position as one of the most popular languages since its first release in 1995 [1–5]. Over the last quarter of a century, the language has evolved through approximately 18 major revisions [6]. Initially being developed with a model of execution that did not rely upon ahead-of-time compilation to machine instructions, its early architecture specifically relied upon an interpretation of its intermediate formats bytecode [7–9] with possible just-in-time (JIT) compilation [10]. The robustness of Java, its cross-platform capabilities and security features [11] has seen the Java Virtual Machine (JVM) in its early days be the target for many languages, for example, Ada [12], Eiffel [13], ML [14], Scheme [15], and Haskell [16] as cited in [17–19]. Nonetheless, performance analysis of early releases of the language was characterised as providing poor performance relative to traditional compiled languages [7,9], with early comparisons of execution time performance of Java code showing significant underperformance relative to C or C++ code [7,20–23]. The Java interpreter was especially slow [8,24] in contrast to code compiled to native code [25–27], or even Java JIT-compiled code [27], performance challenges that persist with contemporary language interpreters [28–30].

Over the last two decades, the language has integrated significant optimisations regarding its runtime environment. The JVM has transitioned from the separated execution models defined by the interpreter, client, and server virtual machines (VM) to a tiered

system that integrates all three. Previous to Java 7, an application's life cycle resided within an interpretive environment, the client or server VM. The tiered VM model of execution harnessing the performance opportunities associated with all three execution models [31]. Tiered execution allows an application to start quickly in the interpreter, transition to the client C1 compiler, and then to the server C2 compiler for extremely hot code. Some recent performance benchmarking of the JVM indicates that, for a small number of application cases, performance has surpassed that offered by ahead-of-time compiled code or is relatively no worse [4,32–39]. Regarding energy performance, Java has been listed as one of the top five languages [40,41]. However, underperformance relative to native code persists [37], with C++, for example, offering more significant performance opportunities [38].

The performance of the JVM interpreter remains of significant contemporary importance [42], not just because of its role within the modern JRE and the information that it provides for guiding just-in-time compilation, but also due to the extent to which it is relied upon for the execution of code on a vast array of platforms. In addition, the contemporary JVM is a popular target for many languages, primarily because of its ability to provide an environment to efficiently and competitively execute bytecode, for example; Scala, Groovy, Clojure, Jython, JRuby, Kotlin, Ceylon, X10, and Fantom [43]. The availability of JVM interpreters allows languages to target architectures without the need to construct dedicated compilers, leveraging decades of research and significantly reducing the costs associated with modern language development. Interpreters can substantially facilitate the development of modern languages, new features, and the testing of alternative paradigms. The Truffle language implementation framework is an example of a contemporary framework that enables the implementation of languages on interpreters that then utilise the GraalVM and its associated optimised JRE [44]. Also, the growing envelope of paradigms and workloads that target the JVM present new optimisation possibilities for current compilers [45,46]. In addition, interpreters are more adaptable than compilers [47], offering simplicity and portability [48], compact code footprint, significantly reduced hardware complexity, as well as a substantial reduction in their development costs [49]. Tasks such as debugging and profiling can be accomplished easier through an interpreter compared to a compiler [47]. Finally, expensive and non-trivial tasks such as register allocation and call-stack management, for example, are not required [50].

This paper considers the JRE and its associated interpreter as a target for optimisation opportunity discovery. We propose that the JVM interpreter, instead of other language interpreters, will benefit from such analysis. This is partly due to its maturity and the considerable amount of research that has been undertaken, from which optimisation techniques have been identified. We have focused on optimisation opportunities associated with changing the GNU compiler release used within the JRE build toolchain and the effect on the JRE and its associated JVM's interpreter. In this current work, we consider this question relative to six open source releases of OpenJDK, specifically; release versions 9 through 14; and the use of four GNU compiler release versions, specifically versions 5.5.0, 6.5.0, 7.3.0, and 8.3.0. We undertook the analysis on a cluster of Raspberry Pi4 Model B lightweight development boards. We undertake this analysis on a more recent contemporary benchmark suite, the Renaissance Benchmark Suite. The specific research questions motivating this work are as follows:

RQ1: When constrained to using a particular release of a JRE, does building with alternative compiler versions affect runtime performance?

RQ2: When free to migrate between JRE versions, which JRE provides the best runtime performance, and what build toolchain compiler was associated with the runtime performance gains?

RQ3: Do workload characteristics influence the choice of JRE and GCC compiler?

This paper is structured as follows. Section 2 presents the background that this research builds upon and the specific related work. In Section 3, we present our methodology.

We detail our results in Section 4. In Section 5, we present a discussion of our results. In the final section, Section 6, we present our conclusions and future work direction.

2. Background and Related Work

The Java interpreter still plays a fundamental role in the boundary of the significant structural changes integrated into the modern JVM. The interpreter acts as the entry point, controlling initial Java code execution, with the tiered JVM model activating greater levels of just-in-time compilation based on the profiling of the behaviour within the interpreter. Although, the interpreter's role within the lifespan of a Java application is not finished in the early stage of application execution. It can be delegated back to [51] when the later stages within tiered compilation identify previously hot methods for deoptimisation.

There are many ways by which Java interpreters have been implemented. The main difference across implementations is how instruction dispatch is handled. Three general strategies have dominated instruction dispatch implementations: switch-based, direct threading, and inline direct threading [48,52]. The switch-based model is the simplest of all. Nevertheless, it is the most inefficient due to the number of control transfer instructions, a minimum of three for each bytecode instruction. Direct threaded dispatch sees the internal opcode structure updated by the address of their implementation, resulting in the elimination of the instruction switch and the encapsulating dispatch loop. A differentiating characteristic between both approaches is that direct threaded code instructions control the program counter [53], whereas the switch expression takes responsibility for its updating. Finally, inline threaded implementations eliminate dispatch overhead associated with control transfer instructions by amalgamating instruction sequences contained within the bytecode's basic blocks. The effect eliminates the dispatch overhead associated with all instructions composing the basic block, bar the last bytecode instruction. Casey et al., in [52], and Gagnon et al., in [48] provide an overview of all three techniques. Early research on interpreter performance identified that instruction dispatching is primarily the most expensive bottleneck regarding bytecode execution.

2.1. Historical JVM Enhancements

Addressing the underperformance of the Java interpreter, the early work relative to the JVM deployment history concentrated on optimisation techniques focused on the cost implications associated with the interpreter dispatch loop, which bore a considerable cost penalty in simple switch-based implementations. Early historical optimisations of the Java interpreter that integrated direct threading were shown to provide significant performance gains over simple switch dispatch [54]. The findings concerning those threaded implementations report increases in runtime performance in the order of up to 200% over switch-based implementations [54]. Those results and observations agree with the earlier works by Ertl and Gregg in [55,56], in which they focused on the performance of a wide variety of interpreters such as Gforth, OCaml, Scheme48, YAP, and XLISP. Nevertheless, the JVM interpreter's direct-threading optimisation encounters underperformance regarding branch misprediction due to the large number of indirect branches introduced into direct threaded code [52].

Continuing to explore the effects of threaded interpreters and intending to reduce the dispatch overhead further, Gagnon et al., in [48] considered the inlining of Java bytecode instruction implementations that compose basic blocks. Their work reported significant performance gains from inline threaded interpretation over direct threaded. A side effect of direct-threaded interpretation is that the number of indirect branches considerably impacts microarchitectural performance, for example, the branch target buffer (BTB) [52]. The work of Casey et al., in [52] addressed those limitations by implementing instruction replication and extending beyond basic block boundaries with super-instructions. Nevertheless, aggressive superinstruction implementations increase cache miss-rates [57]. Continuing to explore opportunities associated with threading: context-threaded interpretation of Java bytecode was proposed by Berndl et al., in [57]. Context-threading eliminates the

indirect branches but adds a subroutine call and a return from that subroutine call. These instructions are optimised on modern architectures and outperform indirect branch-heavy direct threaded code.

Microarchitectural characteristics associated with JVM interpreter behaviour, particularly instruction and data cache effects, have been the focus in [53–56]. Earlier work by Hsieh et al., in [24] had highlighted that the interpreter bytecode is considered data along with the data associated with the application and is accessible through the data caches, ultimately providing more opportunity for increases in data cache misses for the interpreter.

Java's early underperformance had motivated JVM designers to look at native execution alternatives. Microprocessor designs, such as PicoJava and MicroJava, facilitated the native execution of Java bytecode [58]. Due to cost constraints, PicoJava and MicroJava lacked instruction-level parallelism (ILP) circuitry. The work of Watanabe and Li in [25] proposed an alternative architecture incorporating ILP circuitry. The JVM top-of-stack is a considerable bottleneck that inhibits ILP exploitation of the Java bytecode [25,59]. Li et al., in [59] propose a virtual register architecture within Java processor designs due to the distinct shortage of physical registers. The debate on whether a stack architecture or a register architecture can provide a more efficient implementation for interpreters has also been considered [60]. Work by Shi et al., [60] highlights many advantages of a virtual register architecture compared to a more traditional stack architecture.

2.2. Evolution of OpenJDK

Since its first release in 1995, the Java language and, in particular, the JVM specification have been implemented and shipped by many vendors (Sun, CA, USA; Oracle, TX, USA; IBM, NY, USA; Microsoft, WC, USA; AZUL, CA, USA). The codebase of the Sun implementation started its journey to being open-source in 2006. In 2007 the Open Java Development Kit (OpenJDK) was founded, which serves as the reference implementation of the platform [61].

The openjdk.java.net (accessed on 5 May 2022) details the evolution of changes to the components defining OpenJDK through a catalogue of JDK Enhancement Proposals (JEP) that detail all OpenJDK changes from OpenJDK version 8 (OpenJDK8) through to the latest OpenJDK version 19 (OpenJDK19). The proposals serve as an "authoritative", "record" detailing "what" has "changed" within an OpenJDK version and "why" that change was accepted and implemented [62].

The JEP index currently lists 326 enhancement proposals. Of those, 149 (46%) relate to OpenJDK9 through 14. Of those, 58 (18%) enhancement proposals concern proposed changes to OpenJDK10 through to 14. A JEP proposal is associated with a JDK component, for example, core (detailing changes targeted at core libraries), spec (detailing changes targeted at a specification), tools, security, hotspot, and client (detailing changes targeted at client libraries). Components also have associated sub-components; for example, hotspot has associated sub-components for GC (garbage collector changes), runtime, compiler (for changes related to the JIT-compilers), and JVM tool interface changes.

A number of the more significant changes that have been completed and delivered as part of the JEP, specifically related to the JRE, have seen the OpenJDK version 10 (OpenJDK10) G1 garbage collector (G1GC) updated to allow for the parallelisation of full collections. The initial implementation of G1GC "avoided full collections" and fell back to a full garbage reclamation, using a mark-sweep-compact algorithm when the JVM did not release memory quickly enough. Specifically, OpenJDK10 saw the mark-sweep-compact algorithm become fully parallel [63].

OpenJDK version 12 (OpenJDK12) implemented two further enhancements to the G1GC: "abortable mixed collections" [64], and optimisation in regards to the "speedy reclamation of committed memory" [65]. Regarding abortable mixed collections, before the OpenJDK12, G1GC collections, once started, could not terminate until they reclaimed all live objects across all the regions defined within its collection set were reclaimed. The implemented enhancement saw the G1GC collection set divided into two regions, one mandatory

and the other optional when "pause time targets" had not been exceeded [64]. Regarding speedy reclamation defined through JEP346, the implementation enhanced memory management through the release of heap memory back to the operating system when the JVM was in an idle state. This implementation targeted cloud-based services where the financial cost of application execution included the cost of physical resources such as memory [65]. Further enhancements concerning memory management and JVM configurations involved the enhancement of the G1GC in OpenJDK version 14 (OpenJDK14). The enhancement ensured that the G1GC performance was increased for JVM configurations running on multi-socket processor cores, particularly for cores that implement non-uniform memory access (NUMA) strategies [66].

Before OpenJDK10, OpenJDK gave attention to reducing the "dynamic memory footprint" across multiple JVM's. In that regard, OpenJDK implemented Class Data Sharing archives. Although those archives were limited to bootstrap classes, access was only available through the bootstrap class loader [67]. OpenJDK10 expanded class data sharing to allow application class data sharing across multiple JVMs. With that said, to gain the benefits from application class data sharing, the JVM runtime flag `-XX:+UseAppCDS` needs to be passed to the JVM. OpenJDK12 and, in particular, JEP341 [68] turned on application class data sharing at the build stage of the OpenJDK. Finally, JEP350 [69] allows the dumping of application class data when an application completes execution, an enhancement first implemented in OpenJDK version 13 (OpenJDK13).

A list of JDK enhancement proposals related to OpenJDK9 through 14 for the hotspot-runtime and hotspot-gc components are listed in Table 1. The reader can find a detailed list of all JEP enhancements affecting all OpenJDK components at [62].

Table 1. A listing of the major OpenJDK JDK Enhancement Proposals, completed and delivered, that have been integrated into versions of OpenJDK10 through 14 releases.

.

OpenJDK10		OpenJDK11		OpenJDK12		OpenJDK13		OpenJDK14	
JEP286:	Local-Variable Type Inference	**JEP309**:	Dynamic Class-File Constants	**JEP325**:	Switch Expressions	**JEP350**:	Dynamic CDS Archives	**JEP305**:	Pattern Matching for `instanceof`
JEP304:	Garbage Collector Interface	**JEP323**:	Local-Variable Syntax for Lambda Parameters	**JEP341**:	Default CDS Archives	**JEP354**:	Switch Expressions	**JEP345**:	NUMA-Aware Memory Allocation for G1
JEP307:	Parallel Full GC for G1	**JEP331**:	Low-Overhead Heap Profiling	**JEP345**:	Abortable Mixed Collections for G1			**JEP352**:	Non-Volatile Mapped Byte Buffers
JEP310:	Application Class-Data Sharing			**JEP346**:	Promptly Return Unused Committed Memory from G1			**JEP361**:	Switch Expressions
JEP312:	Thread-Local Handshakes								
JEP316:	Heap Allocation on Alternative Memory Devices								
JEP319:	Root Certificates								

2.3. Performance Regression

Focusing specifically on performance measurement of the effects of altering and integrating different build tools within the JDK build cycle, the literature is considerably void. That said, the build specifications associated with OpenJDK versions list GNU GCC compiler versions known to work concerning its building [70]. These specify that versions of OpenJDK should build successfully with GNU GCC native compiler versions from at least 5.0 through to 11.2. OpenJDK does not indicate their performance or regression effects when integrated into the OpenJDK build toolchain.

