International Journal of Environmental Sustainability and Green Technologies
Volume 14 • Issue 1

Comparative Performance and Energy Efficiency Analysis of JVM Variants and GraalVM in Java Applications

Thalita Grange Vergilio, Leeds Beckett University, UK*
Long Do Ha, Leeds Beckett University, UK
Ah-Lian G. Kor, Leeds Beckett University, UK

ABSTRACT

Java has dominated the ICT market for almost thirty years with various applications in nearly every sector all over the world. One of Java's main drawbacks comes from its heavyweight core - the Java Virtual Machine (JVM). Therefore, several JVM distributions have been developed to address this issue. GraalVM is the most promising amongst the recent distributions, providing better performance, low power consumption, and reduced carbon footprint emissions. In this research, a comparative analysis based on performance and energy efficiency metrics was conducted to assess this JVM distribution in light of three other classic JVM distributions: Amazon Corretto, Adopt OpenJDK, and Zulu. Findings showed that, although there was no significant difference between the test candidates, GraalVM seemed to be the leading JVM distribution. It is recommended that programmers and technology businesses consider adopting GraalVM in their future Java applications because of its energy efficiency.

KEYWORDS:

Java, JVM, OpenJDK, Amazon Corretto, Zulu, GraalVM, Oracle, energy consumption, carbon footprint, performance

1. COMPARATIVE PERFORMANCE AND ENERGY EFFICIENCY ANALYSIS OF JVM VARIANTS AND GRAALVM IN JAVA APPLICATIONS

1.1. Background Context

Since its introduction approximately thirty years ago by James Gosling and his colleagues at Sun Microsystems (Rauf, 2018), Java has grown to become one of the most extensively used programming languages. It has acquired popularity and become a market leader in almost every programming environment, particularly enterprise applications, server-side Web development, and mobile phone

DOI: 10.4018/IJESGT.331401 *Corresponding Author

programming (Evans, 2015). According to Floyer (2020), a survey carried out by Wikibon in 2020 showed that 93% of enterprises chose Java as their leading application development platform, with C++ and Python ranking far behind with, respectively, 51% and 29% of respondents. However, Java is still a heavyweight programming language, mainly due to applications processed on the Java Virtual Machine (JVM). Thus, every Java program needs considerable memory and processing resources (GeeksForGeeks, 2019), which creates a financial burden for small and medium enterprises (SMEs) due to its high energy consumption. Understanding this problem, various JVM providers have released improved JVM distributions with reduced runtime memory CPU usage, such as Adopt OpenJDK HotSpot, Amazon Corretto, and Redhat Mandrel. These variants have, to different extent, addressed the original JVM's issues. The most noticeable JVM distribution, however, is Oracle's GraalVM, which provides several significant improvements and the ability to compile Java source code into a standalone binary executable program called native image (Graalvm.org, n. d.). Although several studies have been conducted to compare the performance of the GraalVM with other improved JVM distributions, limited work focuses on the sustainability aspect (Ournani et al., 2021). This paper considers both aspects by carrying out practical experiments with different scenarios to explore whether GraalVM is a worthy upgrade for SMEs in terms of performance and energy efficiency.

1.2. Aim and Objectives

With the support of eight benchmark tests, this study investigates the performance and energy consumption of three selected JVM distributions (Amazon Corretto, Adopt OpenJDK, Zulu), and GraalVM. The research objectives are listed below:

1. Based on the appropriate literature review, design scenarios and test suites to measure performance and energy consumption for chosen JVM distribution candidates.
2. To identify optimal performance and energy consumption measuring methods and tools to apply.
3. Based on outcomes from (1) and (2), carry out experiments and conduct a comparative analysis of these JVM distributions and GraalVM based on performance and energy efficiency criteria.
4. To apply data analysis approaches to the experimental data to answer three research questions:

1.2.1 Level 1: Descriptive Statistical Analysis

a. Utilizing the same specified scenarios and test suites, how do the selected JVM distributions compare with GraalVM in terms of performance and energy consumption?

1.2.2 Level 2: Inferential Statistical Analysis

b. Are there any remarkable differences in performance and energy consumption between GraalVM and the selected JVM distributions?
c. How can GraalVM support SMEs in monthly energy costs reduction?
5. To present conclusions and possible recommendations based on analytical findings and suggest future research work.

1.2.2.1 Rationale

Energy efficiency is becoming a serious challenge due to the prevalence of Information Technology systems and their impact on global energy usage (Ergasheva et al., 2020). If we do not address this problem satisfactorily, it might lead to severe consequences for current and future generations, such as resource depletion and global warming. Software developers frequently state that sustainable solutions should come from more energy-efficient hardware components or efficient algorithms (Ournani et

al., 2021). However, given the complexity of today's software environments, the composition of software layers makes this goal particularly difficult to attain. Therefore, modifying the abstraction layer for a programming language is extremely important, especially for Java, which is one of the current top five programming languages (TIOBE Software, 2023). When comparing Java's JVM distributions, decisive criteria (i.e., performance, memory usage, and CPU usage) would help ascertain which distribution is better for Java applications development. There is limited research on energy efficiency and its trade-off with performance for different JVM distributions. Therefore, this research is conducted to address this pressing issue.

1.2.2.2 Scope

Three distribution candidates were selected, and their performance and energy consumption measured by the experiments in this paper. Apart from GraalVM, the criteria to choose the other JVM distributions were current market usage and non-commercial usage. The latter makes this research particularly relevant to SMEs. Therefore, based on the information reported by Vermeer in 2020, Adopt OpenJDK, Amazon Corretto OpenJDK, and Azul Zulu were identified as ideal candidates with 24%, 4%, and 4% of usage respondents, respectively. This research aims to extend the scope so that it encompasses a comparative analysis of the carbon footprint for GraalVM and selected JVM distributions followed by a discussion of their environmental impact in eight different scenarios based on benchmark tests. This paper will not consider variations in the JVM configurations of any distributions. Therefore, all configurations were set to default before conducting experiments to ensure objectivity and consistency. All experiments were conducted on a Dell Vostro 5502 personal computer powered by an 11th Gen Intel Core i7 processor (8 CPU) with 16GB RAM, 512GB SSD, and running the Windows 11 Pro 64-bit operating system, as specified in Table 1. Performance was measured based on the CPU and memory usage of the computer during the experiments.

1.2.2.3 Contribution of Research

In this study, the three most popular JVM distributions and GraalVM are assessed and compared with each other by using software-defined power toolkit benchmarks and carbon emission estimation metrics in different test scenarios. The study entailed Descriptive and Inferential Statistical Analyses on the experimental data to derive conclusions which might be utilized to inform JVM selection. The positive findings could lead to further future research, enabling software engineers and organizations, especially SMEs, to make optimal evidence informed decisions.

1.2.2.4 Organization of Report

There are five sections in this paper. The first section describes the background context and clarifies the research's main research questions and objectives. The second section provides an overview of related work that supports this research. The methodologies used to design and conduct the experiments and analyse the results are presented in Section 3. In Section 4, the experimental results are collected, analysed, and discussed. Finally, Section 5 utilizes these results to conclude the paper and provide evidence-based recommendations.

2. LITERATURE REVIEW

2.1. ICT and Environment Impact

It is an accepted fact that rapid advancements in ICT have been matched by an increase in the number of digitized devices and services over the last two decades. These have been beneficial in different ways. With the support of AI and big data, for instance, technology devices have been utilized for patient screening, outbreak monitoring, case tracking and tracing, disease prediction, and infection risk assessment. Another beneficial example is 3D printing which enables quicker and more affordable low-volume production as well as quick, iterative product prototyping (United Nations,

2021). However, according to Mahdavi and Sojoodi (2021), this sector is a double-edged sword for the environment. It can have both positive and negative impacts on environmental sustainability. On the one hand, technology products allow humanity to monitor and control the environment, such as measuring GHG emissions to inform drivers to modify their driving behaviour to be more environment friendly. Moreover, by modifying the pattern of household energy usage, ICT could reduce power consumption and improve energy efficiency (Bastida et al., 2019). Other research seems to confirm that ICT mitigates energy loss through optimal solutions, such as intelligent networks and automated lighting, heating, and cooling management systems (Houghton, 2009). On the other hand, ICT applications without environmental awareness might lead to severe problems to our fragile ecosystem. To be more specific, Avom et al. (2020) found that mobile phone and Internet penetration have a significantly negative effect on the environment by increasing the volume of CO2 emissions. In a study conducted by Arushanyan et al. (2014), ICT products and services were found to play a vital role in global energy consumption and global warming. An enhanced ICT use efficiency has the potential to lower energy prices, thus encouraging greater efficient energy consumption.

2.2. Java and the JVM

Back in the 1990s, the Internet blossomed in a way that we have never seen before. Due to this remarkable growth, several new general-purpose programming languages were developed and applied to various sectors in that period, namely Haskell (1990), Python (1991), Visual Basic (1991), Ruby (1993), Lua (1993), R (1993), Java (1995), Javascript (1995), and PHP (1995) (Long, 2017). Among these programming languages, Java has dominated the popularity chart for almost 20 years, ranking first from 2002 to 2017, and staying amongst the top three programming languages since 1987, as shown in Figure 1.

In the past 20 years, platform independence is one of the most distinctive features that has made Java stand out compared with other programming languages. The applications of Java can be found in various places, including computers, smartphones, and even parking metres (Lestal, 2021). This was also the most important principle when James Gosling and a group of engineers at the United States's Sun Microsystems decided to create it in 1991 (Ikedilo et al., 2021). It was later promoted as "Write once, run anywhere" (WORA) to draw the attention of the technology community at that time. Apart from this principle, as Gosling et al. clarified in 2018, Java is a general-purpose, concurrent, class-based, object-oriented programming language. It is designed to be basic enough that even inexperienced coders can quickly pick it up. The Java programming language is similar to

Figure 1. Programming Language Chart Since 1987 (TIOBE Software, 2023)

Very Long Term History

To see the bigger picture, please find below the positions of the top 10 programming languages of many years back. Please note that these are *average* positions for a period of 12 months.

Programming Language	2023	2018	2013	2008	2003	1998	1993	1988
Python	1	4	8	7	13	27	17	-
C	2	2	1	2	2	1	1	1
Java	3	1	2	1	1	19	-	-
C++	4	3	4	4	3	2	2	6
C#	5	5	5	8	10	-	-	-
Visual Basic	6	15	-	-	-	-	-	-
JavaScript	7	7	10	9	8	22	-	-
Assembly language	8	12	-	-	-	-	-	-
SQL	9	251	-	-	7	-	-	-

C and C++, although it is organised differently, with some C and C++ features eliminated, and a few concepts from other languages incorporated. A typical Java application is illustrated in Figure 2.

Normally, a Java application undergoes five distinct phases: edit, compile, load, verify, and execute (Deitel & Deitel, 2019).

The first and the second phases are done by programmers (Figure 3 and Figure 4), while the remaining three phases are handled by the core of the programming language: the Java Virtual Machine (JVM). Basically, when executing a compiled java program with the command "java className," the JVM loads the program into memory for execution by a component named ClassLoader (Figure 5).

Figure 2. A Java application

```
1   package com.octocat.mew.sugoi.api;
2
3   import com.octocat.mew.sugoi.api.definition.IIndexApplication;
4   import org.springframework.web.bind.annotation.RequestMapping;
5   import org.springframework.web.bind.annotation.RequestMethod;
6   import org.springframework.web.bind.annotation.RestController;
7
8   @RestController
9   public class IndexApplication implements IIndexApplication{
10
11      @Override
12      public String index() {
13          return "Greeting from Octocat!";
14      }
15  }
```

Figure 3. Java program execution: Phase 1 (Deitel & Deitel, 2019)

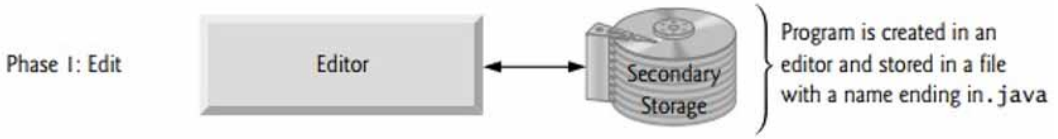


Figure 4. Java program execution:Phase 2 (Deitel & Deitel, 2019)

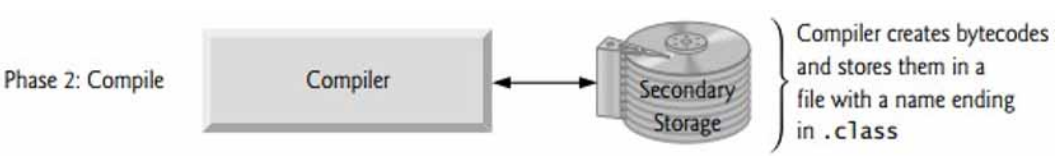
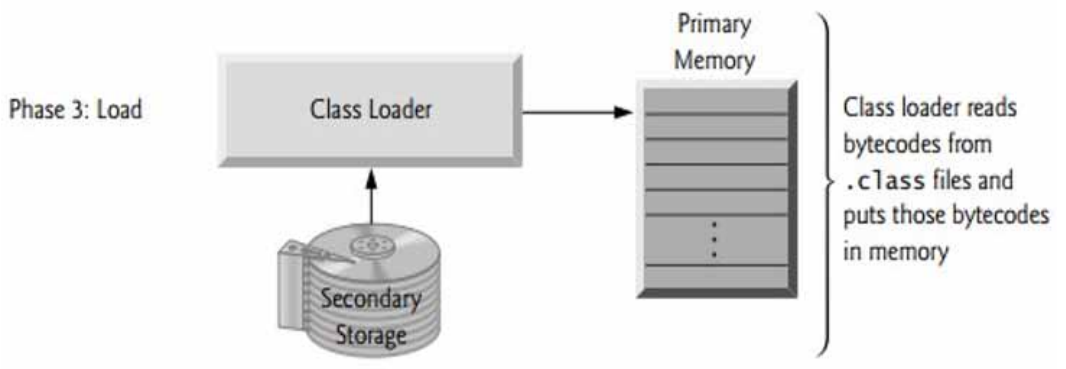


Figure 5. Java program execution: Phase 3 (Deitel & Deitel, 2019)



After the program has been loaded, the Bytecode Verifier component checks whether the program is valid or not based on Java's security restrictions (Figure 6). Subsequently, in the final phase, the JVM executes the bytecodes to run the program's stated instructions (Figure 7). Deitel and Deitel (2019) also stated that the JVM was originally just a Java bytecode interpreter because the JVM interpreted and executed one bytecode at a time and, consequently, most programs would run slowly. Nowadays, it uses a combination of interpretation and just-in-time (JIT) compilation to execute bytecode faster (Deitel & Deitel, 2019).

As illustrated in Figures 3 and 4, the JVM is required for a Java program to be executable. Therefore, a compiled language program such as Java normally utilizes more memory and processing power than its interpreted alternatives. Many JVM improvements and distributions have been released to address this issue and improve Java applications' performance while utilising as minimal computing resources as possible (Ournani et al., 2021). When writing this paper, Java 17 has been published as the latest Long Term Support (LTS) version. Oracle urges developers to upgrade to the most recent version or an LTS release as soon as possible since the legacy versions incur lower performance and will phase out of support in the near future (Ournani et al., 2021).

2.3. GraalVM

As previously mentioned, JVM improvements and distributions have been published continuously to enhance performance and/or patch existing problems. Amongst all distributions, GraalVM is the most distinctive, since it is a high-performance VM built on the JVM and provides a number of optimisation facilities (Kumar, 2021a). It was created as part of OpenJDK's Graal Project in 2012, which aimed to create a next-generation high-performance polyglot virtual machine with an extensive

Figure 6. Java program execution: Phase 4 (Deitel & Deitel, 2019)

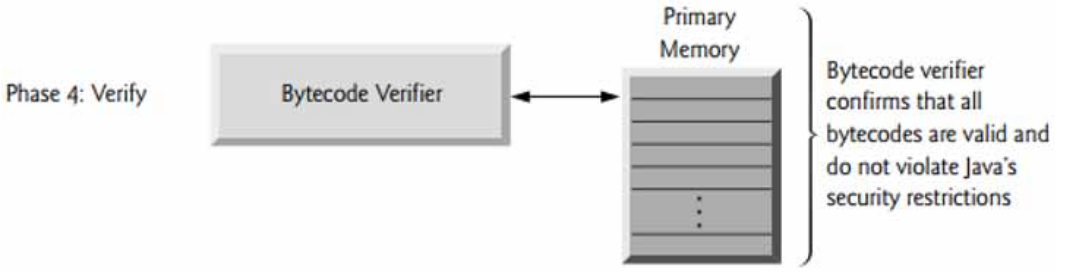
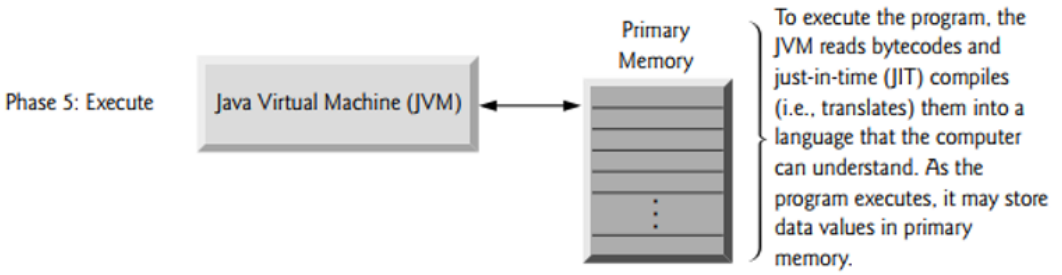


Figure 7. Java program execution: Phase 5 (Deitel & Deitel, 2019)



ecosystem that included the Graal Compiler, GraalVM Native Image Mechanism, Truffle Language Implementation Framework, and Sulong Language Implementation Framework (LLVM) (Sipek et al., 2020). Figure 8 below is an illustration of the Graal stack.

In his blog post, Kumar explained every component in this stack in detail. The JVM (Hotspot) is simply a JVM in the JDK (Kumar, 2021b). The JVM Compiler Interface (JVMCI) was introduced with Java 9, making it possible to write compilers as plugins that the JVM could use for dynamic compilation. It included an API and a protocol for creating compilers with customized implementations and optimisations. Noticeably, the Graal Compiler and Substrate VM offer memory management, garbage collection, thread scheduling, and other necessary VM functionality. Additionally, the GraalVM can build native images for a specific target OS/architecture utilising the Ahead-Of-Time application compilation mechanism. This feature is responsible for a reduced footprint, quick startups, and embeddable runtimes, therefore playing a pivotal role in cloud-native and serverless workloads. Based on the target OS, a native image is compiled which includes the application source code, dependencies, the JDK, and Substrate VM. This means the target OS does not need to install a specific JVM to get the application up and running as the image is executable for the target OS without any further requirements. The whole process is presented in Figure 9.

Lastly, Truffle and Sulong are two components on top of the stack which ensure the polyglot capability, as well as interoperability in a cloud-native environment by enabling multiple non-JVM

Figure 8. Graal Stack (Kumar, 2021b)

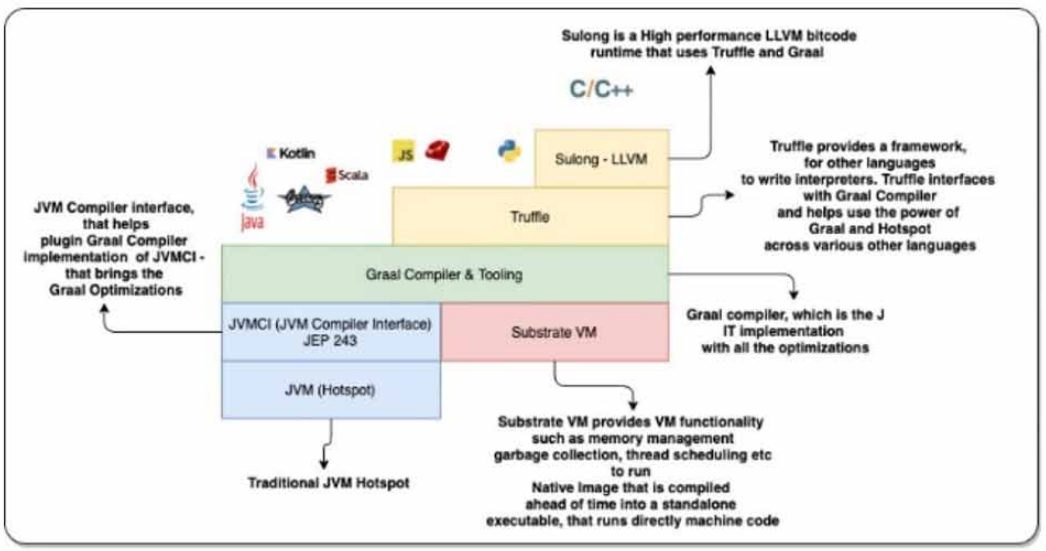
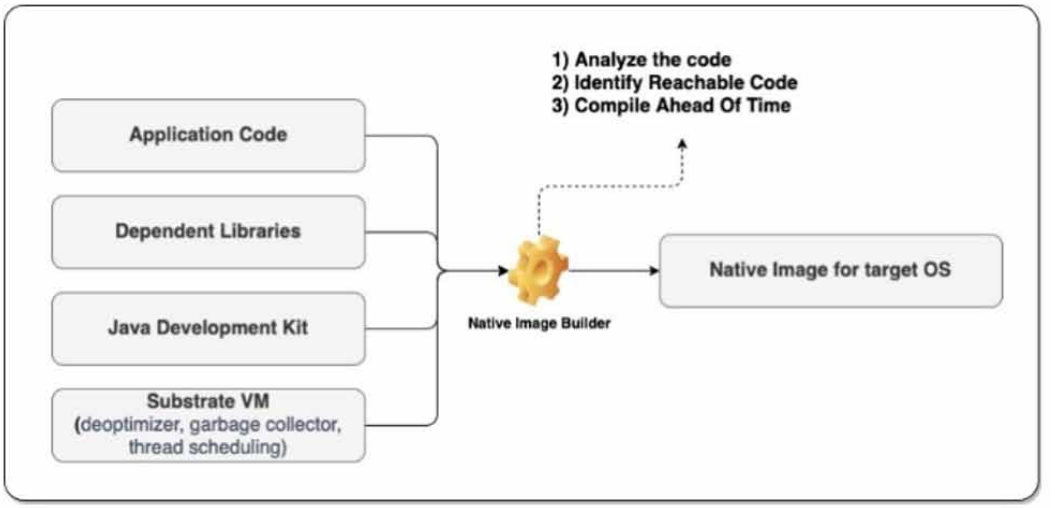


Figure 9. Graal Ahead-Of-Time Compilation (Kumar, 2021b)



programming languages compilation such as C, C++, Rust, Swift, Fortran, Javascript, Python, and R (Kumar, 2021b). Given these advancements, GraalVM is projected to be a pioneering multilingual virtual machine that is already supporting companies to save computer resources and costs (Morales, 2019). In fact, a study conducted by Ournani with eleven JVM distributions showed that, in most cases, GraalVM was the most energy efficient distribution (Ournani et al., 2021). Their results are in line with and corroborate the research presented in this paper.