With that said, the most recent insights suggest that conventional compilers and optimisers fail to produce optimal code [71]. In addition, a more recent focus on the GNU GCC compiler collection has shown that the GNU GCC release used within a build toolchain can produce significantly different performance [72]. The work of Marcon et al., although not analysing JRE performance, analyses the effects of changing the GNU GCC compiler version in the build toolchain for the Geant4 simulation toolkit [73] (as cited in [72]), a simulation toolkit that is used for the analysis of high energy physics experiments and predominately used for experiments performed on the Large Hadron Collider. Their work considered three versions of GNU GCC, versions 4.8.5, 6.2.0 and 8.2.0; and the four standard GCC compiler optimisation flags, `-O1`, `-O2`, `-O3`, and `-Os`. From a high level, their results show that building with a more recent version of GCC can increase performance in the order of 30% in terms of execution time. Marcon et al. also considered the effects associated with static and dynamic compilation. Their results show statistically significant differences in performance between the statically and dynamically compiled systems. Their work also reports less profound effects from switching between GCC 6.2.0 and 8.2.0. The greatest effects are associated relative with GCC 4.8.5.

To our knowledge, no other work has addressed and measured the effects associated with switching GCC compiler in a build toolchain relative to a large codebase such as that of OpenJDK. In addition, no analysis of such changes are available relative to a contemporary set of workloads.

Although little work has concentrated on the effects of changing the underlying compiler toolkit associated with building OpenJDK, the literature has considered and reported the impact that Java-to-bytecode compilers have on producing optimised bytecode. For example, Daly et al., in [19] consider the effect that the variant of the Java compiler has in regards to its emitted Java bytecode. Their findings showed that different Java-to-bytecode compilers implement vastly different bytecode optimisations. Similar to the work presented in [19], Horgan et al., [74] consider the effects associated with Java-to-bytecode compiler choice and the dynamic instruction execution frequencies. Their results have shown that Java-to-bytecode compiler choice produces minimum dynamic differences in bytecode usage. The work of Gregg et al., in [54] is similar to that presented in [19] in that they compared the differences in bytecode distributions associated with different Java-to-bytecode compilers. Their findings showed that most Java-to-bytecode compilers fail to implement what is regarded as the simplest of optimisations to the emitted bytecode, for example, loop optimisation by removing tail `goto` branches. Gregg et al. reported that the Java-to-bytecode compilers have little effect on the dynamic bytecode frequencies.

Contemporary work has reported that Java source code compilation to JVM bytecode by the javac compiler returns minor optimisations and no aggressive optimisations on the intermediate bytecode representation. Such optimisation opportunities are delegated to the JIT-compiler [38]. Furthermore, switching javac compiler versions is not an explanatory factor for execution speed variations in Java applications running on JVMs [19]. As such, optimising bytecode to benefit the interpreter needs to be performed between source compilation and initial execution, for example; [22,75,76] or within the interpretive environment itself. More importantly, the delegation of bytecode optimisation to the JIT compiler results in a loss of opportunities that might positively affect interpreted execution.

2.4. This Study

Although previous studies have considered optimisations associated with the underperformance of the JVM interpreter and the effect that different variants of Java-to-bytecode compilers have on bytecode performance. None to our knowledge have evaluated the impact the compiler used within the JRE build toolchain has on the performance of the JRE and its associated JVM interpreter. As outlined previously, this is of contemporary importance for many reasons. Although, and as cited in [9], Cramer et al., in [8] identified that for all the optimisations targeting interpretation/execution of actual bytecode, approximately 35% of execution time is spent doing other tasks other than interpreting/executing bytecodes,

for example, memory management, synchronisation, and running native methods. Of all these mentioned tasks, object synchronisation is identified by Gu et al., in [7] as being the most costly, accounting for approximately 58% of non-bytecode execution. All of those execution tasks are controlled, defined and implemented across the full JRE implementation; this, in part, has motivated this work in that we investigate several questions relative to OpenJDK JRE performance. Specifically, what effects does altering GNU GCC versions in the OpenJDK build toolchain have on JRE performance. In this regard, this research evaluates three hypotheses:

Hypothesis 1 (H1). *There are statistically significant differences in OpenJDK JRE and associated JVM interpreter performance depending on the GNU GCC version used in the build toolchain.*

Hypothesis 2 (H2). *There are statistically significant differences in performance across all OpenJDK JREs and their associated JVM interpreters, and those differences are in part associated with the version of GNU GCC used in the versions build toolchain.*

Hypothesis 3 (H3). *Workload characteristics influence the choice of JRE and GCC compiler.*

3. Methods

This section provides an overview of the JDK, JRE, JVM, and GNU GCC implementations analysed in this study. We also provide an overview of the collection of benchmark application workloads relied upon for assessing JRE and JVM interpreter performance. We follow this with a specification of the hardware cluster and operating system on which we evaluated JRE runtime environment performance. We then present an a priori analysis of the stability of each cluster machine relative to each other. Finally, we summarise the statistical procedures used to generate our main results.

3.1. JDK, JRE, JVM, and GNU GCC Implementations

We analysed six major version releases of the OpenJDK and, specifically, their associated JREs regarding their performance evolution. In particular, OpenJDK9 through to 14 JREs. We built all OpenJDK versions from source, with four builds for each JRE using GNU GCC compiler release versions 5.5.0, 6.5.0, 7.3.0, and 8.3.0. All JDK builds used a previous immediate version for the build requirement bootstrap JDK. The initial build of OpenJDK9 relies on the pre-built OpenJDK8 binary distribution. All subsequent OpenJDK versions built relied upon one of our built JDK and were used as the bootstrap JDK. The building of OpenJDK10, for example, depends on our OpenJDK9 built from source. We downloaded all source bundles from the OpenJDK project's repository, which the reader can find at https://openjdk.java.net (accessed on 5 May 2022). We downloaded all GNU Compiler Collection releases from https://gcc.gnu.org (accessed on 5 May 2022).

Specifically, regarding the JRE JVM interpreter, two implementation choices are available at build time: the production Template interpreter and a C++ interpreter bundled primarily for ease of implementation and testing of proposed research optimisations. In this work, we focus our concentration on the behaviour of the JRE with an integrated Template Interpreter. Interpreter mode was communicated to each JVM using the `-Xint` runtime flag. Concerning JVM tuning, and in line with benchmark specifications such as SPECjvm2008, no additional JVM tuning is activated [77], except for communicating `-Xms` for heap space allocation, which we have controlled at 2 GB across all JVM variants. An evaluation of all JEP showed that OpenJDK had implemented no fundamental changes regarding the JVM Template Interpreter, and the interpreter can be considered constant across OpenJDK9 through to 14.

We used the default Garbage-First Garbage Collector (G1GC) for heap management across all JRE JVM executions. Changes to the behaviour and operation of the G1GC have occurred across OpenJDK10 through to OpenJDK14. In particular, OpenJDK10 saw the parallelisation of full G1GC cycles; OpenJDK12 saw the introduction of abortable G1GC

collections and the speedy release of memory back to the operating system when the JVM was idle, and finally, OpenJDK14 saw the introduction of "NUMA-Aware memory allocation." Concerning the parallelisation of full G1GC cycles, we initiated all JRE instances with the default number of threads set to 4 by default across all OpenJDK, from OpenJDK10 through 14. In addition, NUMA awareness does not impact the operational characteristics of the JRE and associated JVM interpreters assessed in this study, as the Raspberry Pi 4 model B system-on-chip is not a multi-socket processor. The speedy reclamation of committed memory implemented in OpenJDK12 applies to cases where the JVM is idle, which is relevant for cloud-based applications and not relevant in our system setting.

No JVM warmup iterations were required, as all JVM executions were run in interpreter mode and "the interpreter does not need a warmup" [42]. We ran all benchmark workloads and their iterations on newly initialised JVMs.

3.2. Benchmarks

We assessed all JRE and their associated JVM interpreter runtime performance using the collection of 24 workloads composing the Renaissance Benchmark suite version 11 [78]. The Renaissance Benchmark Suite workloads are partitioned into six classes of applications. Applications targeted for execution on the Apache Spark framework, applications that implement a concurrent programming paradigm, database applications, several applications developed using a functional paradigm and compiled to Java bytecode, web applications serving requests to HTTP servers, and several applications written in the Scala language with their compilation targeting Java bytecode. Table 2 presents an overview of each application workload.

Table 2. A listing of the 24 applications composing the Renaissance Benchmark suite, subdivided into application category, alongside a small description of each application. Source: [78].

Category	Application	Description
Apache-spark	`chi-square`	Chi-square test from Spark MLlib.
	`dec-tree`	Random Forest algorithm from the Spark ML library.
	`gauss-mix`	Gaussian mixture model using expectation-maximization.
	`log-regression`	Logistic Regression algorithm from the Spark ML library.
	`movie-lens`	Recommends movies using the ALS algorithm.
	`naive-bayes`	Multinomial Naive Bayes algorithm from the Spark ML library.
	`page-rank`	PageRank iterations, using RDDs.
Concurrency	`akka-uct`	Unbalanced Cobwebbed Tree actor workload in Akka.
	`fj-kmeans`	K-means algorithm using the fork/join framework.
	`reactors`	Benchmarks inspired by the Savina microbenchmark workloads in a sequence on reactors.IO.
Database	`db-shootout`	Shootout test using several in-memory databases.
	`neo4j-analytics`	Neo4J graph queries against a movie database.
Functional	`future-genetic`	Genetic algorithm using the Jenetics library and futures.
	`mnemonics`	Solves the phone mnemonics problem using JDK streams.
	`par-mnemonics`	Solves the phone mnemonics problem using parallel JDK streams.
	`rx-scrabble`	Solves the Scrabble puzzle using the Rx streams.
	scrabble	Solves the Scrabble puzzle using JDK Streams.
Scala	`dotty`	Dotty compiler on a set of source code files.
	`philosophers`	Solves a variant of the dining philosophers problem using ScalaSTM.
	`scala-doku`	Sudoku Puzzles using Scala collections.
	`scala-kmeans`	K-Means algorithm using Scala collections.
	scala-stm	stmbench7 benchmark using ScalaSTM.
Web	`finagle-chirper`	Microblogging service using Twitter Finagle.
	`finagle-http`	Finagle HTTP requests to a Finagle HTTP server and awaits response.

To assess JRE and JVM interpreter runtime performance: each of the 24 Renaissance benchmark application workloads was executed 20 times across all 24 JRE JVM builds, resulting in 11,520 execution runs. In total, we undertook 94 system days of data acquisition.

Each JRE version accounts for 1920 of those runs. Individual JREs built using one of the four GCC compiler versions account for 480 runs each.

We observed several failed workload runs, specifically for the Renaissance Benchmark application `db-shootout` that failed to run on all JVM versions after OpenJDK11. In addition, several other applications reported deprecated API features, reporting that their dependencies cannot be guaranteed to be supported in later JVM runs, specifically after OpenJDK9. That said, none of the deprecated API features had been removed from the assessed OpenJDK versions.

3.3. Hardware

Our experimental set-up consists of a cluster of 10 Raspberry Pi4 Model B development boards, integrated with Broadcom BCM2711, Quad-core Cortex-A72 (ARM v8) 64-bit system-on-chip processors with an internal main reference clock speed of 1.5 GHz. Each Raspberry Pi has a 32 KB L1 data cache, a 48 KB L1 instruction cache per core, 1 MB L2 cache, and 4 GB of LPDDR4-3200 SDRAM. Network connectivity is provided by 2.4 GHz and 5.0 GHz IEEE 802.11ac wireless interfaces, Bluetooth 5.0, 2 USB 3.0 ports and 2 USB 2.0 ports. Each Raspberry Pi provides general-purpose input-output (GPIO) functionality through a 40 pin GPIO header. A Raspberry Pi can be powered over a 5V DC USB type C connector or 5v DC GPIO headers. All Raspberry Pi's were powered from mains. The maximum tolerable current is 3 Amperes [79]. A specification diagram is presented in Figure 1. We ran each Raspberry Pi4 Model B with the Raspbian Buster Light Debian distribution of Linux, Kernel version 4.19. We loaded the operating system (OS) through an embedded Micro-SD card slot. All Raspberry Pi's had the same OS clone installed. The Raspbian Buster Light distribution is a non-windowing version providing only a command-line interface. All Raspberry Pi4 were SSH enabled.

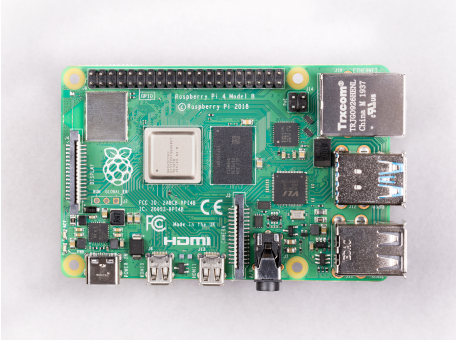

Figure 1. Raspberry Pi4 Model B board peripheral device layout. Detailing position of Broadcom BCM2711 processor, RAM, USB Type-C power supply, and network ports. Source: [80].