2.4. Energy Measurement and Optimisation Tools

Energy management at a higher level requires monitoring or estimation of hardware and software energy as well as resource consumption (Noureddine et al., 2013). Therefore, energy measurement and optimisation tools are vital for optimal energy management. Nowadays, such tools are diverse and could provide insightful energy consumption-related information for various types of applications such as software and media players (Kor et al., 2015).

PowerAPI, developed by the Inria ADAM project team, is one of the most popular energy monitoring tools. It could measure energy consumption at the granularity of a system process, in real-time, and without any external device. To assess the power usage of a software system, PowerAPI uses energy analytical models to estimate based on the consumption of a computer's hardware components such as CPU, memory, and disk (Bourdon et al., 2013).

Another widely used alternative is Microsoft Joulemeter. This software could measure and record the energy consumption of hardware resources such as CPU, base system, and disk in Watt (W) (Sehgal et al., 2022). A study conducted by Kansal et al. (2009) showed that proper usage of Joulemeter might result in about an 8-12% reduction of power provisioning costs when operating in a Virtual Machine environment.

3. METHODOLOGY

3.1. Macro Methodology

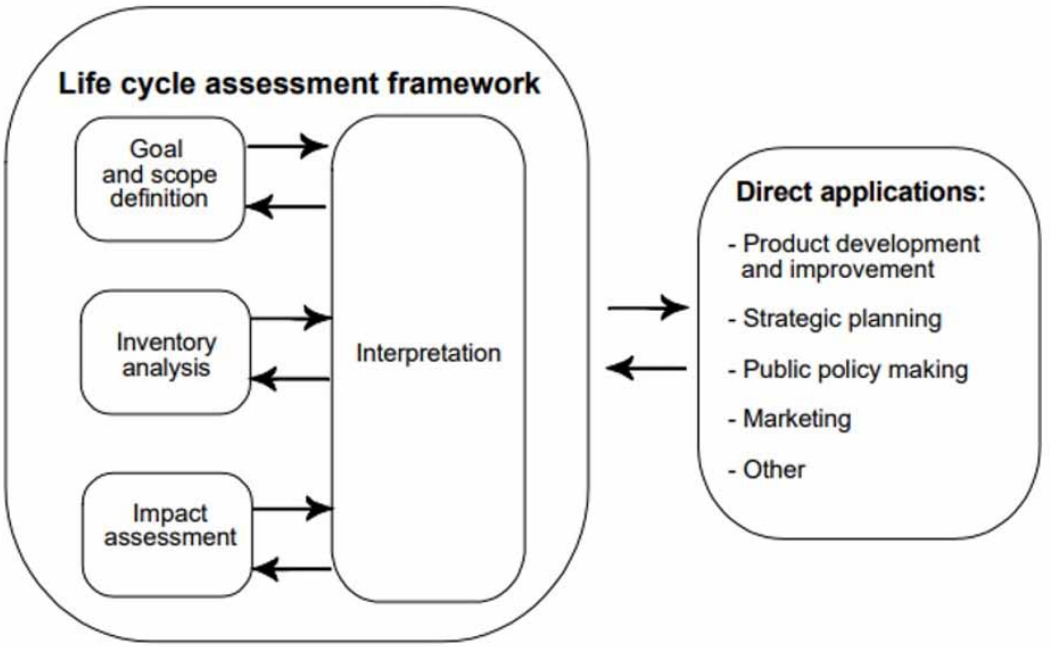
There are three widely used standards, originating from ISO 14040 and 14044, to assess the environmental impact of ICT products/services lifecycle (Stephens & Didden, 2013):

- ETSI – TS 103 199 "Life Cycle Assessment (LCA) of ICT equipment, networks, and services: General methodology and common requirements."
- ITU-T L.1410 "Methodology for environmental impact assessment of information."
- IEC TR 62725 "Quantification methodology of greenhouse gas emissions for electrical and electronic products and systems."

This study will follow the ITU-T L.1410 methodology issued by the International Telecommunication Union (ITU) to conduct an analysis of the energy consumption and carbon emissions of GraalVM and other JVM distributions. According to ITU's recommendation, the environmental effects of ICT goods, networks, and services are assessed by a systematic analytical method called Life Cycle Assessment (LCA) (ITU-T, 2012). This method provides a cradle-to-grave scope that considers all stages of a product such as raw material acquisition, manufacturing, use/reuse/maintenance, and recycle/waste management. The four phases of this methodology are illustrated in Figure 10.

- Goal and scope definition: research purpose, boundaries of the systems, requirements of data quality, and functional units are clearly explained in this phase to provide a solid background for the study.
- Life cycle inventory (LCI): in the second phase, data collection, calculation, and allocation procedures are carried out using information obtained in the first phase.
- Life cycle impact assessment (LCIA): this phase involves evaluating the data that has been obtained. The data gathered throughout the studies will be examined to determine the environmental impact of power consumption and carbon emissions.
- Life cycle interpretation: using results from LCI and LCIA, this phase highlights significant problems, completeness, sensitivity, consistency assessment, and provides conclusions and recommendations for the system.

Figure 10. Fixed Phases of an LCA



- Reporting (additional phase): after interpreting the results, reporting is crucial to convey the findings to stakeholders conveniently and support users to make evidence-informed decisions.

3.2. Micro Methodology

To measure the performance and energy consumption of GraalVM and other JVM distributions, a dedicated test environment was designed and installed on a laptop with the specifications shown in Table 1.

Based on these hardware specifications, Joulemeter 1.2 was the ideal candidate to quantitatively measure and gather program execution duration and energy usage data. This tool is Windows-friendly, easy to use, and provides detailed data on the CPU, software applications, and the monitor (Kor et al., 2015).

Windows Subsystem for Linux (WSL) 2 was used as a standalone testing environment for all experiments conducted in this paper to ensure flexibility when switching between several experiments and the objectivity of the results. This impressive feature has been introduced since Windows 10 version 1607, which provides identical capabilities to utilising Linux applications and libraries within the Bash shell (Singh & Gupta, 2019).

3.3. Environment Setup

As mentioned above, all experiments in this paper were conducted in a standalone environment, and the results were collected via Joulemeter as another process in Windows. Figure 11 shows the overall experiment design.

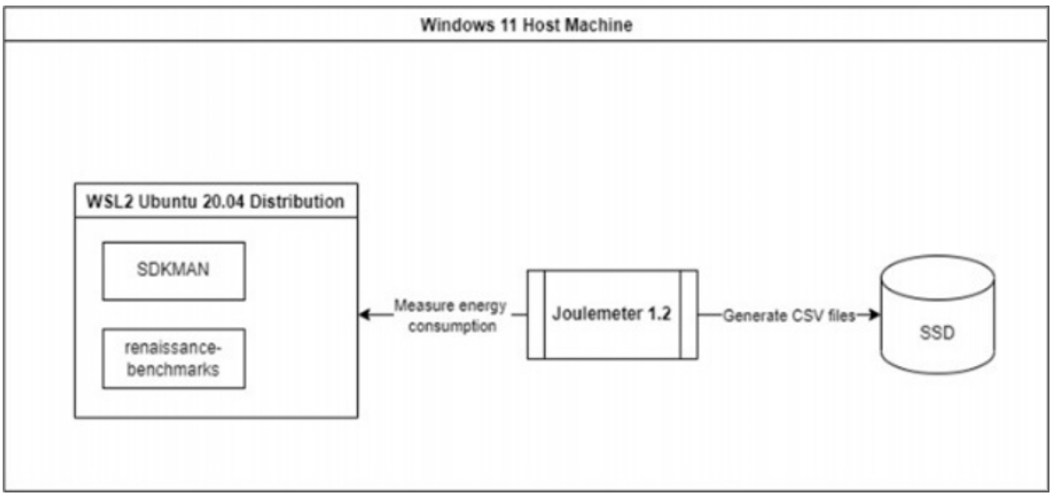
Inside the Windows 11 host machine, there are three components: WSL2 with Ubuntu 20.04 Distribution, Joulemeter 1.2, and SSD storage to store CSV files:

- WSL2 with Ubuntu 20.04 Distribution hosts all necessary applications and libraries for executing the benchmark tests. The two most important parts of this component are Software Development Kit Manager (SDKMAN) and renaissance-benchmarks. SDKMAN is a program for managing the installation and selection of Software Development Kits, which includes multiple versions and releases of Java and tools for creating, debugging, monitoring, documenting, and deploying applications. It is compatible with Windows (not natively supported, it needs to be installed via WSL, Cygwin, or MinGW), Linux, and macOS (Gilliard, 2020). Before carrying out the experiments, several JDK candidates were installed using SDKMAN. OpenJDK 11.0.2, Amazon Corretto 11.0.14.1, and Zulu 11.0.14 (three examples of classic JVM distributions)

Table 1. Laptop specifications

Type	Specification
Model	Dell Vostro 5502
Operating System	Windows 11 Pro 64-bit(10.0, Build 22000)
Processor	11th Gen Intel Core i7-1165G7,2.8GHz (8 CPUs), 1.7GHz
GPU	NVIDIA GeForce MX330,GDDRAM5 2GB
Battery	40Wh
Screen	15.6-inch, resolution 1920 x 1080p,Full HD. Luminance 300 nits
Storage	SSD 512GB
RAM	16GB
Maximum Power Supply	65 Watt

Figure 11. Experiment design



and GraalVM 21.3.1 (build 11.0.14) (as a GraalVM alternative) were selected for this study. To assess the performance and power consumption of these JVM distributions, the Renaissance benchmark suite was used as a benchmark suite. It comprises 21 benchmarks created in popular Java and Scala frameworks that illustrate modern concurrency and parallelism workloads. The Renaissance benchmark suite was invented to provide a more efficient tool to identify new compiler optimisations compared with existing test benchmarks such as DaCapo (used by Ournani et al., 2021), ScalaBench, and SPECjvm2008 (Prokopec et al., 2019). This paper has considered all Apache Spark test cases in the benchmark to measure performance and power consumption.

- Before measurement, it was necessary to configure Joulemeter in advance. In fact, the configurations depend on the computer model and hardware. Based on the information from the previous section, the laptop profile was chosen as the profile configuration for Joulemeter. The configuration information is shown in Figure 12. On the power usage tab, the program name 'vmmem' is specified as the target process to measure and collect the power consumption data. Figure 13 depicts this step in detail.

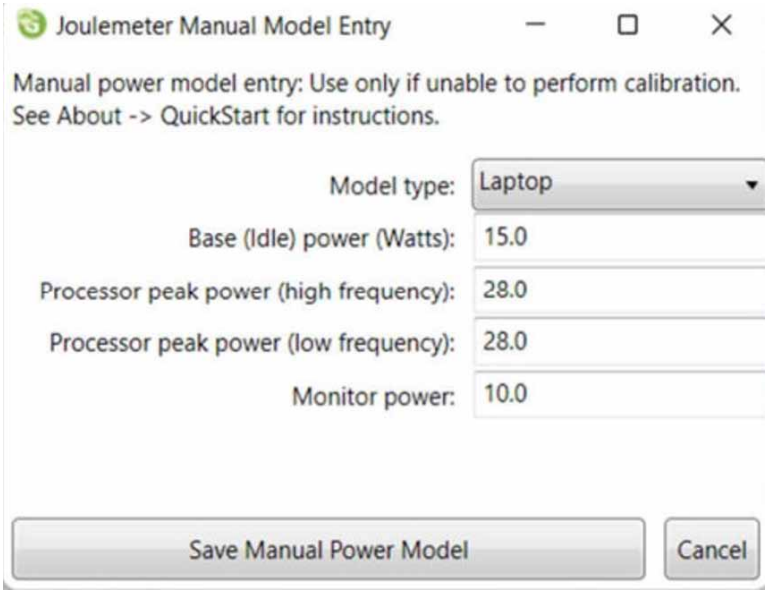
3.4. Conducting the Experiments

Eight benchmark tests were used to conduct experiments for five test candidates: OpenJDK 11.0.12, Amazon Corretto 11.0.14.1, Zulu 11.0.14, GraalVM 21.3.1 (build 11.0.14), and GraalVM 21.3.1 (build 11.0.14) with the native image. Each test was repeated five times to ensure fairness and consistency. The specification of each benchmark test is summarised in Table 2. The results obtained from each test iteration were stored in a CSV file. Subsequently, information from the CSV files was grouped by individual benchmark test to be analysed and discussed.

4. FINDINGS AND DISCUSSION

In this study, a total of 200 experiments were carried out using 8 benchmark tests. The results for both performance and energy consumption criteria are discussed for two distinct groups: Classic JVM distribution candidates (OpenJDK, Amazon Corretto and Zulu) and GraalVM-based JVM distribution candidates (GraalVM and GraalVM's native image). Furthermore, the carbon footprint (based on

Figure 12. Joulemeter configurations



energy consumption information) for each test iteration was also analysed. This paper focuses on the program execution time in each test for comparative performance analysis. A lower execution time for any JVM candidate infers better performance.

This study also focuses on energy consumption and carbon emissions analyses. Apart from descriptive statistics, Welch's t-test and ANOVA test were used to explore if there was any significant difference between the two candidate groups. The following sub-section presents this analysis in detail.

4.1. Performance Analysis

The aggregate of all five experimental runs for each benchmark test was used to calculate the total execution time for three classic JVM distribution candidates and two GraalVM-based candidates, as shown in Table 3.

The figures in Table 3 also show there are differences between the first and second groups. To be more specific, GraalVM 21.3.1 (build 11.0.14) and its native image performed well in ALS, Chi-square, Decision Tree, Log Regression, Movie Lens, and Page Rank benchmark tests with the lowest aggregated execution time over five iterations amongst all candidates. In contrast, they had poor results when used in Gauss-mix and Naive Bayes benchmark tests, occupying the fourth and fifth ranked positions, respectively. Figure 14 below uses Table 3's data to illustrate the total execution time gap in graphs, and Table 4 utilizes experiment results to conclude the overall performance ranking of each candidate.

From these tables and the figure, it seems to imply that in general GraalVM 21.3.1 applications provide better performance compared with applications utilising OpenJDK 11.0.12, Amazon Corretto 11.0.14.1, and Zulu 11.0.14. Although the difference is not significant, saving even one second of execution time might bring enormous technology benefits for SMEs in the long run.

4.2. Energy Consumption Analysis

A one-way ANOVA one-tailed test and a two-tailed Welch's t-test were employed as inferential analysis methods for Aggregated Average Energy Consumption per second. ANOVA statistical analysis were

Figure 13. Joulemeter with target program name

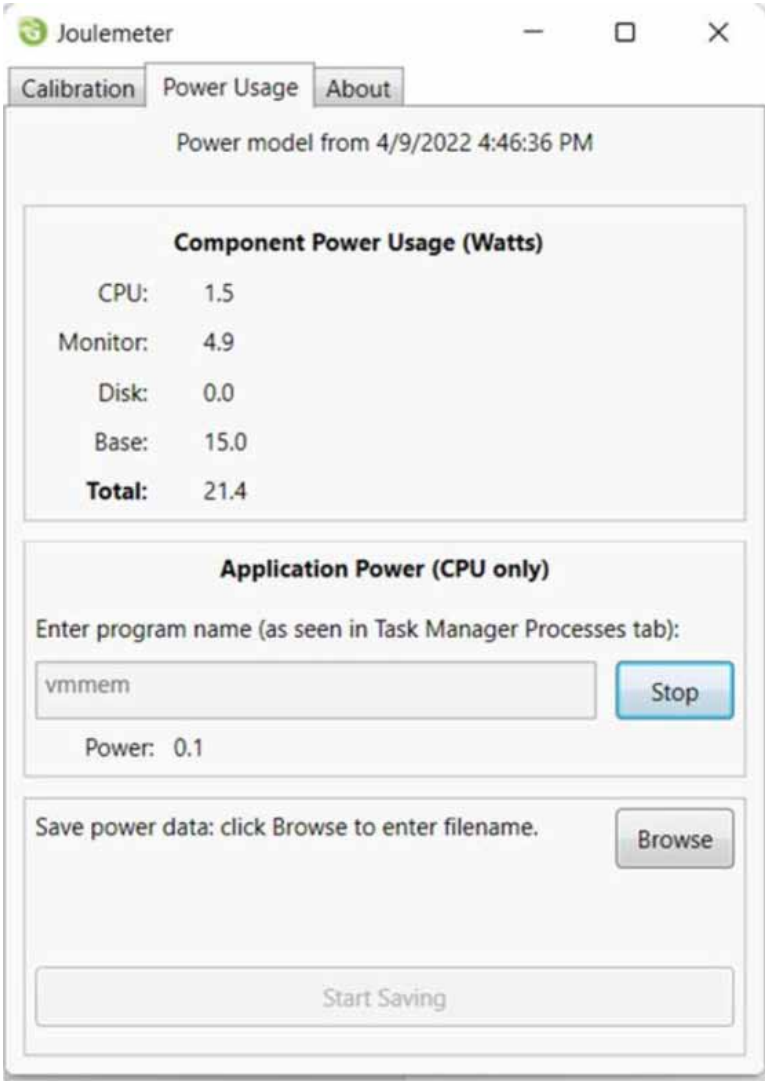


Table 2. Benchmark test specifications (Renaissance, no date)

JVM Distributions	Aggregated Test Case Execution Time (s)							
	ALS	Chi-square	Decision Tree	Gauss-mix	Log Regression	Movie Lens	Naive Bayes	Page Rank
OpenJDK 11.0.12	1346.26	302.30	241.84	258.77	213.14	1336.61	2092.98	961.76
Amazon Corretto 11.0.14	1192.59	272.96	234.62	236.97	193.11	1237.31	2052.57	940.81
Zulu 11.0.14	1135.50	288.14	232.50	241.84	191.76	1237.87	2012.46	908.08
GraalVM 21.3.1 (build 11.0.14)	1125.23	263.17	218.14	255.85	186.53	1189.21	2237.55	864.30
GraalVM 21.3.1 (build 11.0.14) Native Image	1129.64	255.58	217.99	250.40	186.71	1182.09	2206.81	875.43

Table 3. Aggregated execution time in second

Test Case	Description	Repetitions
Chi-square	Execute the chi-square test with the support of Apache Spark library	60
Decision Tree	Runs the Spark ML library's Random Forest algorithm	40
Gaussian Mixture	Using expectation-maximisation, creates a Gaussian mixture model	40
Logistic Regression	Execute the Spark ML library's Logistic Regression algorithm	20
Movie Lens (using the ALS algorithm)	Provide movie recommendations with ALS algorithm	20
Naive Bayes	Runs the Spark ML library's multinomial Naive Bayes algorithm	30
Page Rank	Using Apache Spark's Resilient Distributed Datasets, iterate a number of PageRank	20

conducted for all 8 benchmark tests while Welch's t-tests were conducted for each pair of benchmark tests. Before implementing the ANOVA test and Welch's t-test, two hypotheses were drawn as below:

- ANOVA test
  - H0 (Null hypothesis): There is no significant energy consumption difference between OpenJDK, Amazon Corretto, Zulu, GraalVM, and GraalVM's native image.
  - H1: There is a significant energy consumption difference between OpenJDK, Amazon Corretto, Zulu, GraalVM, and GraalVM's native image.
- Welch's t-test
  - H0 (Null hypothesis): There is no significant energy consumption difference between classic JVM distribution candidates (OpenJDK, Amazon Corretto, and Zulu) and GraalVM-based candidates in the benchmark test.
  - H1: There is a significant energy consumption difference between classic JVM distribution candidates (OpenJDK, Amazon Corretto, and Zulu) and GraalVM-based candidates in the benchmark test.

The chosen confidence level in both tests was 95%, and the alpha value was 0.05. The data presented in Table 5 and Figure 15 was used as inputs for implementing these two tests. A detailed discussion of the results is delivered in the following subsection.