3.4. Runtime Stability across Cluster Nodes

A random sample of five applications were selected from the Renaissance Benchmark Suite, namely: `chi-square`, `future-genetic`, `neo4j-analytics`, `scala-kmeans`, and `scrabble`. A single build of OpenJDK9, built with GNU GCC version 8, was distributed across six of the cluster's Raspberry Pi4. We executed the benchmark samples on each of the six Raspberry Pi4. We conducted an analysis of variance on their respective execution times. At the 5% significance level, we observed no statistically significant differences in runtime performance for four of the benchmark applications, all $p > 0.05$. The single benchmark case `scala-kmeans` did show differences in runtime performance between the six Raspberry Pi4 development boards ($p < 0.05$). With that said, the average runtime across the `scala-kmeans` benchmark is approximately 155 s; the observed difference from the single Raspberry Pi4 accounts for an effect of approximately 1.97 s (1.28%). Albeit cognoscente of the small effect difference associated with the `scala-kmeans` benchmark, predominately benchmark application runs across the Raspberry Pi4 development boards are stable.

3.5. JRE and JVM Build Stability

A more recent concern regarding consistency, reliability, and robustness of the analysis of runtime systems has raised an important question regarding the effects of application build order. In particular, questioning the determinism concerning the layout of an application in memory. An effect that can be attributed to application build cycle linking and later application loading into memory. Focusing on the impact attributable to build cycle linking, four homogeneous builds of OpenJDK9, built with GNU GCC compiler release version 7, were analysed. The results from an analysis of variance showed no statistically significant differences in application runtime performance across the four homogeneous builds ($p > 0.05$). An indicator that runtime environment stability is reliable and that there are no differences in OpenJDK JRE and associated JVM performance.

3.6. Statistical Analysis

To determine if there are differences in JRE performance depending on the GNU compiler build version used within the JDK build toolchain, we conducted an analysis of variance of the runtime behaviour of the 24 applications composing the Renaissance Benchmark suite. In the cases where statistically significant differences in JRE performance has been identified, follow-up Tukey HSD post hoc tests were carried out to identify specific differences between pair-wise comparisons of JRE build versions built with one of the four GNU compilers running on individual OpenJDK versions.

We also conducted a two-way analysis of variance to assess the interaction effects between the GNU GCC compiler version and the OpenJDK version. The analysis of variance evaluated the effects associated with updates to OpenJDK versions and updates to GNU GCC versions.

4. Results

4.1. The Effect of GCC Evolution on Performance

In this section, we present the results from an analysis of the runtime characteristics of Renaissance Benchmark applications and the performance of the six JRE and associated JVM interpreter versions distributed with OpenJDK9, OpenJDK10, OpenJDK11, OpenJDK12, OpenJDK13, and OpenJDK14. Each JRE binary was built from source by integrating one of four GNU GCC compilers (gcc and g++) within the respective OpenJDK build toolchains. In particular, each JRE was built with the GNU GCC compiler versions 5.5.0 (gcc-5 and g++-5), 6.5.0 (gcc-6 and g++-6), 7.3.0 (gcc-7 and g++7), and 8.3.0 (gcc-8 and g++-8). In total, 24 JREs were assessed. Our results are presented in Tables 3–8 and Figures 2–7. In all table cases, the first column (JVM) lists the OpenJDK version considered, the second column (Benchmark) lists the Renaissance benchmark for which a statistically significant difference in performance was observed, the third column (GCC) lists the two GCC compiler versions compared (the first in the pairing is the fastest, with the second being relatively slower), the fourth column (M_diff) lists the mean difference in performance, this is followed by associated 95% confidence intervals for the mean difference, the penultimate column lists the observed p-value for the test of difference, and the final column (Gain) lists the percentage gain to be achieved by switching build toolchain compiler to the fastest.

4.1.1. OpenJDK9

Of 24 analyses of variances undertaken on the runtime behaviour of four of the OpenJDK9 JRE, 7 (29%) of the 24 analyses showed differences in OpenJDK9 runtime behaviour based on the GNU compiler version used within the OpenJDK9 build toolchain. At the application workload level, follow-up Tukey HSD post hoc tests showed that of the possible 144 pairwise comparisons (6 for each application: GCC5-v-GCC6, GCC5-v-GCC7, ..., GCC7-v-GCC8) 14 (10%) showed that GCC build version influenced JRE runtime behaviour, the results of which are presented in Table 3. Performance differences ranging between 0.8% and 5.9% were observed. The Renaissance `db-shootout` benchmark application workload had the most significant impact on OpenJDK9 JRE performance based

on the choice of GCC compiler used within its build toolchain. GCC compiler versions 6 and 7 provide those gains relative to GCC versions 5 and 8. In contrast, GNU GCC compiler version 8 significantly increased JRE performance for 6 (43%) of the 14 differences observed.

Table 3. A listing of Tukey HSD post hoc test results for the differences between OpenJDK9 JRE builds and the associated benchmark application loads eliciting those differences. Mean differences and confidence intervals reported in seconds.

JRE	Benchmark	GCC	M_{diff}	95%CI	p	Gain (%)
j09	`akka-uct`	GCC8-GCC5	−13	(−25, −2)	0.020	1.210
j09	`akka-uct`	GCC8-GCC6	−14	(−26, −3)	0.011	1.297
j09	`chi-square`	GCC7-GCC5	−8	(−13, −3)	0.000	1.158
j09	`chi-square`	GCC8-GCC5	−6	(−11, −1)	0.007	0.856
j09	`chi-square`	GCC7-GCC6	−8	(−13, −3)	0.001	1.098
j09	`chi-square`	GCC8-GCC6	−6	(−10, −1)	0.014	0.796
j09	`db-shootout`	GCC6-GCC5	−39	(−61, −18)	0.000	4.283
j09	`db-shootout`	GCC7-GCC5	−54	(−75, −32)	0.000	5.929
j09	`db-shootout`	GCC6-GCC8	36	(15, 57)	0.000	3.907
j09	`dec-tree`	GCC7-GCC5	−3	(−6, 0)	0.025	0.872
j09	`dotty`	GCC8-GCC5	−3	(−6, 0)	0.017	0.822
j09	`finagle-chirper`	GCC8-GCC6	−2	(−5, 0)	0.044	0.898
j09	`scala-kmeans`	GCC5-GCC7	2	(0, 4)	0.010	1.264
j09	`scala-kmeans`	GCC6-GCC7	2	(1, 4)	0.004	1.383

For a descriptive comparative analysis of the execution time performance of each OpenJDK9 JRE build, irrespective of a finding of statistical significance, Figure 2 presents grouped box-and-whisker plots illustrating a side-by-side comparison of the OpenJDK JRE execution times observed across GCC versions 5, 6, 7, and 8 for each benchmark workload.

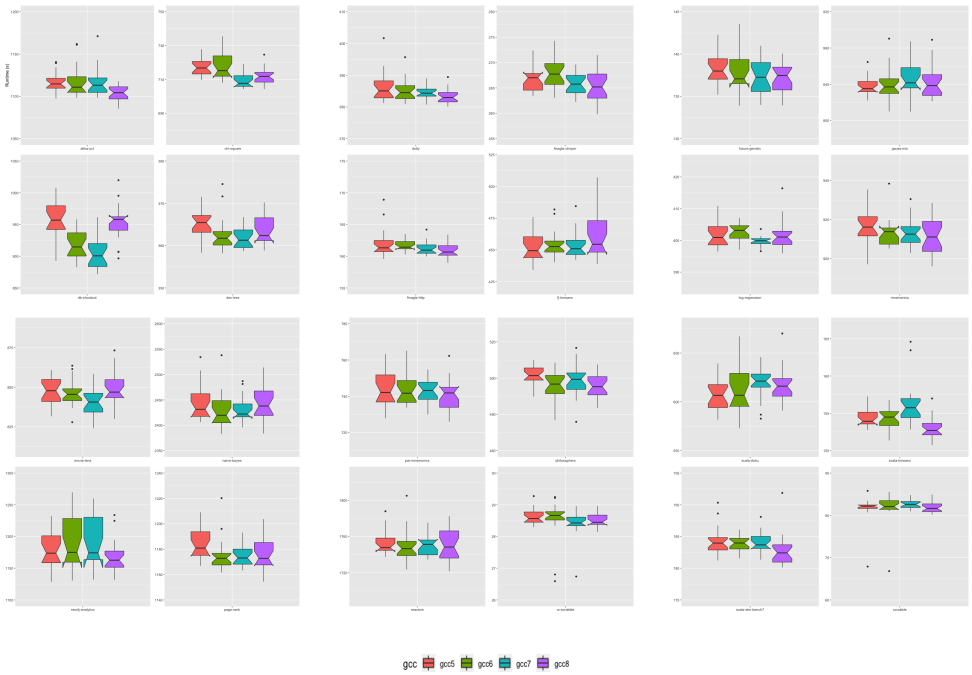

Figure 2. Execution time analysis of the 24 Renaissance benchmark suite applications. Each panel depicting the runtime performance for each OpenJDK9 JRE built with the four GNU GCC compiler versions 5, 6, 7, and 8.

4.1.2. OpenJDK10

Considering OpenJDK10 JREs and their associated JVM interpreters, the 24 analyses of variances performed provided evidence of statistically significant differences in OpenJDK10 performance for 7 (29%) analyses undertaken. At the application level, 14 (10%) pairwise

comparisons between OpenJDK10 JREs showed differences in runtime behaviour based on the GCC version used within the build toolchain for the JVM. Of all comparisons, no evidence was found to suggest any differences in performance between OpenJDK10 JREs and their associated JVM interpreter built using GCC version 7 or 8. The complete set of results for OpenJDK10 JRE builds are detailed in Table 4. Of the small number of pairwise comparisons showing performance differences, those differences ranged between 0.6% and 3.0%. Except for a single workload, that of `scala-kmeans`, compared across compiler GCC versions 5 and 6, GCC compiler versions 7 and 8 predominately provided better performance over GCC versions 5 and 6.

Table 4. A listing of Tukey HSD post hoc test results for the differences between OpenJDK10 JRE builds and the associated benchmark application loads eliciting those differences. Mean differences and confidence intervals reported in seconds.

JRE	Benchmark	GCC	M_{diff}	95%CI	p	Gain (%)
j10	`chi-square`	GCC7-GCC5	−8	(−14, −3)	0.001	1.089
j10	`chi-square`	GCC7-GCC6	−6	(−11, 0)	0.043	0.746
j10	`dotty`	GCC8-GCC5	−2	(−4, 0)	0.008	0.572
j10	`dotty`	GCC8-GCC6	−2	(−4, −1)	0.004	0.604
j10	`finagle-http`	GCC7-GCC5	−2	(−5, 0)	0.045	1.394
j10	`finagle-http`	GCC7-GCC6	−3	(−5, −1)	0.006	1.764
j10	`movie-lens`	GCC5-GCC8	9	(3, 15)	0.002	1.070
j10	`naive-bayes`	GCC7-GCC5	−53	(−78, −27)	0.000	1.919
j10	`naive-bayes`	GCC7-GCC6	−48	(−73, −22)	0.000	1.737
j10	`scala-kmeans`	GCC5-GCC6	2	(0, 4)	0.031	1.345
j10	`scala-kmeans`	GCC8-GCC5	−2	(−4, −1)	0.005	1.653
j10	`scala-kmeans`	GCC8-GCC6	−4	(−6, −2)	0.000	3.021
j10	`scala-stm`	GCC7-GCC6	−2	(−3, 0)	0.039	0.866
j10	`scala-stm`	GCC8-GCC6	−2	(−3, 0)	0.012	0.997

A descriptive comparative analysis of the execution time performance of each OpenJDK10 JRE build, irrespective of a finding of statistical significance, is presented in Figure 3.

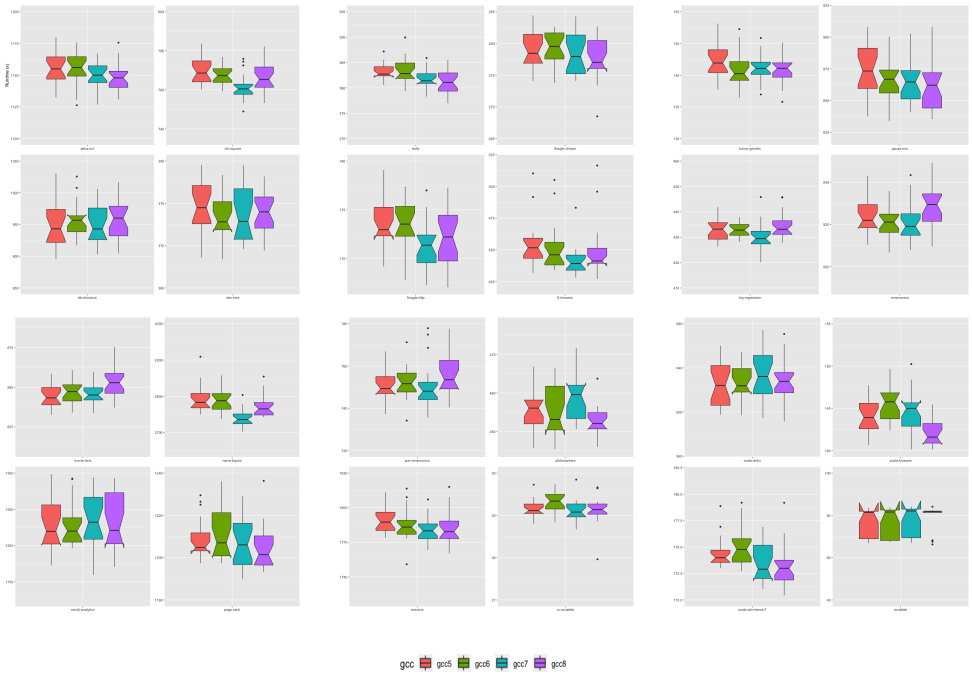

Figure 3. Execution time analysis of the 24 Renaissance benchmark suite applications. Each panel depicting the runtime performance for each OpenJDK10 JRE built with the four GNU GCC compiler versions 5, 6, 7, and 8.