Table 4. Overall performance ranking

JVM Distributions	Individual Performance Ranking								Average	Overall Ranking
	ALS	Chi-square	Decision Tree	Gauss-mix	Log Regression	Movie Lens	Naive Bayes	Page Rank		
OpenJDK 11.0.12	5	5	5	5	5	5	3	5	4.75	5
Amazon Corretto 11.0.14	4	3	4	1	4	3	2	4	3.13	4
Zulu 11.0.14	3	4	3	2	3	4	1	3	2.88	3
GraalVM 21.3.1 (build 11.0.14)	1	2	2	4	1	2	5	1	2.25	2
GraalVM 21.3.1 (build 11.0.14) Native Image	2	1	1	3	2	1	4	2	2	1

Table 5. Aggregated average energy consumption per second (J/s)

JVM Distributions	Aggregated Average Energy Consumption per second (J/s)							
	ALS	Chi-square	Decision Tree	Gauss-mix	Log Regression	Movie Lens	Naive Bayes	Page Rank
OpenJDK 11.0.12	17191.43	2717.43	2329.65	2184.67	2301.90	15874.61	29208.98	11051.08
Amazon Corretto 11.0.14	15134.75	2451.18	2256.14	1974.39	2246.82	14332.56	28513.69	10762.44
Zulu 11.0.14	14506.91	2555.63	2214.98	2021.06	2223.55	14251.94	27908.34	10155.16
GraalVM 21.3.1 (build 11.0.14)	14351.81	2367.38	2051.26	2119.02	2168.60	13682.63	31193.58	9827.60
GraalVM 21.3.1 (build 11.0.14) Native Image	14348.20	2291.95	2039.06	2079.51	2165.58	13690.66	30756.33	10047.12

This paper also analyses and compares other factors in each benchmark test. Those factors include Aggregated Hardware Energy Consumption, Aggregated Application Energy Consumption, Aggregated CPU, Aggregated Monitor, Aggregated Disk, and Aggregated Base Energy Consumption.

4.3. ANOVA test

Given the data in Table 5, the p-value for the one-way ANOVA one-tailed test is 0.999821686. Since this p-value is greater than the alpha value (0.05), the null hypothesis is accepted. The overall conclusion is that there is no significant difference between all testing candidates in terms of energy consumption when implementing the eight benchmark tests with five iterations. Table 6 presents this result and other relevant information.

4.4. Welch's T-Test

Results from conducting the two-tailed Welch's t-test for unequal variances are reported for each individual benchmark test in Tables 7-14. A t-test was used in this manner to test the hypothesis for candidates in every specific benchmark test.

1. *ALS:* p-value = 0.260367 > alpha = 0.05, therefore, the null hypothesis is accepted for ALS benchmark test.
2. *Chi-square:* p-value = 0.066003 > alpha = 0.05, therefore, the null hypothesis is accepted for the Chi-square benchmark test.
3. *Decision Tree:* p-value = 0.023023 < alpha = 0.05, therefore, the null hypothesis is rejected for the Decision Tree benchmark test.
4. *Gaussian Mixture:* p-value = 0.60798 > alpha = 0.05, therefore, the null hypothesis is accepted for the Gaussian Mixture benchmark test.
5. *Log Regression:* p-value = 0.061698 > alpha = 0.05, therefore, the null hypothesis is accepted for the Log Regression benchmark test.
6. *Movie Lens:* p-value = 0.165228 > alpha = 0.05, therefore, the null hypothesis is accepted for the Movie Lens benchmark test.
7. *Naive Bayes:* p-value = 0.011279 < alpha = 0.05, therefore, the null hypothesis is rejected for the Naive Bayes benchmark test.
8. *Page Rank:* p-value = 0.086889 > alpha = 0.05, therefore, the null hypothesis is accepted for the Page Rank benchmark test.

From the above-reported results, it is clear that in 6/8 cases the null hypothesis can be accepted. It means there is no significant difference in power consumption between the classic JVM distribution candidates and the GraalVM-based candidates when executing algorithms or programs related to ALS, Chi-square, Gaussian Mixture, Movie Lens, Page Rank, and Log Regression. For Decision Tree and Naive Bayes relevant programs, however, the energy consumption of candidate groups is significantly different.

Although the t-test results (table 8-14) showed that the difference between these two groups is not significant, GraalVM-based candidates were still leading in most cases. Based on Table 5, a ranking table was generated to rank the energy consumption between candidates (Table 15). This result matches the finding of Ournani et al. in 2021 that GraalVM's energy consumption was lower than other candidates in most cases.

Table 6. ANOVA test result

Groups	Count	Sum	Average	Variance
OpenJDK 11.0.12	8	82859.74	10357.47	98215009.00
Amazon Corretto 11.0.14.1	8	77671.97	9708.99	89895943.00
Zulu 11.0.14	8	75837.58	9479.69	85302863.00
GraalVM 21.3.1 (build 11.0.14)	8	77761.88	9720.24	103707342.90
GraalVM 21.3.1 (build 11.0.14) Native Image	8	77418.41	9677.30	101346468.30

Table 6. B

Source of Variation	SS	df	MS	F	P-value	F crit
Between Groups	3539430.00	4	884857.60	0.009246786	0.999822	2.641465
Within Groups	3349273380.00	35	95693525.00			
Total	3352812811.00	39				

Table 7. ALS T-test results

	Variable 1	Variable 2
Mean	2574.748	2329.664
Variance	17995.810	2844.457
Observations	3	2
Hypothesised Mean Difference	1	
df	3	
t Stat	2.833428	
P(T=t) one-tail	0.033001	
t Critical one-tail	2.353363	
P(T=t) two-tail	**0.066003**	
t Critical two-tail	3.182446	

Table 8. Chi-square T-test results

	Variable 1	Variable 2
Mean	15611.03	14350.00
Variance	1971790.00	6.5126
Observations	3	2
Hypothesised Mean Difference	1	
df	2	
t Stat	1.554209	
P(T=t) one-tail	0.130184	
t Critical one-tail	2.919986	
P(T=t) two-tail	**0.260367**	
t Critical two-tail	4.302653	

Table 9. Decision tree t-test results

	Variable 1	Variable 2
Mean	2266.926	2045.160
Variance	3374.351	74.527
Observations	3	2
Hypothesised Mean Difference	1	
df	2	
t Stat	6.476201	
P(T=t) one-tail	0.011511	
t Critical one-tail	2.919986	
P(T=t) two-tail	**0.023023**	
t Critical two-tail	4.302653	

Table 10. Gaussian mixture t-test results

	Variable 1	Variable 2
Mean	2060.042	2099.265
Variance	12194.560	780.335
Observations	3	2
Hypothesised Mean Difference	1	
df	2	
t Stat	-0.602640	
P(T=t) one-tail	0.303990	
t Critical one-tail	2.919986	
P(T=t) two-tail	**0.607980**	
t Critical two-tail	4.302653	

Table 11. Log regression t-test results

	Variable 1	Variable 2
Mean	2257.423	2167.091
Variance	1619.141	4.561
Observations	3	2
Hypothesised Mean Difference	1	
df	2	
t Stat	3.837187	
P(T=t) one-tail	0.030849	
t Critical one-tail	2.919986	
P(T=t) two-tail	**0.061698**	
t Critical two-tail	4.302653	

Table 12. Movie lens t-test results

	Variable 1	Variable 2
Mean	14819.70	13686.65
Variance	836242.10	32.26
Observations	3	2
Hypothesised Mean Difference	1	
df	2	
t Stat	2.144126	
P(T=t) one-tail	0.082614	
t Critical one-tail	2.919986	
P(T=t) two-tail	**0.165228**	
t Critical two-tail	4.302653	

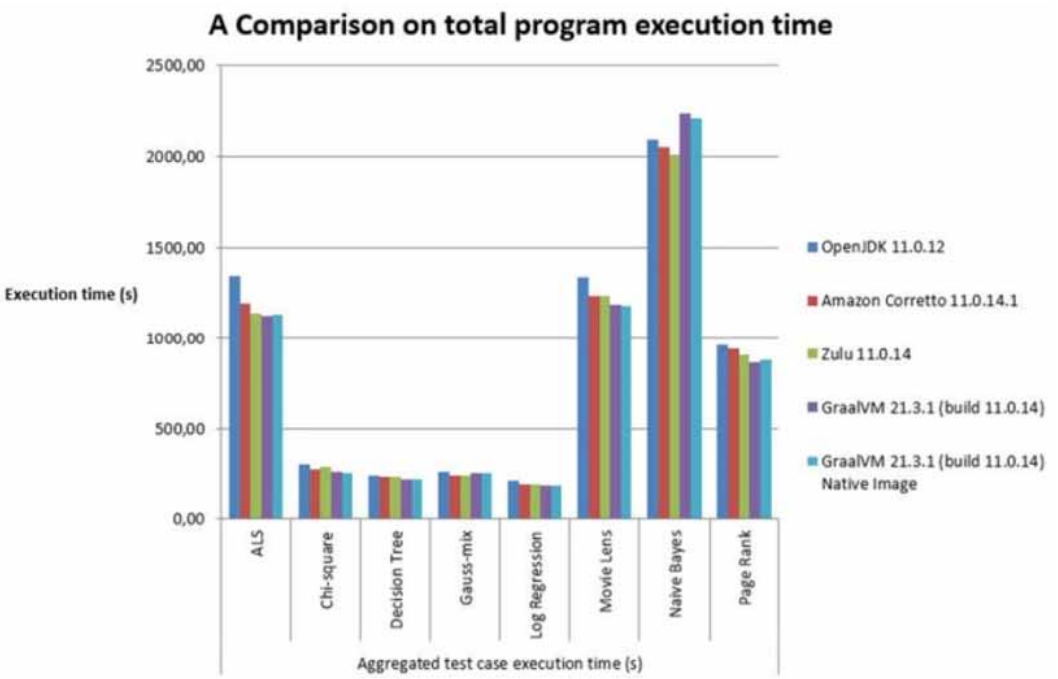
Table 13. Naive bayes t-test results

	Variable 1	Variable 2
Mean	28543.67	30974.96
Variance	423585.90	95591.56
Observations	3	2
Hypothesised Mean Difference	1	
df	3	
t Stat	-5.594920	
P(T=t) one-tail	0.005639	
t Critical one-tail	2.353363	
P(T=t) two-tail	**0.011279**	
t Critical two-tail	3.182446	

Table 14. Page rank t-test results

	Variable 1	Variable 2
Mean	10656.23	9937.36
Variance	209127.10	24093.18
Observations	3	2
Hypothesised Mean Difference	1	
Df	3	
t Stat	2.510635	
P(T=t) one-tail	0.043445	
t Critical one-tail	2.353363	
P(T=t) two-tail	**0.086889**	
t Critical two-tail	3.182446	

Figure 14. Comparison of total program execution time in seconds



4.5. Other Factors

The previous subsections clearly demonstrated that, when analysing the aggregated average energy consumption, GraalVM and its native image use less power than classic JVM distributions. In this sub-section, to further support this conclusion, sub-factors such as aggregated CPU, aggregated monitor, aggregate disk, aggregated base, and aggregated application energy consumed when executing benchmark tests are also considered. The following tables (Table 16-23) and figures (Figure 16 -31) for each benchmark test's power consumption components follow the same pattern as aggregated average energy consumption statistics. In terms of total hardware energy and aggregated application

Table 15. Energy consumption ranking table

JVM Distributions	Individual Aggregated Average Energy Consumption Ranking								Average	Overall Ranking
	ALS	Chi-square	Decision Tree	Gauss-mix	Log Regression	Movie Lens	Naive Bayes	Page Rank		
OpenJDK 11.0.12	5	5	5	5	5	5	3	5	4.75	5
Amazon Corretto 11.0.14	4	3	4	1	4	4	2	4	3.25	4
Zulu 11.0.14	3	4	3	2	3	3	1	3	2.75	3
GraalVM 21.3.1 (build 11.0.14)	2	2	2	4	2	1	5	1	2.38	2
GraalVM 21.3.1 (build 11.0.14) Native Image	1	1	1	3	1	2	4	2	1.88	1

Table 16. Energy consumption in detail for ALS benchmark test by components

JVM Distribution	Aggregated Average Hardware Energy Consumption (KJ)					Aggregated Application (KJ)	Aggregated Total Time (s)	Aggregated Energy Consumption (KJ)	Aggregated Energy Consumption (J/s)
	CPU (KJ)	Monitor (KJ)	Disk (KJ)	Base (KJ)	Total Hardware Energy (KJ)				
OpenJDK 11.0.12	7330.90	3500.40	62.87	5250.60	16144.78	6999.32	1346.26	23144.10	17191.43
Amazon Corretto 11.0.14.1	5705.84	2744.33	49.27	4116.49	12615.93	5433.63	1192.59	18049.57	15134.75
Zulu 11.0.14	5211.12	2488.81	40.62	3733.22	11473.77	4998.82	1135.50	16472.59	14506.91
GraalVM 21.3.1 (build 11.0.14)	5134.83	2442.69	41.20	3664.03	11282.74	4866.34	1125.23	16149.08	14351.81
GraalVM 21.3.1 (build 11.0.14) Native Image	5143.56	2460.85	45.89	3691.28	11341.58	4866.70	1129.64	16208.28	14348.20

Table 17. Energy consumption in detail for chi-square benchmark test by components

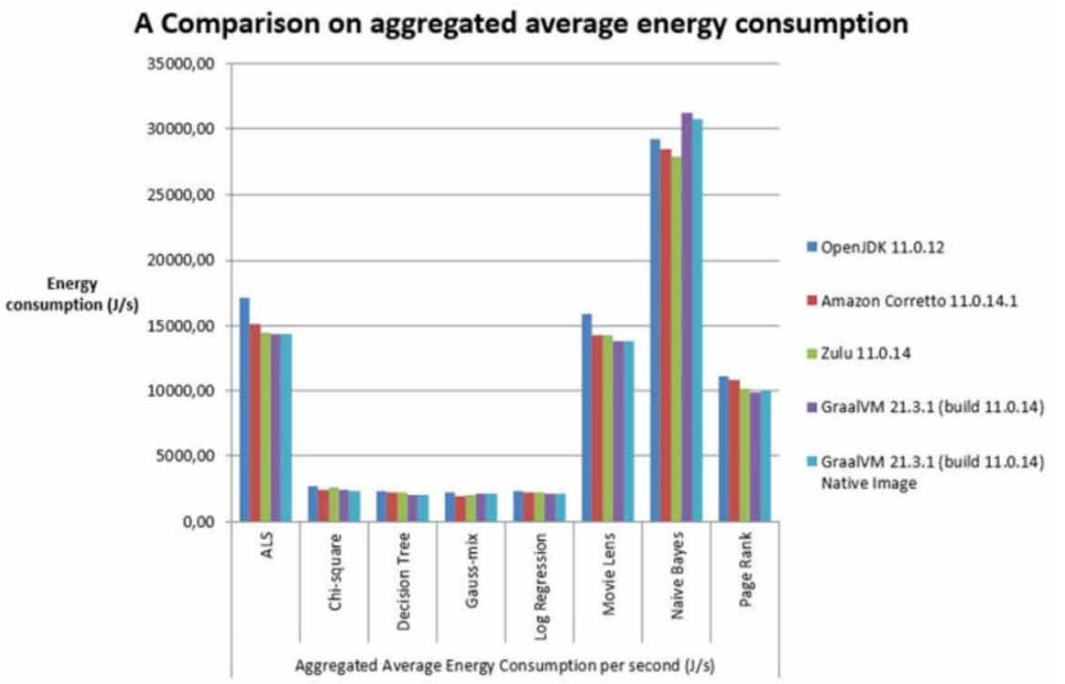
JVM Distribution	Aggregated Average Hardware Energy Consumption (KJ)					Aggregated Application (KJ)	Aggregated Total Time (s)	Aggregated Energy Consumption (KJ)	Aggregated Energy Consumption (J/s)
	CPU (KJ)	Monitor (KJ)	Disk (KJ)	Base (KJ)	Total Hardware Energy (KJ)				
OpenJDK 11.0.12	200.71	177.68	3.83	266.52	648.74	172.73	302.30	821.47	2717.43
Amazon Corretto 11.0.14.1	161.84	146.01	3.43	218.65	529.93	139.15	272.96	669.08	2451.18
Zulu 11.0.14	177.38	161.23	3.59	241.84	584.05	152.32	288.14	736.37	2555.63
GraalVM 21.3.1 (build 11.0.14)	154.20	134.66	3.26	201.99	494.11	128.90	263.17	623.01	2367.38
GraalVM 21.3.1 (build 11.0.14) Native Image	144.95	126.41	3.18	189.62	464.16	121.62	255.58	585.77	2291.95

power consumption, GraalVM 21.3.1 and its native image are the least energy consumption candidates among five JVM distributions in ALS, Chi-square, Decision Tree, Logistic Regression, Movie Lens, and Page Rank test cases. However, similar to the aggregated average energy consumption statistics

Table 18. Energy consumption in detail for decision tree test case by components

JVM Distribution	Aggregated Average Hardware Energy Consumption (KJ)					Aggregated Application (KJ)	Aggregated Total Time (s)	Aggregated Energy Consumption (KJ)	Aggregated Energy Consumption (J/s)
	CPU (KJ)	Monitor (KJ)	Disk (KJ)	Base (KJ)	Total Hardware Energy (KJ)				
OpenJDK 11.0.12	143.53	113.19	2.74	169.78	429.24	134.15	241.84	563.39	2329.65
Amazon Corretto 11.0.14.1	134.69	106.55	2.71	159.82	403.77	125.55	234.62	529.32	2256.14
Zulu 11.0.14	128.78	104.65	2.74	156.97	393.14	121.84	232.50	514.98	2214.98
GraalVM 21.3.1 (build 11.0.14)	111.83	92.06	2.54	138.09	344.53	102.93	218.14	447.46	2051.26
GraalVM 21.3.1 (build 11.0.14) Native Image	110.22	92.02	2.53	138.04	342.81	101.69	217.99	444.50	2039.06

Figure 15. A comparison on aggregated average energy Consumption



and performance results, they performed poorly in Naive Bayes and Gaussian Mixture test cases with high total hardware energy and aggregated application consumption.

4.6. Carbon Equivalent Footprint Analysis

The carbon equivalent emissions statistical analyses in this paper are computed based on the aggregated total energy consumption in each benchmark test. Using GHG conversion factors for electricity in the UK as a reference, four types of emission metrics were considered: kgCO2e/kWh, kgCO2/kWh, kgCH4/kWh, and kgN2O/kWh (Department for Business, Energy & Industrial Strategy, 2021). Table 24 below presents calculated data for all these constructs. To reiterate, the aggregated total energy

Table 19. Energy consumption in detail for gaussian mixture test case by components

JVM Distribution	Aggregated Average Hardware Energy Consumption (KJ)					Aggregated Application (KJ)	Aggregated Total Time (s)	Aggregated Energy Consumption (KJ)	Aggregated Energy Consumption (J/s)
	CPU (KJ)	Monitor (KJ)	Disk (KJ)	Base (KJ)	Total Hardware Energy (KJ)				
OpenJDK 11.0.12	126.56	130.01	2.4	195.02	454.00	111.33	258.77	565.33	2184.67
Amazon Corretto 11.0.14.1	100.15	109.07	2.08	163.61	374.90	92.96	236.97	467.87	1974.39
Zulu 11.0.14	105.35	113.82	2.19	170.73	392.09	96.69	241.84	488.78	2021.06
GraalVM 21.3.1 (build 11.0.14)	116.37	127.00	2.58	190.50	436.45	105.70	255.85	542.16	2119.02
GraalVM 21.3.1 (build 11.0.14) Native Image	112.72	121.71	2.79	182.56	419.77	100.94	250.40	520.71	2079.51

Table 20. Energy consumption in detail for logistic regression test case by components

JVM Distribution	Aggregated Average Hardware Energy Consumption (KJ)					Aggregated Application (KJ)	Aggregated Total Time (s)	Aggregated Energy Consumption (KJ)	Aggregated Energy Consumption (J/s)
	CPU (KJ)	Monitor (KJ)	Disk (KJ)	Base (KJ)	Total Hardware Energy (KJ)				
OpenJDK 11.0.12	139.23	88.79	1.91	133.18	363.11	127.51	213.14	490.62	2301.90
Amazon Corretto 11.0.14.1	132.21	71.84	2.40	107.77	314.22	119.67	193.11	433.89	2246.82
Zulu 11.0.14	129.00	70.97	2.03	106.45	308.46	117.92	191.76	426.38	2223.55
GraalVM 21.3.1 (build 11.0.14)	123.25	67.17	1.36	100.76	292.54	111.96	186.53	404.51	2168.60
GraalVM 21.3.1 (build 11.0.14) Native Image	122.05	67.22	1.59	100.82	291.68	112.66	186.71	404.34	2165.58

Table 21. Energy consumption in detail for movie lens test case by components

JVM Distribution	Aggregated Average Hardware Energy Consumption (KJ)					Aggregated Application (KJ)	Aggregated Total Time (s)	Aggregated Energy Consumption (KJ)	Aggregated Energy Consumption (J/s)
	CPU (KJ)	Monitor (KJ)	Disk (KJ)	Base (KJ)	Total Hardware Energy (KJ)				
OpenJDK 11.0.12	6417.36	3463.43	20.38	5195.15	15096.32	6121.87	1336.61	21218.19	15874.61
Amazon Corretto 11.0.14.1	5281.71	2970.32	17.07	4455.48	12724.56	5009.20	1237.31	17733.79	14332.56
Zulu 11.0.14	5285.95	2968.58	35.31	4452.86	12742.70	4899.38	1237.87	17642.08	14251.94
GraalVM 21.3.1 (build 11.0.14)	4823.10	2745.20	19.10	4117.80	11705.20	4566.29	1189.21	16271.49	13682.63
GraalVM 21.3.1 (build 11.0.14) Native Image	4822.33	2709.09	18.02	4063.63	11613.07	4570.47	1182.09	16183.54	13690.66

consumption is the input for the relevant carbon equivalent computation. In most cases (except Gauss Mix and Naïve Bayes), GraalVM-based benchmark tests emitted the lowest amount of carbon equivalent into the environment.