4.1.3. OpenJDK11

We observed the most significant frequency of JRE performance differences for OpenJDK11 JREs and their associated JVM interpreter, particularly OpenJDK11 built with GCC versions 5, 6, 7, and 8. Of the 24 analyses of variance, 17 (71%) showed statistically significant differences in OpenJDK11 JRE performance. Of all the 144 possible pairwise comparisons of OpenJDK11 builds, 36 (25%) showed differences in JVM performance based on the GCC compiler version used within the JVM build toolchain. Of the statistically significant differences observed, more recent GNU compilers have provided better performance gain opportunities. The GCC compiler version 7 was the best compiler in 17 (47%) of the 36 cases, GCC compiler version 8 was identified as providing the best performance in 12 (33%) cases. Our results identified GCC compiler version 6 as giving the best OpenJDK11 performance in 7 (20%) cases. Performance differences ranged between 0.7% and 4.6%. Similar to OpenJDK9 JVM behaviour, the Renaissance application `db-shootout` had benefited most, with performance gain opportunities ranging between 2% and 4.6% Table 5.

Table 5. A listing of Tukey HSD post hoc test results for the differences between OpenJDK11 JRE builds and the associated benchmark application loads eliciting those differences. Mean differences and confidence intervals reported in seconds.

JRE	Benchmark	GCC	M_{diff}	95%CI	p	Gain (%)
j11	akka-uct	GCC7-GCC5	−20	(−34, −5)	0.004	1.683
j11	chi-square	GCC6-GCC5	−7	(−12, −2)	0.004	0.905
j11	chi-square	GCC7-GCC5	−6	(−11, 0)	0.029	0.733
j11	chi-square	GCC8-GCC5	−11	(−16 , −6)	0	1.418
j11	db-shootout	GCC6-GCC5	−26	(−45, −7)	0.004	2.527
j11	db-shootout	GCC7-GCC5	−46	(−65, −27)	0.000	4.608
j11	db-shootout	GCC8-GCC5	−35	(−53, −17)	0.000	3.485
j11	db-shootout	GCC7-GCC6	−20	(−39, −1)	0.034	2.031
j11	dec-tree	GCC6-GCC5	−3	(−6, −1)	0.014	0.911
j11	dec-tree	GCC7-GCC5	−5	(−8, −3)	0.000	1.445
j11	dec-tree	GCC8-GCC5	−4	(−7, −1)	0.004	1.013
j11	dotty	GCC7-GCC5	−3	(−5, −1)	0.001	0.932
j11	dotty	GCC8-GCC5	−4	(−7, −2)	0.000	1.303
j11	dotty	GCC8-GCC6	−3	(−5, −1)	0.002	0.835
j11	finagle-chirper	GCC7-GCC5	−3	(−5, 0)	0.034	0.987
j11	finagle-http	GCC7-GCC6	−2	(−5, 0)	0.041	1.369
j11	finagle-http	GCC8-GCC6	−3	(−5, −1)	0.009	1.598
j11	future-genetic	GCC6-GCC7	2	(0, 3)	0.031	1.272
j11	log-regression	GCC7-GCC5	−5	(−10, −1)	0.003	1.355
j11	log-regression	GCC8-GCC5	−5	(−9, −1)	0.005	1.269
j11	log-regression	GCC7-GCC6	−4	(−8, 0)	0.039	1.036
j11	mnemonics	GCC7-GCC6	−10	(−17, −3)	0.003	1.027
j11	mnemonics	GCC8-GCC6	−10	(−17, −3)	0.002	1.055
j11	movie-lens	GCC6-GCC5	−10	(−17, −4)	0.001	1.190
j11	movie-lens	GCC7-GCC5	−12	(−19, −6)	0.000	1.433
j11	movie-lens	GCC8-GCC5	−11	(−17, −4)	0.000	1.230
j11	naive-bayes	GCC7-GCC5	−45	(−71, −19)	0.000	1.760
j11	naive-bayes	GCC8-GCC5	−43	(−69, −18)	0.000	1.708
j11	neo4j-analytics	GCC6-GCC5	−34	(−65, −2)	0.031	2.745
j11	neo4j-analytics	GCC7-GCC5	−33	(−65, −2)	0.035	2.696
j11	neo4j-analytics	GCC8-GCC5	−36	(−67, −5)	0.016	2.944
j11	page-rank	GCC7-GCC5	−13	(−22, −4)	0.002	1.083
j11	page-rank	GCC7-GCC6	−9	(−18, 0)	0.038	0.787
j11	par-mnemonics	GCC7-GCC6	−8	(−15, −1)	0.020	1.041
j11	philosophers	GCC8-GCC5	−6	(−12, −1)	0.023	1.328
j11	scala-kmeans	GCC8-GCC5	−2	(−4, 0)	0.012	1.674
j11	scala-kmeans	GCC6-GCC7	3	(1, 5)	0.004	1.885

In Figure 4 we present a set of box-and-whisker plots of the execution time performance of OpenJDK11 under load from each of the 24 Renaissance benchmark workloads.

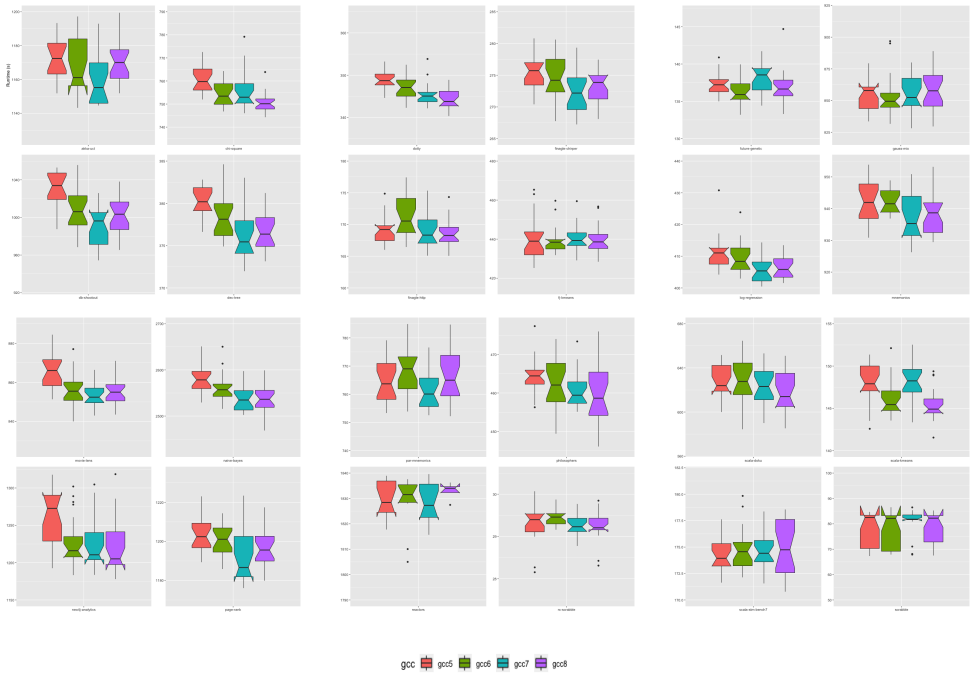

Figure 4. Execution time analysis of the 24 Renaissance benchmark suite applications. Each panel depicting the runtime performance for each OpenJDK11 JRE built with the four GNU GCC compiler versions 5, 6, 7, and 8.

4.1.4. OpenJDK12

Analysis of variances performed on the OpenJDK12 JVM interpreters showed statistically significant differences in their performance for 4 (17%) of the 24 analyses. The results associated with follow-up Tukey HSD post hoc comparisons are detailed in Table 6. Of 144 possible pairwise comparisons, 6 (4%) showed statistically significant differences in pairwise comparisons of OpenJDK12 builds depending on GCC compiler choice. Performance differences ranged between 0.4% and 1.4% were observed. No difference in performance between OpenJDK12 versions built with GCC versions 7 and 8 has been observed.

Table 6. A listing of Tukey HSD post hoc test results for the differences between OpenJDK12 JRE builds and the associated benchmark application loads eliciting those differences. Mean differences and confidence intervals reported in seconds.

JRE	Benchmark	GCC	M_{diff}	95%CI	p	Gain (%)
j12	`dec-tree`	GCC7-GCC6	−3	(−5, 0)	0.016	0.688
j12	`dotty`	GCC5-GCC6	1	(0, 3)	0.047	0.399
j12	`dotty`	GCC5-GCC7	1	(0, 3)	0.031	0.424
j12	`dotty`	GCC5-GCC8	2	(0, 3)	0.011	0.465
j12	`finagle-chirper`	GCC7-GCC5	−4	(−7, −1)	0.003	1.390
j12	`finagle-chirper`	GCC8-GCC5	−4	(−6, −1)	0.002	1.387
j12	`finagle-chirper`	GCC7-GCC6	−4	(−7, −1)	0.002	1.441
j12	`finagle-chirper`	GCC8-GCC6	−4	(−7, −1)	0.001	1.438
j12	`log-regression`	GCC8-GCC6	−4	(−8, 0)	0.046	1.026

A descriptive overview of the execution time performance of each OpenJDK12 JRE, built with GNU GCC versions 5 through 9 are depicted in Figure 5. In particular, we present side-by-side box-and-whisker plots illustrating execution time distributions for each of the Renaissance benchmark workloads.

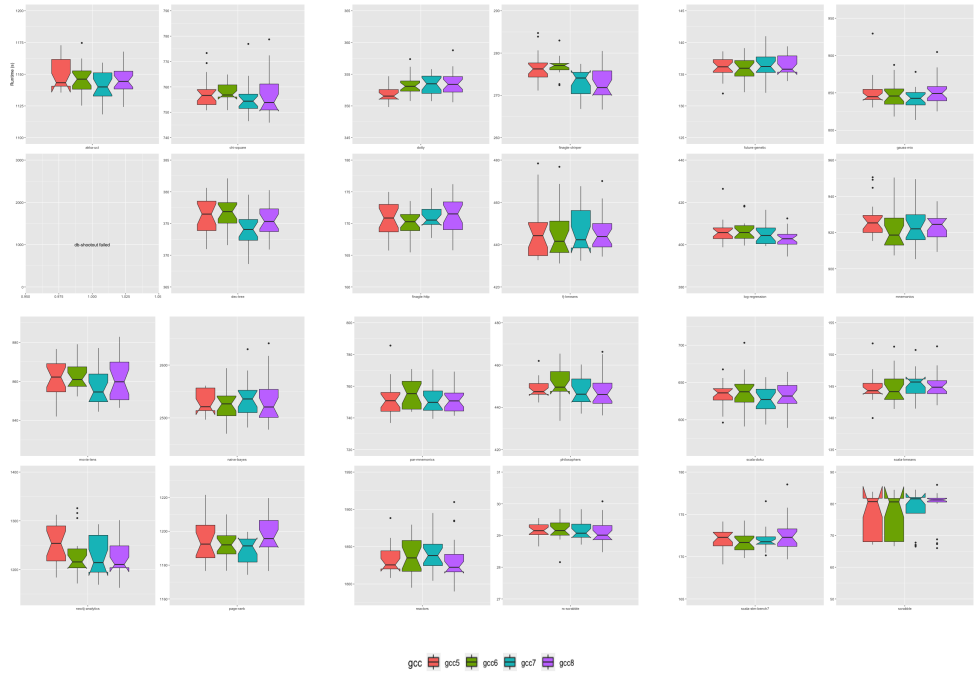

Figure 5. Execution time analysis of the 24 Renaissance benchmark suite applications. Each panel depicting the runtime performance for each OpenJDK12 JRE built with the four GNU GCC compiler versions 5, 6, 7, and 8.

4.1.5. OpenJDK13

Focusing on OpenJDK13 (Table 7), 6 (25%) of the 24 analyses of variance showed statistically significant differences in OpenJDK13 performance based on the GCC compiler version used within the OpenJDK13 build toolchain. Follow up post hoc Tukey HSD tests identified 18 (13%) differences in performance from the possible 144 pairwise comparisons. Performance differences ranged between 0.6% and 3.5%. All four GCC compiler versions were represented across all statistically significant comparisons.

Table 7. A listing of Tukey HSD post hoc test results for the differences between OpenJDK13 JRE builds and the associated benchmark application loads eliciting those differences. Mean differences and confidence intervals reported in seconds.

JRE	Benchmark	GCC	M_{diff}	95%CI	p	Gain (%)
j13	`dotty`	GCC7-GCC5	−3	(−4, −1)	0.000	0.840
j13	`dotty`	GCC8-GCC5	−2	(−3, 0)	0.006	0.554
j13	`dotty`	GCC7-GCC6	−2	(−4, −1)	0.000	0.717
j13	`dotty`	GCC8-GCC6	−1	(−3, 0)	0.046	0.432
j13	`fj-kmeans`	GCC5-GCC7	12	(2, 23)	0.014	2.329
j13	`fj-kmeans`	GCC5-GCC8	15	(5, 25)	0.001	2.846
j13	`movie-lens`	GCC6-GCC5	−8	(−15, −1)	0.025	0.797
j13	`movie-lens`	GCC6-GCC8	11	(4, 18)	0.001	1.074
j13	`philosophers`	GCC7-GCC5	−18	(−29, −6)	0.001	3.525
j13	`philosophers`	GCC8-GCC5	−16	(−28, −5)	0.001	3.279
j13	`scala-kmeans`	GCC6-GCC5	−2	(−4, 0)	0.044	1.228
j13	`scala-kmeans`	GCC7-GCC5	−3	(−5, −1)	0.000	1.932
j13	`scala-kmeans`	GCC8-GCC5	−3	(−5, −1)	0.000	2.229
j13	`scala-stm`	GCC6-GCC5	−3	(−4, −1)	0.000	1.531
j13	`scala-stm`	GCC7-GCC5	−6	(−7, −4)	0.000	3.291
j13	`scala-stm`	GCC8-GCC5	−5	(−6, −3)	0.000	2.646
j13	`scala-stm`	GCC7-GCC6	−3	(−4, −2)	0.000	1.733
j13	`scala-stm`	GCC8-GCC6	−2	(−3, −1)	0.002	1.098

In Figure 6 we present a number of panels depicting side-by-side box-and-whisker plots for each of the GNU GCC versions used in building OpenJDK13 JREs. Each panel, depicting the execution time distributions for each of the 24 Renaissance benchmark workloads.

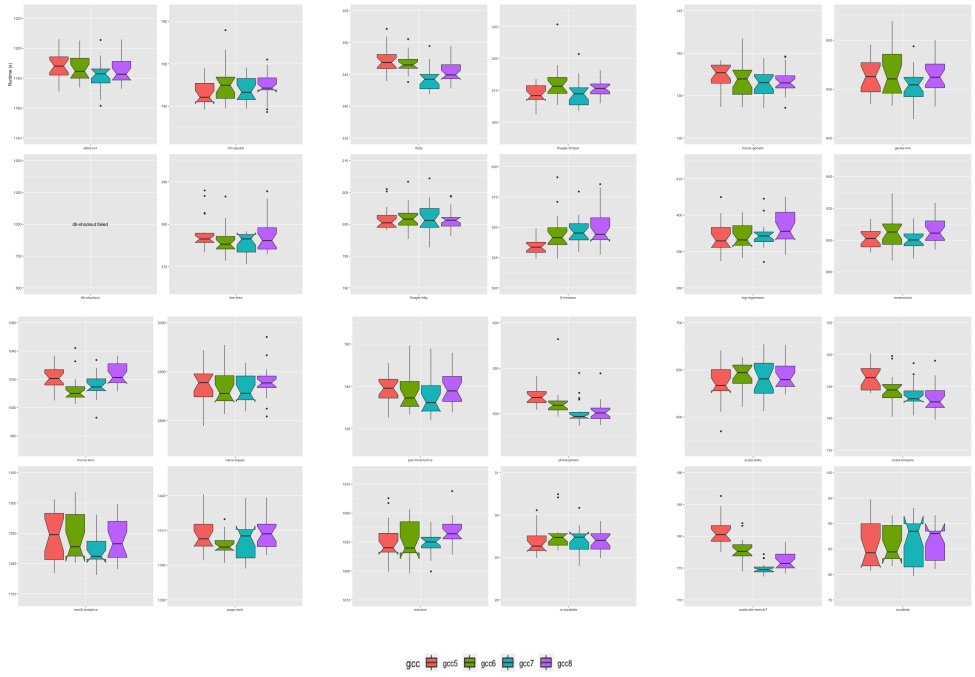

Figure 6. Execution time analysis of the 24 Renaissance benchmark suite applications. Each panel depicting the runtime performance for each OpenJDK13 JRE built with the four GNU GCC compiler versions 5, 6, 7, and 8.

4.1.6. OpenJDK14

In Table 8 we present the results associated with OpenJDK14 builds. Of the 24 analyses of variance of OpenJDK JREs and their associated JVM interpreters, 5 (21%) were associated with statistically significant differences in OpenJDK14 runtime performance depending on the GCC compiler used within the source build toolchain. Across all GCC release version pairwise comparisons, 9 (6%) showed differences in performance. Performance differences ranged between 0.6% and 3.5%.

Table 8. A listing of Tukey HSD post hoc test results for the differences between OpenJDK14 JRE builds and the associated benchmark application loads eliciting those differences. Mean differences and confidence intervals reported in seconds.

JRE	Benchmark	GCC	M_{diff}	95%CI	p	Gain (%)
j14	`chi-square`	GCC7-GCC5	−7	(−11, −3)	0.000	0.951
j14	`dotty`	GCC7-GCC5	−2	(−4, 0)	0.006	0.584
j14	`dotty`	GCC7-GCC6	−2	(−4, −1)	0.002	0.635
j14	`finagle-chirper`	GCC7-GCC5	−3	(−5, 0)	0.043	0.985
j14	`fj-kmeans`	GCC8-GCC5	−10	(−19, −1)	0.022	2.267
j14	`fj-kmeans`	GCC7-GCC6	−11	(−20, −2)	0.010	2.539
j14	`fj-kmeans`	GCC8-GCC6	−15	(−24, −6)	0.000	3.471
j14	`scala-stm`	GCC7-GCC5	−1	(−3, 0)	0.012	0.857
j14	`scala-stm`	GCC7-GCC6	−2	(−3, 0)	0.003	0.983

In Figure 7 we present a collection of panels depicting OpenJDK14 JRE execution time performance under load from each of the 24 Renaissance benchmark workloads. Each panel illustrating the execution time performance associated with the four GNU GCC versions.

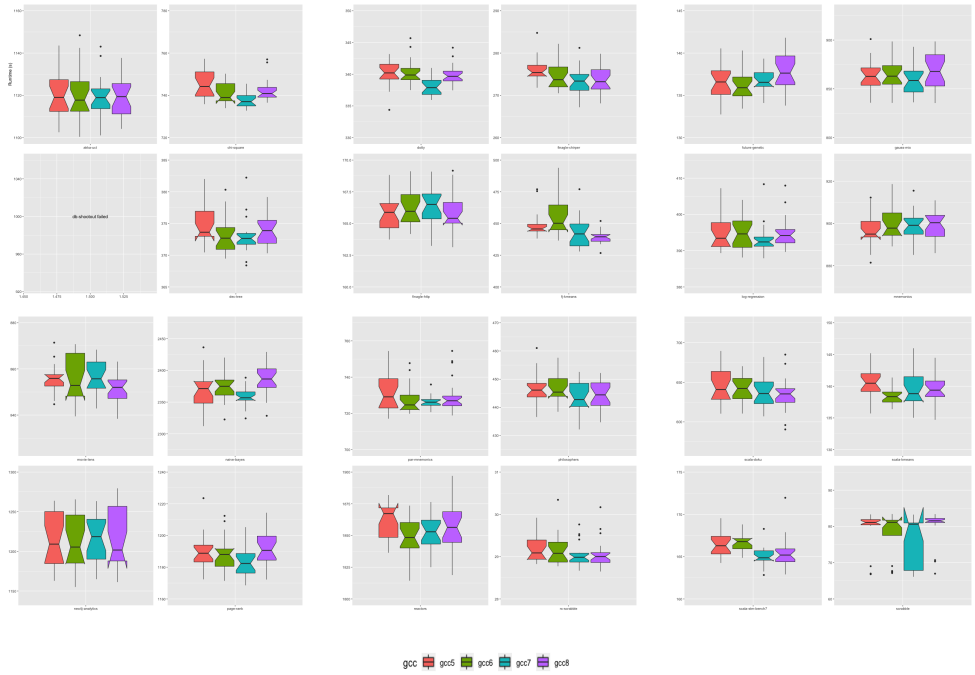

Figure 7. Execution time analysis of the 24 Renaissance benchmark suite applications. Each panel depicting the runtime performance for each OpenJDK14 JRE built with the four GNU GCC compiler versions 5, 6, 7, and 8.

4.2. Best Performing JRE Build for Each OpenJDK Source Bundle

The previous subsections presented detailed results for all statistically significant differences from the pairwise comparison between the four GNU GCC compiler versions used within OpenJDK JRE build toolchains. In some cases, our results have shown statistically significant differences between multiple GNU GCC compilers for the same workload. For example, in Table 8 we have shown that for the `fj-kmeans` benchmark workload that both OpenJDK14 JREs built with GCC8 and GCC7 outperform those built with GCC5 and GCC6. This subsection identifies the best overall performing OpenJDK JRE build and the associated GCC compiler used in its build toolchain. Specifically, we present the sequence of GCC compiler versions, listed from best through to worst. The results are presented in Table 9. For example, OpenJDK9 JRE and its associated JVM interpreter built with GCC version 7 had the best runtime performance while stressed by the `chi-square` workload. The second-best performing OpenJDK JRE was the OpenJDK9 JRE built with GCC version 8. The third-best performing OpenJDK JRE was the OpenJDK9 JRE built with GCC version 6. Regarding the worst-performing OpenJDK9 release, was OpenJDK9 built with GCC 5. Instances where no listings are provided, indicate no statistically significant differences between GCC compiler versions.

Table 9. Rank ordering from best to worst GCC compiler version for each OpenJDK JRE version. Application categories (Cat) are listed, where (A) represents Apache-spark, (C) Concurrency, (D) Database, (F) Functional, (S) Scala, and (W) Web based applications. After this, there is the application name (App), and then the six OpenJDK releases analyzed.

		JRE					
Cat	**App**	**j09**	**j10**	**j11**	**j12**	**j13**	**j14**
(A)	chi-square	7-8-6-5	7-8-6-5	8-6-7-5			7-6-8-5
	dec-tree	7-6-8-5		7-8-6-5	7-8-5-6		
	gauss-mix						
	log-regression			7-8-6-5	8-7-5-6		
	movie-lens		5-7-6-8	7-8-6-5		6-7-5-8	
	naive-bayes		7-8-6-5	7-8-6-5			
	page-rank			7-8-6-5			
(C)	akka-uct	8-7-5-6		7-6-8-5			
	fj-kmeans					5-6-7-8	8-7-5-6
	reactors						
(D)	db-shootout	7-6-8-5		7-8-6-5			
	neo4j-analytics			8-6-7-5			
(F)	future-genetic			6-8-5-7			
	mnemonics			8-7-5-6			
	par-mnemonics			7-5-8-6			
	rx-scrabble						
	scrabble						
(S)	dotty	8-7-6-5	8-7-5-6	8-7-6-5	5-6-7-8	7-8-6-5	7-8-5-6
	philosophers			8-7-6-5		7-8-6-5	
	scala-doku						
	scala-kmeans	8-6-5-7	8-5-7-6	8-6-5-7		8-7-6-5	
	scala-stm		8-7-5-6			7-8-6-5	7-8-5-6
(W)	finagle-chirper	8-7-5-6		7-8-6-5	7-8-5-6		7-8-6-5
	finagle-http		7-8-5-6	8-7-5-6			

In Figure 8 we present collections of stacked bar-plots for each OpenJDK JRE version, each panel depicting the individual stacked bars representing a ranking of the GNU GCC compilers, the longer the bar the better the resulting performance, for each benchmark application. In cases where stacked bars are of equal height, those bars indicate that there were no statistically significant differences for effect of the GNU GCC version used within the OpenJDK JRE build toolchain under load from the associated benchmark workload.

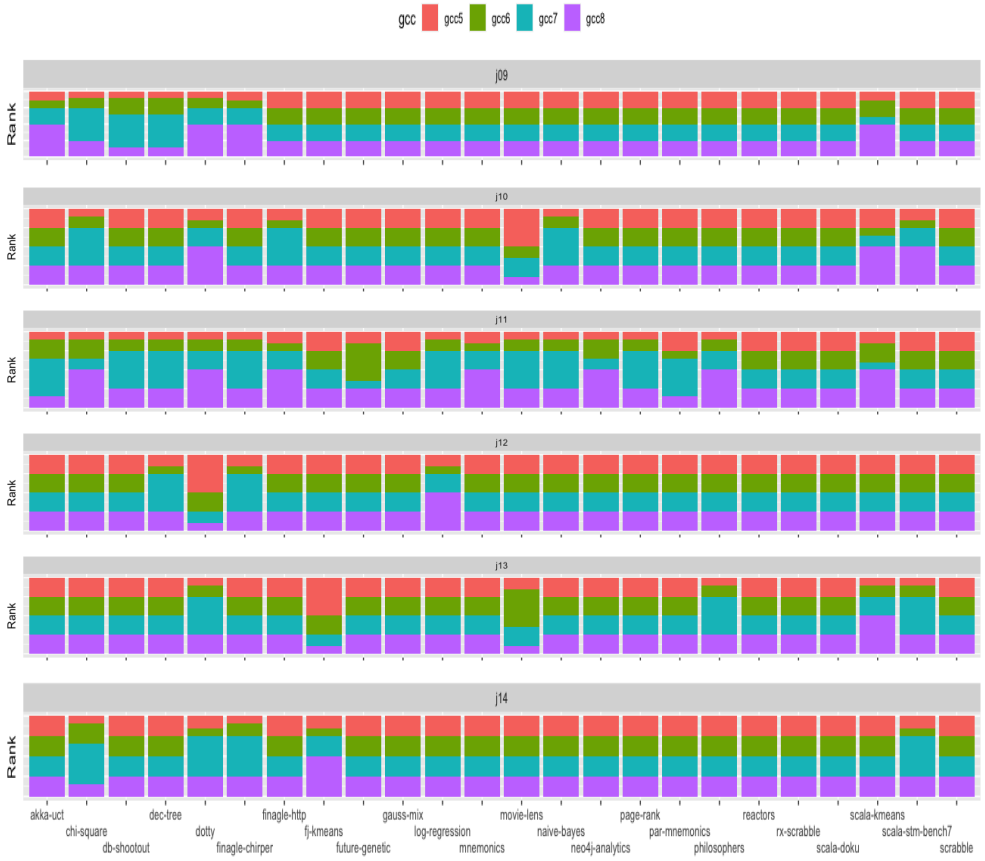

Figure 8. Stacked barplots depicting a ranking of each GNU GCC version for each benchmark workload. Each panel depicting the rankings for each OpenJDK JRE release.

4.3. Interaction Effects between GNU GCC Version and OpenJDK JRE Releases

In this subsection we present the results from an analysis of the interaction effects between GNU GCC version and OpenJDK JRE release and their effect on JRE runtime performance. In particular, we present the results associated with a two-way analysis of variance, undertaken at the level of individual Renaissance benchmark workload. The results of those statistically significant interactions are presented in Table 10 along with the individual impact of both GNU GCC version and OpenJDK JRE release.

Table 10. Details of the results associated two-way ANOVA that examined the effect of GNU GCC version and OpenJDK JRE release on JRE runtime performance.