Table 22. Energy consumption in detail for Naive Bayes test case by components

JVM Distribution	Aggregated Average Hardware Energy Consumption (KJ)					Aggregated Application (KJ)	Aggregated Total Time (s)	Aggregated Energy Consumption (KJ)	Aggregated Energy Consumption (J/s)
	CPU (KJ)	Monitor (KJ)	Disk (KJ)	Base (KJ)	Total Hardware Energy (KJ)				
OpenJDK 11.0.12	21004.53	8315.54	123.78	12473.32	41917.17	19216.69	2092.98	61133.86	29208.98
Amazon Corretto 11.0.14.1	20072.74	7999.36	168.78	11999.04	40239.92	18286.48	2052.57	58526.40	28513.69
Zulu 11.0.14	19207.79	7678.28	120.07	11517.42	38523.57	17640.88	2012.46	56164.45	27908.34
GraalVM 21.3.1 (build 11.0.14)	23941.84	9523.88	138.06	14285.82	47889.59	21907.66	2237.55	69797.26	31193.58
GraalVM 21.3.1 (build 11.0.14) Native Image	23192.35	9263.21	129.28	13894.81	46479.66	21393.61	2206.81	67873.26	30756.33

Table 23. Energy consumption in detail for page rank test case by components

JVM Distribution	Aggregated Average Hardware Energy Consumption (KJ)					Aggregated Application (KJ)	Aggregated Total Time (s)	Aggregated Energy Consumption (KJ)	Aggregated Energy Consumption (J/s)
	CPU (KJ)	Monitor (KJ)	Disk (KJ)	Base (KJ)	Total Hardware Energy (KJ)				
OpenJDK 11.0.12	3222.08	1789.95	23.72	2684.93	7720.67	2907.78	961.76	10628.45	11051.08
Amazon Corretto 11.0.14.1	3058.59	1714.52	20.59	2571.78	7365.49	2759.88	940.81	10125.37	10762.44
Zulu 11.0.14	2745.79	1598.57	17.25	2397.86	6759.48	2462.19	908.08	9221.66	10155.16
GraalVM 21.3.1 (build 11.0.14)	2549.51	1448.05	19.32	2172.07	6188.94	2305.03	864.30	8493.97	9827.60
GraalVM 21.3.1 (build 11.0.14) Native Image	2659.90	1485.13	20.07	2227.70	6392.81	2402.71	875.43	8795.52	10047.12

5. CONCLUSION AND RECOMMENDATIONS

Overall, this study has clearly demonstrated that, in most cases, GraalVM-based Java 11 applications provide better performance, consume less energy, and emit less carbon equivalent footprint than those utilising OpenJDK and two other popular JVM distributions in the market (Amazon Corretto and Zulu). In terms of energy efficiency, this result supports Ournani's research outcome, which indicated that GraalVM had the highest energy efficiency for most benchmarks (Ournani et al., 2021).

To achieve these results, eight Apache Spark benchmark tests from the Renaissance benchmark suite have been chosen for analysing three classic JVM distribution candidates and two GraalVM-based distribution candidates. Each test was conducted five times to ensure fairness and consistency of the experiments.

Although the outcomes of the test groups are not significantly different, in the long term GraalVM would still add technology-related value to SMEs in terms of reduced monthly electricity bills and carbon equivalent footprint emissions while maximising business profits (as a results of such technological acceptance). For GraalVM's founders and contributors, their products reveal weakness in specific benchmark tests such as Naive Bayes and Gaussian Mixture Model, with poor performance, high energy consumption, and high carbon equivalent emissions. Therefore, some work is still needed to enable GraalVM categorically outperform other JVM choices.

Table 24. Carbon equivalent footprint statistics for all test cases

Test Case	Aggregated Total Energy Consumption (KJ)	Aggregated Total Energy Consumption (kWh)	Carbon Emission (kgCO2e/ kWh)	Carbon Emission (kgCO2/ kWh)	Carbon Emission (kgCH4/ kWh)	Carbon Emission (kgN2O/ kWh)
OpenJDK 11.0.12 ALS	23144.10	6.428916589	1.365051859	1.351101110	0.005143133	0.008807616
Amazon Corretto 11.0.14.1 ALS Zulu 11.0.14 ALS	18049.57 16472.59	5.013768075 4.575718169	1.064573375 0.971562239	1.053693499 0.961632930	0.004011014 0.003660575	0.006868862 0.006268734
GraalVM 21.3.1 (build 11.0.14) ALS	**16149.08**	**4.485856692**	**0.952481951**	**0.942747642**	**0.003588685**	**0.006145624**
Native Image GraalVM 21.3.1 (build 11.0.14) ALS	**16208.28**	**4.502301192**	**0.955973612**	**0.946203619**	**0.003601841**	**0.006168153**
OpenJDK 11.0.12 Chi-square	821.47	0.228185429	0.048450612	0.047955450	0.000182548	0.000312614
Amazon Corretto 11.0.14.1 Chi-square	669.08	0.185855355	0.039462668	0.039059361	0.000148684	0.000254622
Zulu 11.0.14 Chi-square	736.37	0.204548048	0.043431687	0.042987818	0.000163638	0.000280231
GraalVM 21.3.1 (build 11.0.14) Chi-square	**623.01**	**0.173058532**	**0.036745518**	**0.036369981**	**0.000138447**	**0.000237090**
Native Image GraalVM 21.3.1 (build 11.0.14) Chi-square	**585.77**	**0.162715216**	**0.034549322**	**0.034196230**	**0.000130172**	**0.000222920**
OpenJDK 11.0.12 Decision Tree	563.39	0.156498235	0.033229270	0.032889669	0.000125199	0.000214403
Amazon Corretto 11.0.14.1 Decision Tree	529.32	0.147034604	0.031219857	0.030900792	0.000117628	0.000201437
Zulu 11.0.14 Decision Tree	514.98	0.143050492	0.030373911	0.030063491	0.000114440	0.000195979
GraalVM 21.3.1 (build 11.0.14) Decision Tree	**447.46**	**0.124294081**	**0.026391362**	**0.026121644**	**0.000099440**	**0.000170283**
Native Image GraalVM 21.3.1 (build 11.0.14) Decision Tree	**444.50**	**0.123472173**	**0.026216847**	**0.025948912**	**0.000098780**	**0.000169157**
OpenJDK 11.0.12 Gauss Mix	565.33	0.157035603	0.033343370	0.033002602	0.000125628	0.000215139
Amazon Corretto 11.0.14.1 Gauss Mix	**467.87**	**0.129963077**	**0.027595060**	**0.027313040**	**0.000103970**	**0.000178049**
Zulu 11.0.14 Gauss Mix	**488.78**	**0.135771513**	**0.028828365**	**0.028533741**	**0.000108617**	**0.000186007**
GraalVM 21.3.1 (build 11.0.14) Gauss Mix	542.16	0.150599178	0.031976724	0.031649923	0.000120479	0.000206321
Native Image GraalVM 21.3.1 (build 11.0.14) Gauss Mix	520.71	0.144641642	0.030711760	0.030397887	0.000115713	0.000198159
OpenJDK 11.0.12 Log Regression	490.62	0.136282184	0.028936796	0.028641064	0.000109026	0.000186707
Amazon Corretto 11.0.14.1 Log Regression Zulu 11.0.14 Log Regression	433.89 426.38	0.120524361 0.118437944	0.025590938 0.025147929	0.025329400 0.024890918	0.000096420 0.000094750	0.000165118 0.000162260
GraalVM 21.3.1 (build 11.0.14) Log Regression	**404.51**	**0.112363637**	**0.023858171**	**0.023614342**	**0.000089890**	**0.000153938**
Native Image GraalVM 21.3.1 (build 11.0.14) Log Regression	**404.34**	**0.112315420**	**0.023847933**	**0.023604209**	**0.000089850**	**0.000153872**
OpenJDK 11.0.12 Movie Lens	21218.19	5.893941545	1.251460608	1.238670755	0.004715153	0.008074700
Amazon Corretto 11.0.14.1 Movie Lens	17733.79	4.926054004	1.045949047	1.035259509	0.003940843	0.006748694
Zulu 11.0.14 Movie Lens	17642.08	4.900576426	1.040539392	1.029905142	0.003920461	0.006713790
GraalVM 21.3.1 (build 11.0.14) Movie Lens	**16271.49**	**4.519859026**	**0.959701667**	**0.949893573**	**0.003615887**	**0.006192207**
Native Image GraalVM 21.3.1 (build 11.0.14) Movie Lens	**16183.54**	**4.495427734**	**0.954514171**	**0.944759093**	**0.003596342**	**0.006158736**
OpenJDK 11.0.12 Naive Bayes	61133.86	16.98162829	3.605709135	3.568859002	0.013585303	0.023264831
Amazon Corretto 11.0.14.1 Naive Bayes	**58526.40**	**16.25733415**	**3.451919760**	**3.416641345**	**0.013005867**	**0.022272548**
Zulu 11.0.14 Naive Bayes	**56164.45**	**15.60123658**	**3.312610562**	**3.278755879**	**0.012480989**	**0.021373694**
GraalVM 21.3.1 (build 11.0.14) Naive Bayes	69797.26	19.38812666	4.116680934	4.074608700	0.015510501	0.026561734
Native Image GraalVM 21.3.1 (build 11.0.14) Naive Bayes	67873.26	18.853684300	4.003202787	3.962290292	0.015082947	0.025829547
OpenJDK 11.0.12 Page Rank	10628.45	2.952346927	0.626871823	0.620465230	0.002361878	0.004044715
Amazon Corretto 11.0.14.1 Page Rank	10125.37	2.812602721	0.597199936	0.591096588	0.002250082	0.003853266
Zulu 11.0.14 Page Rank	9221.66	2.561571577	0.543898493	0.538339883	0.002049257	0.003509353
GraalVM 21.3.1 (build 11.0.14) Page Rank	**8493.97**	**2.359435893**	**0.500979023**	**0.495859047**	**0.001887549**	**0.003232427**
Native Image GraalVM 21.3.1 (build 11.0.14) Page Rank	**8795.52**	**2.443199686**	**0.518764589**	**0.513462846**	**0.001954560**	**0.003347184**

Due to hardware and software limitations, this research was conducted in a non-dedicated environment. Thus, external factors could have interfered with the final results, such as phantom processes within the WSL2 environment, or CPU usage from other Windows background processes. For this reason, future work can utilize a more specialised environment with more advanced tools to measure performance, energy consumption, and carbon equivalent footprint emissions of the