Benchmark	Source of Variation	Df	SS	MSS	F	$Pr(>F)$
`akka-uct`	JRE	5	330,274	66,055	394.08	***
	GCC	3	3667	1222	7.29	***
	JRE:GCC	15	5659	377	2.25	**
	Residual	474	79,452	168		
`chi-square`	JRE	5	136,792	27,358	695.36	***
	GCC	3	1651	550	13.99	***
	JRE:GCC	15	2127	142	3.6	***
	Residual	474	18,649	39		
`db-shootout`	JRE	3	45,427,126	15,142,375	22,355.72	***
	GCC	3	31,696	10,565	15.6	***
	JRE:GCC	8	36,587	4573	6.75	***
	Residual	294	199,137	677		

Table 10. *Cont.*

Benchmark	Source of Variation	Df	SS	MSS	F	$Pr(>F)$
`db-shootout`	JRE	3	45,427,126	15,142,375	22,356.72	***
	GCC	3	31,696	10,565	15.6	***
	JRE:GCC	8	36,587	4573	6.75	***
	Residual	294	199,137	677		
`dotty`	JRE	5	162,612	32,522	6090.16	***
	GCC	3	303	101	18.9	***
	JRE:GCC	15	333	22	4.16	***
	Residual	474	2531	5		
`finagle-chirper`	JRE	5	98,380	19,676	1772.37	***
	GCC	3	428	143	12.86	***
	JRE:GCC	15	302	20	1.81	*
	Residual	474	5262	11		
`finagle-http`	JRE	5	80,601	16,120	3181.5	***
	GCC	3	54	18	3.53	*
	JRE:GCC	15	185	12	2.43	**
	Residual	474	2402	5		
`fj-kmeans`	JRE	5	642,104	128,421	524.9	***
	GCC	3	701	234	0.96	0.414
	JRE:GCC	15	11,620	775	3.17	***
	Residual	474	115,967	245		
`log-regression`	JRE	5	82,256	16,451	813.39	***
	GCC	3	400	133	6.58	***
	JRE:GCC	15	742	49	2.45	**
	Residual	474	9587	20		
`mnemonics`	JRE	5	102,008	20,402	258.9	***
	GCC	3	476	159	2.01	0.111
	JRE:GCC	15	2965	198	2.51	**
	Residual	474	37,352	79		
`movie-lens`	JRE	5	1,900,161	380,032	5223.56	***
	GCC	3	1603	534	7.34	***
	JRE:GCC	15	4355	290	3.99	***
	Residual	474	34,485	73		
`naive-bayes`	JRE	5	15,432,623	3,086,525	3059.23	***
	GCC	3	28529	9510	9.43	***
	JRE:GCC	15	51,442	3429	3.4	***
	Residual	469	473,184	1009		
`page-rank`	JRE	5	3,028,739	605,748	4540.5	***
	GCC	3	2809	936	7.02	***
	JRE:GCC	15	3733	249	1.87	*
	Residual	465	62,036	133		
`philosophers`	JRE	5	296,446	59,289	889.78	***
	GCC	3	2429	810	12.15	***
	JRE:GCC	15	3567	238	3.57	***
	Residual	465	30,985	67		
`reactors`	JRE	5	1,526,650	305,330	911.97	***
	GCC	3	645	215	0.64	0.588
	JRE:GCC	15	8609	574	1.71	*
	Residual	465	155,683	335		
scala-kmeans	JRE	5	9964	1993	402.43	***
	GCC	3	237	79	15.94	***
	JRE:GCC	15	408	27	5.49	***
	Residual	465	2303	5		
`scala-stm-bench7`	JRE	5	14,254	2851	927.04	***
	GCC	3	152	51	16.43	***
	JRE:GCC	15	347	23	7.52	***
	Residual	465	1430	3		

* indicates that $p < 0.05$. ** indicates that $p < 0.01$. *** indicates that $p < 0.001$.

Of the 24 two-way analysis of variance conducted, 16 (67%) showed statistically significant interactions between the GNU GCC version and OpenJDK JRE release on JRE runtime performance. The workloads that showed no interaction effects were: `dec-tree`, `future-genetic`, `gauss-mix`, `neo4j-analytics`, `par-mnemonics`, `rx-scrabble`, `scala-doku`, `scrabble`. Of those workloads 4 (50%) are classified as implementations adhering to a functional paradigm.

4.4. Best Performing JRE Build across All JVM Source Bundles

In the previous subsection, we presented our findings regarding the interaction effects associated with GNU GCC version and OpenJDK JRE build. In this section we detail the results from comparing all OpenJDK JRE GCC builds and their performances at the benchmark application workload level, identifying the best overall OpenJDK JRE build for each benchmark application workload. The analysis identified that OpenJDK9, OpenJDK12, and OpenJDK14 provided the best performance opportunities for application execution on their JRE and associated JVM interpreters. OpenJDK10 and 11 do not feature in this list. All GNU GCC compiler versions are an essential determinant, although; only relative to specific OpenJDK versions. Our results show that upgrading or downgrading the OpenJDK version can significantly affect performance, and upgrading or downgrading the GNU GCC compiler also affects performance. The complete set of results is presented in Table 11.

Table 11. Listings of the best performing OpenJDK JRE and associated JVM interpreter and the associated GNU compiler used within its build toolchain. The JVM-GCC column lists the OpenJDK JRE version and GCC lists the compiler version, for example, the J9G7 mnemonic represents the OpenJDK9 JRE built with the GNU compiler release version 7. The Min and Max columns list the minimum and maximum execution times, respectively. The Min_g and Max_g list the minimum and maximum expected percentage performance gains calculated relative to all other JVM builds.

Cat	App	JVM-GCC	Min	Max	Min_g (%)	Max_g (%)
(A)	`chi-square`	J9G7	712	770	0.2993	8.4700
	`dec-tree`	J9G7	363	382	0.2860	5.5288
	`gauss-mix`	J9G5	840	875	0.3191	4.5187
	`log-regression`	J14G7	394	434	0.1618	10.4412
	`movie-lens`	J9G7	844	1024	0.4546	21.7986
	`naive-bayes`	J14G7	2371	2879	0.5042	22.0627
	`page-rank`	J9G7	1175	1407	0.0226	19.6997
(C)	`akka-uct`	J9G8	1116	1188	1.1826	7.7073
	`fj-kmeans`	J14G8	439	549	0.0487	25.1022
	`reactors`	J9G8	1750	1934	0.0246	10.5718
(D)	`db-shootout`	J9G7	918	1036	1.5785	14.6811
	`neo4j-analytics`	J9G8	1180	1260	1.1334	7.9697
(F)	`future-genetic`	J12G6	136	142	0.1237	4.6159
	`mnemonics`	J14G5	899	948	0.2551	5.7351
	`par-mnemonics`	J14G7	728	769	0.1885	5.8624
	`rx-scrabble`	J9G7	29	29	0.3785	3.7780
	`scrabble`	J12G5	76	86	0.9335	14.3167
(S)	`dotty`	J14G7	340	386	0.5357	14.2982
	`philosophers`	J14G7	444	519	0.2206	17.1383
	`scala-doku`	J9G5	612	646	0.8467	6.3272
	`scala-kmeans`	J14G6	140	156	0.9683	12.9829
	`scala-stm`	J14G7	166	184	0.3791	11.6980
(W)	`finagle-chirper`	J9G8	266	312.19	0.0510	17.5514
	`finagle-http`	J9G8	161	201.04	0.2090	24.9095
				Mean	0.4627	12.4069
				Standard Deviation	0.4228	6.7492

4.5. Overall Observed Effects

Further analysis of the distribution of performance gain opportunities identified average performance gain opportunities of approximately 4.90% with an associated standard

deviation of approximately 4.89%. Our results show the minimum gain in performance at 0.02% ranging through to a maximum performance opportunity of approximately 25.10%. Our results show a median performance gain of approximately 3.58%, with associated first and third quartile measures being 1.81% and 5.74%, respectively. A histogram of the overall distribution is presented in Figure 9.

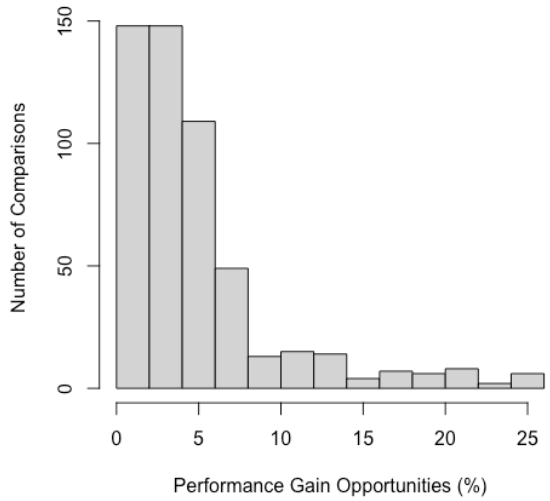

Figure 9. The distribution of observed performance gain opportunities across the complete set of benchmark applications.

5. Discussion

This paper has focused on optimisation opportunities associated with switching GNU compiler versions within the JDK build toolchain. In this current work, we considered this question relative to six open source releases of OpenJDK, specifically; release versions 9 through 14; and the use of four GNU compiler release versions in the OpenJDK build toolchain, specifically versions 5.5.0, 6.5.0, 7.3.0, and 8.3.0.

Our analyses are multi-faceted; first, we have undertaken a within-JRE-JVM version analysis, focusing on the effect of a change of the GNU compiler release version within the build toolchain for specific OpenJDK JRE and their associated JVM interpreter. We believe our analysis is of interest to those constrained to a particular OpenJDK JRE and JVM. In addition, this tests the hypothesis that compiler selection, at build time, affects overall JRE performance. Second, we have undertaken a between-JRE-JVM version analysis and considered the overall effect of changing the GNU compiler release version within the JVM build toolchain. We have identified a pair of OpenJDK JRE with GNU compiler that provides the best performance based on workload type. In addition, this tests the hypothesis that there exists an optimum pairing of OpenJDK JRE with GNU compiler that provides the best overall performance opportunities.

From a within-JRE release perspective, OpenJDK11 JRE had the greatest number of applications for which a change in GCC compiler in the JRE build toolchain had an effect. We observed the effect under loads from 17 (71%) of the 24 Renaissance applications running on OpenJDK11 builds. In these cases, GCC version 7 accounted for 51% of the observed differences, GCC version 8 accounted for 41%, and GCC version 6 was the best OpenJDK11 JVM build toolchain compiler for a single case, namely for the application `future-genetic`. GCC compiler version 7 was predominately associated with the Apache-spark category of applications, and GCC 8 was associated with better performance for Scala applications. A GNU GCC compiler version change did not affect JRE performance for seven Renaissance application workloads running on OpenJDK11 JRE builds. OpenJDK12 JRE had the smallest number of applications for which a change in GCC compiler in the JRE build toolchain had an effect. Only four cases showed statistically significant differences, two Apache-spark applications workloads, one Scala and the Web application `finagle-chirper`. The Scala `dotty` compiler was the only application workload with the best performance on a JRE

built with the oldest GNU GCC compiler—GCC version 7. OpenJDK JRE performance degraded under load from `dotty` as newer compilers were used in the JRE build toolchain. We observed similar proportions of significant differences for OpenJDK9, OpenJDK10, and OpenJDK13 JREs. GNU Compiler version alternatives greatly influenced OpenJDK10 and OpenJDK13 JRE interpreters under load from the Scala applications.

Ranking JRE release performance across all six OpenJDK versions has shown that three of the six OpenJDK JRE choices provide the best runtime performance. The OpenJDK9 JRE and associated interpreter provided the best runtime performance under load from 13 (54%) of the 24 Renaissance benchmark applications. The OpenJDK JRE that provided the second to best overall performance opportunities was the OpenJDK14 JRE. It gives the best overall performance for 9 (38%) of the 24 Renaissance workloads. The only other OpenJDK JRE associated with better performance was the OpenJDK JRE version 12, which provided the best performance for 2 (8%) of the 24 workloads. The OpenJDK releases 10, 11, and 13, and their associated JRE provided no performance enhancement opportunities relative to either OpenJDK JRE 9, 12, and 14 releases.

Ranking of the GNU GCC compiler version used within the build toolchains of the OpenJDK releases showed that GCC version 7 provided the best performance in 12 (50%) workload cases. GCC version 8 delivered the best performance in 6 (25%) workload cases when specified as the build compiler. GCC versions 5 and 6 accounts for the remaining 25% of best build compilers.

Of all the compiler performance rankings, version ranking sequence 7-8-6-5 accounted for 25% of all performance rankings, with 7 providing the best performance and 5 delivering the worst performance. In addition, this was followed by the version ranking list 8-7-5-6, which accounted for 14%. The top-ranked GCC compilers were version rankings 7-8, which accounted for 33% of all performance rankings. Compiler version rankings 8-7 accounted for 21% of performance rankings. The third most prominent pairing of version rankings was 7-6, which accounted for approximately 14% of performance ranks.

The interacting playoff between the GNU GCC compiler version and the OpenJDK source bundle versions showed interacting effects for 67% of loads executed on the JRE associated JVMs. The workloads with no observed interaction effects were predominately associated with those written in a functional paradigm.

Overall our results show that from all the GNU GCC compiler comparisons undertaken across all OpenJDK builds, GCC versions 7 and 8 provide the best performance opportunities. Those GCC compilers used in the build toolchains of OpenJDK9 and OpenJDK14 give the best overall combination. Our results show that changing the OpenJDK version, which includes considering downgrading from a newer release to an older release, can positively impact performance. Using a recent GNU GCC compiler release in the build cycle can significantly affect performance. For example, the OpenJDK9 release date was September 2017, and the GNU GCC version 8 release date was February 2019. Our results show that builds of OpenJDK9 undertaken before 2019 would underperform relative to a new build of OpenJDK9 with the GNU GCC version 8 compiler collection released in 2019.

Attempting to isolate and attach meaning to the observed interactions between the GNU GCC version and the OpenJDK version and the effects on OpenJDK JRE runtime performance reported within this article, consideration must be given to the components within the JRE that could have been affected through the build processes. Some components, such as the garbage collector, have undergone several revisions across OpenJDK releases. For example, the G1 garbage collector has seen four major updates, OpenJDK10 saw G1 updated for full parallelisation of its full-collection phase, and OpenJDK12 implemented abortable mixed-collections as well as speedy reclamation of committed memory. Additionally, OpenJDK version 14 saw the G1 collector become NUMA aware. Those implementations filtered through to later OpenJDK versions without deprecation or removal.

It is possible that JEP enhancements have had a collective impact and contributed to the observed statistically significant interactions between the GNU GCC version and the OpenJDK version. Although, none of the interactions in OpenJDK versions 10, 11,

or 13 appeared in the list of best performing OpenJDK JRE. In addition, concerning the parallelisation of the full G1GC cycles, we initiated all JRE instances with the default number of parallel GC threads set to 4 across all OpenJDK, from OpenJDK10 through 14. NUMA awareness does not impact the operational characteristics of the JRE and associated JVM interpreters assessed in this study, as the Raspberry Pi 4 model B system-on-chip is not a multi-socket processor. With that said, NUMA awareness is only available when running the parallel GC (`-XX:+UseParallelGC`) [81]. Also, the speedy reclamation of committed memory implemented in OpenJDK12 applies to cases when the JVM is idle, and the release of memory back to the operating system would have no impact. It is more relevant for cloud-based applications and not relevant in our system setting. In addition, an inspection of all G1GC runtime flags showed that all default settings are the same across OpenJDK releases. Other candidate JRE components that could have contributed to the observed interactions are concerned with the Java Native Interface (JNI) and the Java Class Library (JCL) that was built during the OpenJDK build cycle. There are a considerable number of those interface methods and library classes. Any impacts from a specific GNU GCC compiler would have a significant collective impact.

Additionally, the building of the Java compiler, `javac`, with the different GNU GCC compilers could impact the generated Java library classes and their respective bytecodes, as the build `javac` compiler is used to compile the JCL. Daly et al., in [19] addressed the effect the Java-to-bytecode compiler has on the emitted bytecode was addressed. They showed that different Java-to-bytecode compilers implement vastly different bytecode optimisations. Although, Gregg et al., in [54] showed that most Java-to-bytecode compilers fail to implement the simplest of optimisations to the emitted bytecode.

With that said, if the GNU GCC versions did not affect OpenJDK JRE performance, we would expect to find no significant differences in JRE performance when all JRE components are kept constant. Our analysis included that case by studying the effects of the GNU GCC compiler on specific versions of OpenJDK. For example, the four builds of the OpenJDK9 JRE all used the same source code bundle. As such, the OpenJDK version was constant.

Regarding workload factors, application workload is undoubtedly an essential factor. OpenJDK builds with specific compilers offer different performance opportunities for certain workloads. Our results have indicated that changing workload would, in many cases, necessitate a change of the compiler used within the build toolchain, even in cases where the same OpenJDK release is required. For example, OpenJDK9 or 14, built with GCC 7, provides the best performance opportunities for application workloads that target the Apache Spark framework. OpenJDK9 and OpenJDK14, made with GNU GCC version 8, give the best options for performance gains when workloads predominately involve concurrent execution. Of the two Database applications comprising the Renaissance Benchmark Suite, OpenJDK9 delivered the best overall performance coupled with GCC compiler versions 7 and 8. Of the applications written in a functional paradigm, three of the six OpenJDK builds provided the best results. Specifically, OpenJDK9, 12, and 14 built with GCC 5, 6, and 7. Scala applications performed best when running on OpenJDK14 built with GCC version 7. Our results also indicate that for the two web-based applications, `finagle-http` and `finagle-chirper`, that OpenJDK9 built with GNU compiler version 8 provided the best overall performance opportunities.

How much performance change can we expect? Our results have shown an average OpenJDK JRE performance increase of approximately 4.90% with an associated standard deviation of 4.89%. Interquartile measures show that, in approximately 50% of cases run with the optimal OpenJDK and GCC pairing, we can expect performance gains between 1.81% and 5.74%. In many instances, depending on the current OpenJDK JVM deployment, rebuilding the runtime environment can increase performance by as much as 20%.

6. Conclusions and Future Work

In this paper, we have considered the impact that GCC compiler releases used in the build toolchain of OpenJDK have on JVM interpreter performance under modern

workloads. Our findings show that, when constrained to a particular OpenJDK release, the choice of GCC compiler version can significantly impact overall application performance for many application workloads. With the extra freedom to consider other OpenJDK versions and their associated JVM interpreters, there is an optimum pairing of OpenJDK and GCC compiler for each workload. Our results show that performance can be enhanced by, on average, 5%. Many workloads show performance gains above 10%, with a smaller number exceeding 20%, peaking at approximately 25%.

Our results have clearly shown that careful consideration needs to be given to the decision regarding which OpenJDK release to choose and specifically what compiler was used in its associated build toolchain. More importantly, our results would offer options to those who are constrained to working within a specific OpenJDK release and have shown that performance opportunities still exist through careful selection of compiler collection versions used within build toolchains. Interestingly, our results in this regard suggest that a richer set of OpenJDK binary distributions should be made available to the market, not just binary builds that target specific architectures but also binaries built using different compiler collections. Furthermore, OpenJDK binaries that have a specific target workload type would be of considerable interest.

Is there scope for leveraging performance? Our results have shown that this is undoubtedly the case. In particular, we have shown that the JVM interpreter can benefit from a rebuild using an alternative toolchain compiler. In particular, older versions of OpenJDK, for example OpenJDK9, predominately exhibit better performance if built with a later GNU GCC compiler.

Our future work will consider low level microarchitectural performance metrics, such as branch misprediction, and data cache behaviour, and the impact that build toolchain compiler choice has on those performance characteristics. In addition, we plan to expand this work and include the full set of available garbage collector algorithms and assess their impact on JRE performance. Additionally, we intend to explore the effects associated with the JRE Native Libraries and Java Class Libraries and the effect of building those with alternative GCC GNU compilers. In addition, we plan to explore similar effects associated with hotspot compilation. Finally, we are also interested in exploring if hotspot-style compilation, applied to internal JVM structures, could provide additional performance opportunities.

Author Contributions: Conceptualisation, J.L.; methodology, J.L., R.M. and K.C.; validation, J.L.; formal analysis, J.L.; investigation, J.L.; resources, J.L., R.M. and K.C.; data curation, J.L.; writing—original draft preparation, J.L.; writing—review and editing, J.L., R.M. and K.C.; visualisation, J.L. and K.C.; supervision, R.M. and K.C.; project administration, R.M. and K.C. All authors have read and agreed to the published version of the manuscript.

Funding: The APC was funded by the National College of Ireland, Mayor Street, Dublin 1, Ireland.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Sample data are available at the repository: https://github.com/jonathan-lambert/JVM-GCC-Data (accessed on 1 June 2022).

Acknowledgments: Jonathan Lambert's PhD research at Maynooth University is funded by the National College of Ireland (NCI) under the NCI Educational Assistance Programme.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Gough, K.J.; Corney, D. Evaluating the Java Virtual Machine as a Target for Languages Other than Java. In *Modular Programming Languages*; Weck, W., Gutknecht, J., Eds.; Springer: Berlin/Heidelberg, Germany, 2000; pp. 278–290.
2. Cloud Foundry. Top Languages for Enterprise Application Development. Available online: https://www.cloudfoundry.org/wp-content/uploads/Developer-Language-Report_FINAL.pdf (accessed on 24 April 2021).
3. TIOBE. TIOBE Index for April 2021. Available online: https://www.tiobe.com/tiobe-index (accessed on 23 April 2021).

4. Azul Systems, Inc. Java at Speed: The Azul Zing JVM Optimized For Modern Hardware. Available online: https://www.azul.com/resources-hub/whitepapers (accessed on 27 April 2021).
5. Cass, S. IEEE Spectrum Top Programming Languages 2020. Available online: https://spectrum.ieee.org/top-programming-language-2020 (accessed on 6 June 2022).
6. Oracle. Oracle Java Archive. Available online: https://www.oracle.com/java/technologies/oracle-java-archive-downloads.html (accessed on 24 April 2021).
7. Gu, W.; Burns, N.A.; Collins, M.T.; Wong, W.Y.P. The evolution of a high-performing Java virtual machine. *IBM Syst. J.* **2000**, *39*, 135–150. [CrossRef]
8. Cramer, T.; Friedman, R.; Miller, T.; Seberger, D.; Wilson, R.; Wolczko, M. Compiling Java Just in Time. *IEEE Micro* **1997**, *17*, 36–43. [CrossRef]
9. Aycock, J. A Brief History of Just-in-Time. *ACM Comput. Surv.* **2003**, *35*, 97–113. https://doi.org/10.1145/857076.857077. [CrossRef]
10. Cook, R. *Java Schism? Embedded Leads the Way; Embedded Systems Companies Want Their Java—With or Without Sun; Embedded, Realtime, and Near—Realtime*; InfoWorld: MA, USA, 2022. Available online: https://www.infoworld.com/article/2076420/java-schism--embedded-leads-the-way.html (accessed on 6 July 2022).
11. IBM. Advantages of Java. Available online: https://www.ibm.com/docs/en/aix/7.2?topic=monitoring-advantages-java (accessed on 23 April 2021).
12. Taft, S.T. Programming the internet in Ada 95. In *International Conference on Reliable Software Technologies*; Springer: Berlin/Heidelberg, Germany, 1996; pp. 1–16.
13. Colnet, D.; Zendra, O. Optimizations of Eiffel programs: SmallEiffel, the GNU Eiffel compiler. In Proceedings of the Technology of Object-Oriented Languages and Systems, TOOLS 29, Nancy, France, 7–10 June 1999; pp. 341–350.
14. Benton, N.; Kennedy, A.; Russell, G. Compiling standard ML to Java bytecodes. In Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming, Baltimore, MD, USA, 26–29 September 1998; pp. 129–140.
15. Clausen, L.; Danvy, O. Compiling proper tail recursion and first-class continuations: Scheme on the Java Virtual Machine. *J. C Lang. Transl.* **1998**, *6*, 20–32.
16. Wakeling, D. Mobile Haskell: Compiling lazy functional languages for the Java virtual machine. In Proceedings of the 10th International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP'98), Pisa, Italy, 16–18 September 1998.
17. Gregg, D.; Power, J.F.; Waldron, J. Platform independent dynamic Java virtual machine analysis: The Java Grande Forum benchmark suite. *Concurr. Comput. Pract. Exp.* **2003**, *15*, 459–484. [CrossRef]
18. Jones, S.L.P.; Ramsey, N.; Reig, F. C–: A Portable Assembly Language that Supports Garbage Collection. In *Lecture Notes in Computer Science; Proceedings of the Principles and Practice of Declarative Programming, International Conference PPDP'99, Paris, France, 29 September–1 October 1999*; Nadathur, G., Ed.; Springer: Berlin/Heidelberg, Germany, 1999; Volume 1702, pp. 1–28. [CrossRef]
19. Daly, C.; Horgan, J.; Power, J.; Waldron, J. Platform Independent Dynamic Java Virtual Machine Analysis: The Java Grande Forum Benchmark Suite. In Proceedings of the JGI '01: 2001 Joint ACM-ISCOPE Conference on Java Grande, Palo Alto, CA, USA, 2–4 June 2001; Association for Computing Machinery: New York, NY, USA, 2001; pp. 106–115. [CrossRef]
20. O'Connor, J.M.; Tremblay, M. picoJava-I: The Java virtual machine in hardware. *IEEE MICRO* **1997**, *17*, 45–53. [CrossRef]
21. Romer, T.H.; Lee, D.; Voelker, G.M.; Wolman, A.; Wong, W.A.; Baer, J.L.; Bershad, B.N.; Levy, H.M. The Structure and Performance of Interpreters. In Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems, Cambridge, MA, USA, 1–5 October 1996; ASPLOS VII; Association for Computing Machinery: New York, NY, USA, 1996; pp. 150–159. [CrossRef]
22. Vallée-Rai, R.; Gagnon, E.; Hendren, L.; Lam, P.; Pominville, P.; Sundaresan, V. Optimizing Java Bytecode Using the Soot Framework: Is It Feasible? In *Compiler Construction*; Watt, D.A., Ed.; Springer: Berlin/Heidelberg, Germany, 2000; pp. 18–34.
23. Nikishkov, G.; Nikishkov, Y.; Savchenko, V. Comparison of C and Java performance in finite element computations. *Comput. Struct.* **2003**, *81*, 2401–2408. [CrossRef]
24. Hsieh, C.H.; Conte, M.T.; Johnson, T.L.; Gyllenhaal, J.C.; Hwu, W.M. A study of the cache and branch performance issues with running Java on current hardware platforms. In Proceedings of the IEEE COMPCON 97, Digest of Papers, San Jose, CA, USA, 23–26 February 1997; IEEE: Piscataway, NJ, USA, 1997; pp. 211–216.
25. Watanabe, K.; Li, Y. Parallelism of Java bytecode programs and a Java ILP processor architecture. *Aust. Comput. Sci. Commun.* **1999**, *21*, 75–84.
26. Hoogerbrugge, J.; Augusteijn, L. Pipelined Java Virtual Machine Interpreters. In *Compiler Construction*; Watt, D.A., Ed.; Springer: Berlin/Heidelberg, Germany, 2000; pp. 35–49.
27. Kazi, I.H.; Chen, H.H.; Stanley, B.; Lilja, D.J. Techniques for Obtaining High Performance in Java Programs. *ACM Comput. Surv.* **2000**, *32*, 213–240. [CrossRef]
28. Cao, H.; Gu, N.; Ren, K.; Li, Y. Performance research and optimisation on CPython's interpreter. In Proceedings of the 2015 Federated Conference on Computer Science and Information Systems (FedCSIS), Lodz, Poland, 13–16 September 2015; IEEE: Piscataway, NJ, USA, 2015; pp. 435–441. [CrossRef]

29. Strout, M.M.; Debray, S.; Isaacs, K.; Kreaseck, B.; Cárdenas-Rodríguez, J.; Hurwitz, B.; Volk, K.; Badger, S.; Bartels, J.; Bertolacci, I.; et al. Language-Agnostic Optimization and Parallelization for Interpreted Languages. In *Languages and Compilers for Parallel Computing*; Rauchwerger, L., Ed.; Springer: Cham, Switzerland 2019; pp. 36–46.
30. Bartels, J.; Stephens, J.; Debray, S. Representing and Reasoning about Dynamic Code. In Proceedings of the 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE), Virtual Event, Australia, 21–25 December 2020; IEEE: Piscataway, NJ, USA, 2020; pp. 312–323.
31. Oracle. Java Platform, Standard Edition JRockit to HotSpot Migration Guide. 2015. Available online: https://docs.oracle.com/javacomponents/jrockit-hotspot/migration-guide/comp-opt.htm#JRHMG117 (accessed on 6 June 2022).
32. Abdulsalam, S.; Lakomski, D.; Gu, Q.; Jin, T.; Zong, Z. Program energy efficiency: The impact of language, compiler and implementation choices. In Proceedings of the International Green Computing Conference, Dallas, TX, USA, 3–5 November 2014; IEEE: Piscataway, NJ, USA, 2014; pp. 1–6. [CrossRef]
33. Costanza, P.; Herzeel, C.; Verachtert, W. A comparison of three programming languages for a full-fledged next-generation sequencing tool. *BMC Bioinform.* **2019**, *20*, 301. [CrossRef]
34. Oancea, B.; Rosca, I.G.; Andrei, T.; Iacob, A.I. Evaluating Java performance for linear algebra numerical computations. *Procedia Comput. Sci.* **2011**, *3*, 474–478. . [CrossRef]
35. Merelo-Guervós, J.J.; Blancas-Alvarez, I.; Castillo, P.A.; Romero, G.; García-Sánchez, P.; Rivas, V.M.; García-Valdez, M.; Hernández-Águila, A.; Román, M. Ranking the Performance of Compiled and Interpreted Languages in Genetic Algorithms. In Proceedings of the International Conference on Evolutionary Computation Theory and Applications, Porto, Portugal, 11 November 2016; SCITEPRESS: Setúbal, Portugal, 2016; Volume 2, pp. 164–170.
36. Magalhaes, G.G.; Sartor, A.L.; Lorenzon, A.F.; Navaux, P.O.A.; Beck, A.C.S. How programming languages and paradigms affect performance and energy in multithreaded applications. In Proceedings of the 2016 VI Brazilian Symposium on Computing Systems Engineering (SBESC), João Pessoa, Brazil, 1–4 November 2016; pp. 71–78.
37. Lin, L.; Xu, Z.; Huan, H.; Jian, Z.; Li-Xin, L. An Empirical Comparison of Implementation Efficiency of Iterative and Recursive Algorithms of Fast Fourier Transform. In *Machine Learning and Intelligent Communications*; Guan, M., Na, Z., Eds.; Springer: Cham, Switzerland, 2021; pp. 73–81.
38. Källén, M.; Wrigstad, T. Performance of an OO Compute Kernel on the JVM: Revisiting Java as a Language for Scientific Computing Applications. In Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes, Athens Greece, 21–22 October 2019; pp. 144–156. [CrossRef]
39. Eichhorn, H.; Cano, J.L.; McLean, F.; Anderl, R. A comparative study of programming languages for next-generation astrodynamics systems. *CEAS Space J.* **2018**, *10*, 115–123. [CrossRef]
40. Pereira, R.; Couto, M.; Ribeiro, F.; Rua, R.; Cunha, J.; Fernandes, J.A.P.; Saraiva, J.A. Efficiency across Programming Languages: How Do Energy, Time, and Memory Relate? In Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering, Vancouver, BC, Canada, 23–24 October 2017; Association for Computing Machinery: New York, NY, USA, 2017; pp. 256–267. [CrossRef]
41. Pereira, R.; Couto, M.; Ribeiro, F.; Rua, R.; Cunha, J.; Fernandes, J.P.; Saraiva, J. Ranking programming languages by energy efficiency. *Sci. Comput. Program.* **2021**, *205*, 102609. [CrossRef]
42. Polito, G.; Palumbo, N.; Tesone, P.; Labsari, S.; Ducasse, S. *Interpreter Register Autolocalisation: Improving the Performance of Efficient Interpreters*; Université de Lille: Lille, France, 2022.
43. Urma, R.G. Alternative Languages for the JVM. A Look at Eight Features from Eight JVM Languages. 2014. Available online: https://www.oracle.com/technical-resources/articles/java/architect-languages.html (accessed on 23 April 2021).
44. Oracle. Oracle GraalVM Enterprise Edition 22: Truffle Language Implementation Framework. 2022. Available online: https://docs.oracle.com/en/graalvm/enterprise/22/docs/graalvm-as-a-platform/language-implementation-framework/ (accessed on 20 April 2022).
45. Prokopec, A.; Rosa, A.; Leopoldseder, D.; Duboscq, G.; Tuma, P.; Studener, M.; Bulej, L.; Zheng, Y.; Villazon, A.; Simon, D.; et al. On evaluating the renaissance benchmarking suite: Variety, performance, and complexity. *arXiv* **2019**, arXiv:1903.10267.
46. Prokopec, A.; Rosa, A.; Leopoldseder, D.; Duboscq, G.; Tuma, P.; Studener, M.; Bulej, L.; Zheng, Y.; Villazon, A.; Simon, D.; et al. Renaissance: A modern benchmark suite for parallel applications on the JVM. In Proceedings of the Companion of the 2019 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity, Athens, Greece, 20–25 October 2019; pp. 11–12.
47. Proebsting, T.A. Optimizing an ANSI C interpreter with Superoperators. In Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, CA, USA, 23–25 January 1995; pp. 322–332.
48. Gagnon, E.; Hendren, L. Effective inline-threaded interpretation of Java bytecode using preparation sequences. In Proceedings of the International Conference on Compiler Construction, Warsaw, Poland, 7–11 April 2003; Springer: Berlin/Heidelberg, Germany, 2003; pp. 170–184.
49. Maierhofer, M.; Ertl, M.A. Local Stack Allocation. In Proceedings of the CC '98: Proceedings of the 7th International Conference on Compiler Construction, Lisbon, Portugal, 28 March–4 April 1998; Springer: London, UK, 1998; pp. 189–203.
50. Park, J.; Park, J.; Song, W.; Yoon, S.; Burgstaller, B.; Scholz, B. Treegraph-based instruction scheduling for stack-based virtual machines. *Electron. Notes Theor. Comput. Sci.* **2011**, *279*, 33–45. [CrossRef]

51. Lengauer, P.; Bitto, V.; Mössenböck, H.; Weninger, M. A Comprehensive Java Benchmark Study on Memory and Garbage Collection Behavior of DaCapo, DaCapo Scala, and SPECjvm2008. In Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering, L'Aquila, Italy, 22–26 April 2017; Association for Computing Machinery: New York, NY, USA, 2017; pp. 3–14. [CrossRef]
52. Casey, K.; Ertl, A.; Gregg, D. Optimizing Indirect Branch Prediction Accuracy in Virtual Machine Interpreters. *ACM Trans. Program. Lang. Syst. (TOPLAS)* **2007**, *29*, 37-es. [CrossRef]
53. Davis, B.; Waldron, J. A Survey of Optimisations for the Java Virtual Machine. In Proceedings of the 2nd International Conference on Principles and Practice of Programming in Java, Kilkenny, Ireland, 16–18 June 2003; Computer Science Press, Inc.: New York, NY, USA, 2003; pp. 181–183.
54. Gregg, D.; Ertl, M.A.; Krall, A. A fast java interpreter. In Proceedings of the Workshop on Java Optimisaton Strategies for Embedded Systems (JOSES), Genoa, Italy, 1 April 2001; Citeseer: Genoa, Italy, 2001.
55. Ertl, M.A.; Gregg, D. The Behavior of Efficient Virtual Machine Interpreters on Modern Architectures. In Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing, Manchester, UK, 28–31 August 2001; Springer: Berlin/Heidelberg, Germany, 2001; pp. 403–412.
56. Ertl, M.A.; Gregg, D. The structure and performance of efficient interpreters. *J. Instr.-Level Parallelism* **2003**, *5*, 1–25.
57. Berndl, M.; Vitale, B.; Zaleski, M.; Brown, A. Context threading: A flexible and efficient dispatch technique for virtual machine interpreters. In Proceedings of the International Symposium on Code Generation and Optimization, San Jose, CA, USA, 22–23 March 2005; IEEE: Piscataway, NJ, USA, 2005; pp. 15–26. [CrossRef]
58. McGhan, H.; O'Connor, M. PicoJava: A Direct Execution Engine For Java Bytecode. *Computer* **1998**, *31*, 22–30. [CrossRef]
59. Li, Y.; Li, S.; Wang, X.; Chu, W. Javir-exploiting instruction level parallelism for java machine by using virtual register. In Proceedings of the The Second European IASTED International Conference on Parallel and Distributed Systems, Vienna, Austria, 1–3 July 1998, pp. 80–86; Citeseer: Princeton, NJ, USA, 1998; pp. 1–3.
60. Shi, Y.; Casey, K.; Ertl, M.A.; Gregg, D. Virtual Machine Showdown: Stack versus Registers. *ACM Trans. Archit. Code Optim.* **2008**, *4*, 1–36. [CrossRef]
61. Evans, B.J. *Java: The Legend*, 1st ed.; O'Reilly Media, Inc.: Sebastopol, CA, USA, 2015.
62. Reinhold, M. JDK Enhancement Proposal Index. Available online: https://openjdk.java.net/jeps/0 (accessed on 26 May 2022).
63. Johansson, S. JDK Enhancement Proposal 307: Parallel Full GC for G1. Available online: https://openjdk.java.net/jeps/307 (accessed on 26 May 2022).
64. Helin, E. JDK Enhancement Proposal 344: Abortable Mixed Collections for G1. Available online: https://openjdk.java.net/jeps/344 (accessed on 26 May 2022).
65. Rodrigo, B.; Schatzl, T.; Synytsky, R. JDK Enhancement Proposal 346: Promptly Return Unused Committed Memory from G1. Available online: https://openjdk.java.net/jeps/346 (accessed on 26 May 2022).
66. Kim, S. JDK Enhancement Proposal 345: NUMA-Aware Memory Allocation for G1. Available online: https://openjdk.java.net/jeps/345 (accessed on 26 May 2022).
67. Lam, I. JDK Enhancement Proposal 310: Application Class-Data Sharing. Available online: https://openjdk.java.net/jeps/310 (accessed on 26 May 2022).
68. Jiangli, Z.; Calvin, C.; Ioi, L. JDK Enhancement Proposal 341: Default CDS Archives. Available online: https://openjdk.java.net/jeps/341 (accessed on 26 May 2022).
69. Jiangli, Z.; Calvin, C.; Ioi, L. JDK Enhancement Proposal 346: Dynamic CDS Archives. Available online: https://openjdk.java.net/jeps/350 (accessed on 26 May 2022).
70. OpenJDK. Building the JDK. Available online: https://openjdk.java.net/groups/build/doc/building.html (accessed on 26 May 2022).
71. Kumar, R.; Singh, P.K. An Approach for Compiler Optimization to Exploit Instruction Level Parallelism. In *Advanced Computing, Networking and Informatics—Volume 2*; Kumar Kundu, M., Mohapatra, D.P., Konar, A., Chakraborty, A., Eds.; Springer International Publishing: Cham, Switzerland, 2014; pp. 509–516.
72. Marcon, C.; Smirnova, O.; Muralidharan, S. Impact of different compilers and build types on Geant4 simulation execution time. *EPJ Web Conf.* **2020**, *245*, 05037. [CrossRef]
73. Agostinelli, S.; Allison, J.; Amako, K.; Apostolakis, J.; Araujo, H.; Arce, P.; Asai, M.; Axen, D.; Banerjee, S.; Barrand, G.; et al. Geant4—A simulation toolkit. *Nucl. Instruments Methods Phys. Res. Sect. A Accel. Spectrometers Detect. Assoc. Equip.* **2003**, *506*, 250–303. [CrossRef]
74. Horgan, J.; Power, J.F.; Waldron, J.T. Measurement and Analysis of Runtime Profiling Data for Java Programs. In Proceedings of the First IEEE International Workshop on Source Code Analysis and Manipulation, Florence, Italy, 10 November 2001; pp. 122–130.
75. Clausen, L.R. A Java bytecode optimizer using side-effect analysis. *Concurr. Pract. Exp.* **1997**, *9*, 1031–1045. [CrossRef]
76. Foley, S. Japt (Java Optimizer) Bytecode Optimizer for Java. 2016. Available online: https://seancfoley.github.io/Japt-Bytecode-Optimizer-for-Java/index.html (accessed on 6 June 2022).
77. Oi, H. A Comparative Study of JVM Implementations with SPECjvm2008. 2010 Second International Conference on Computer Engineering and Applications, Bali, Indonesia, 19–21 March 2010; Volume 1, pp. 351–357. [CrossRef]
78. Prokopec, A.; Rosà, A.; Leopoldseder, D.; Duboscq, G.; Tůma, P.; Studener, M.; Bulej, L.; Zheng, Y.; Villazón, A.; Simon, D.; et al. Renaissance: Benchmarking Suite for Parallel Applications on the JVM. In Proceedings of the 40th ACM SIGPLAN Conference on

Programming Language Design and Implementation, Phoenix, AZ, USA, 22–26 June 2019; Association for Computing Machinery: New York, NY, USA, 2019; pp. 31–47. [CrossRef]
79. RaspberryPi.org. Raspberry Pi Foundation BCM2711. Available online: https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2711/README.md (accessed on 24 April 2020).
80. RaspberryPi.com. Raspberry Pi Foundation Pi 4 Specifications. Available online: https://www.raspberrypi.org/products/raspberry-pi-4-model-b/specifications/ (accessed on 24 April 2020).
81. Oracle. The Java Command. Available online: https://docs.oracle.com/en/java/javase/13/docs/specs/man/java.html#advanced-garbage-collection-options-for-java (accessed on 26 May 2020).