Figure 16. Total hardware and application energy consumption of ALS

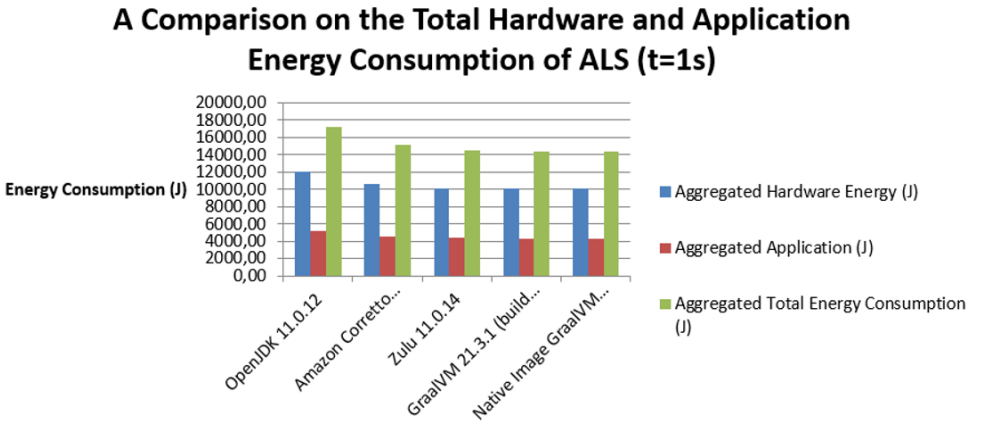


Figure 17. Comparison of hardware energy consumption on ALS

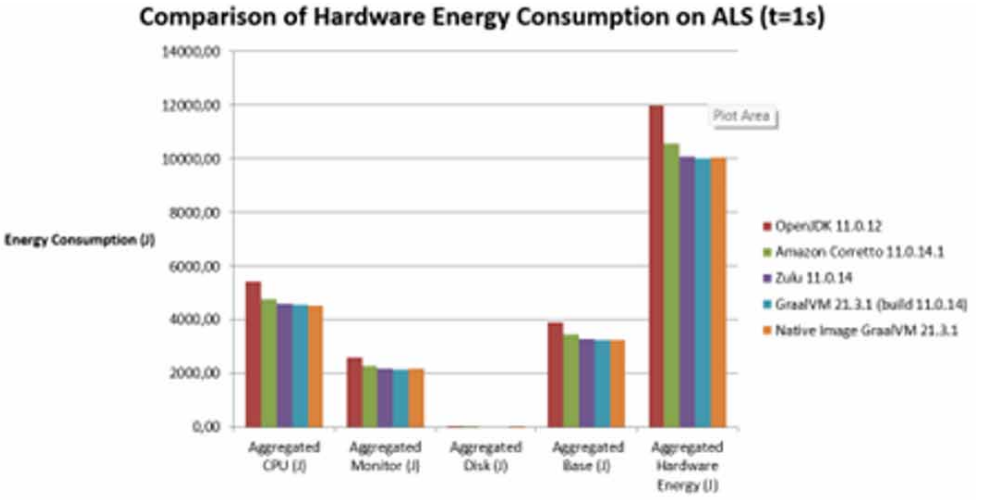


Figure 18. Total hardware and application energy consumption of chi-square

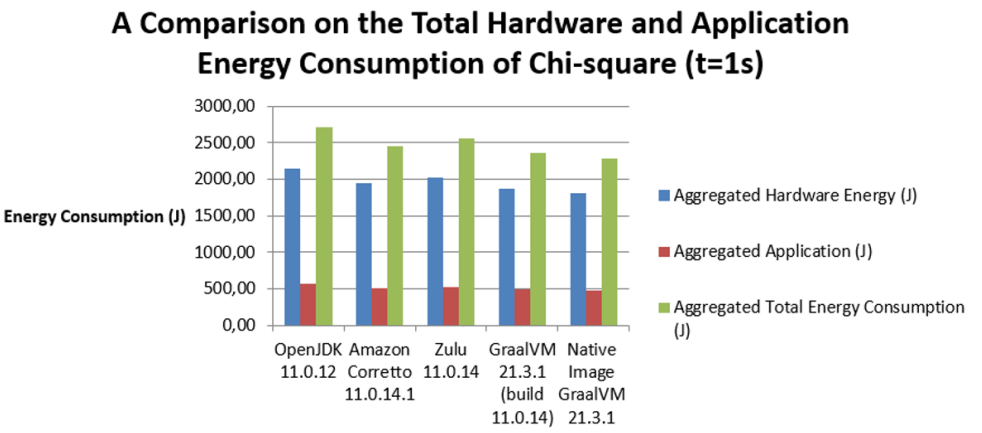


Figure 19. A Comparison of hardware energy consumption on chi-square

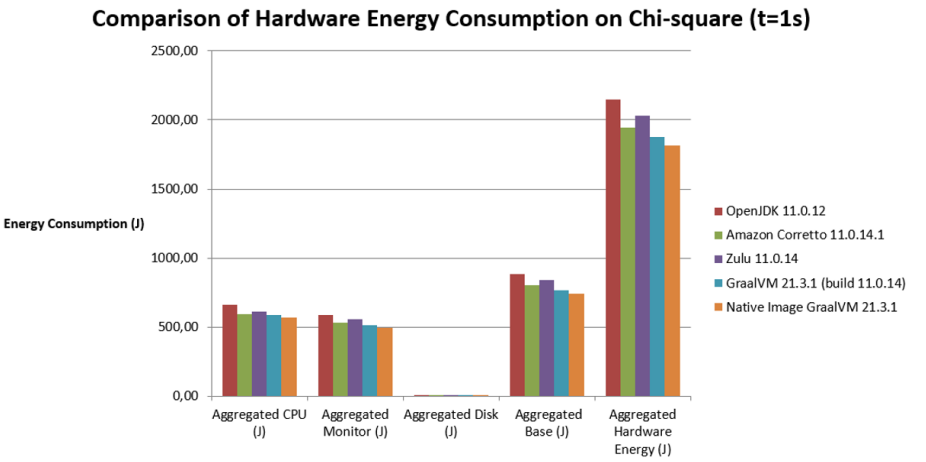


Figure 20. Total hardware and application energy consumption of decision tree

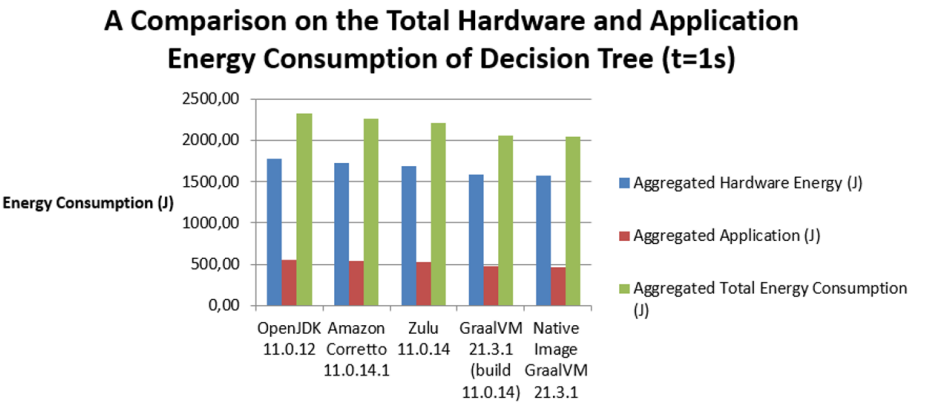


Figure 21. Comparison of hardware energy consumption on decision tree

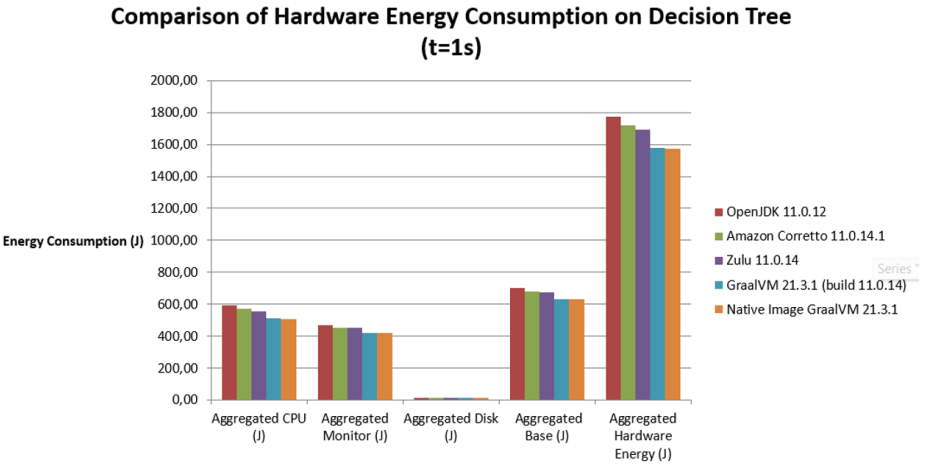


Figure 22. Total hardware and application energy consumption of Gaussian mixture

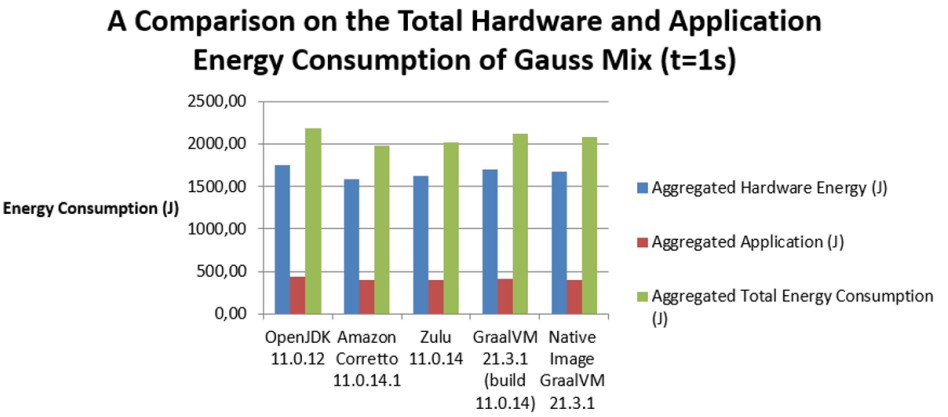


Figure 23. A comparison of hardware energy consumption on Gaussian mixture

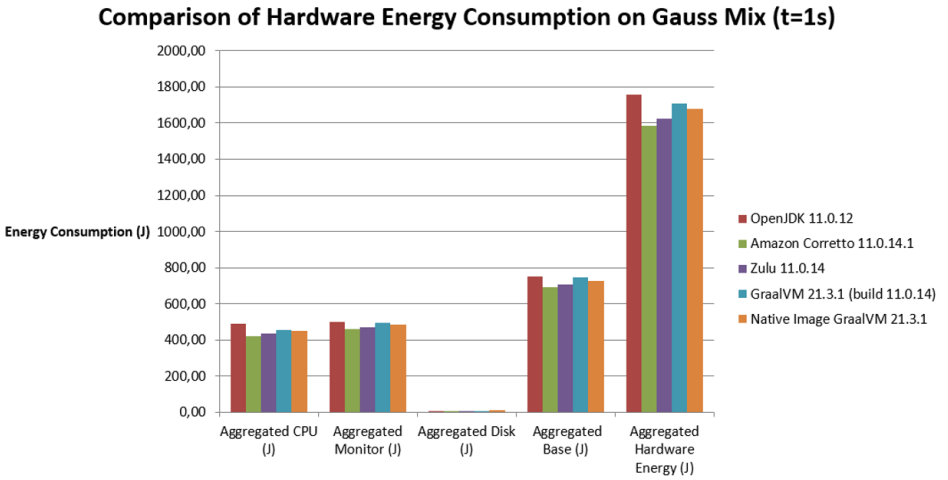


Figure 24. Total hardware and application energy consumption of log regression

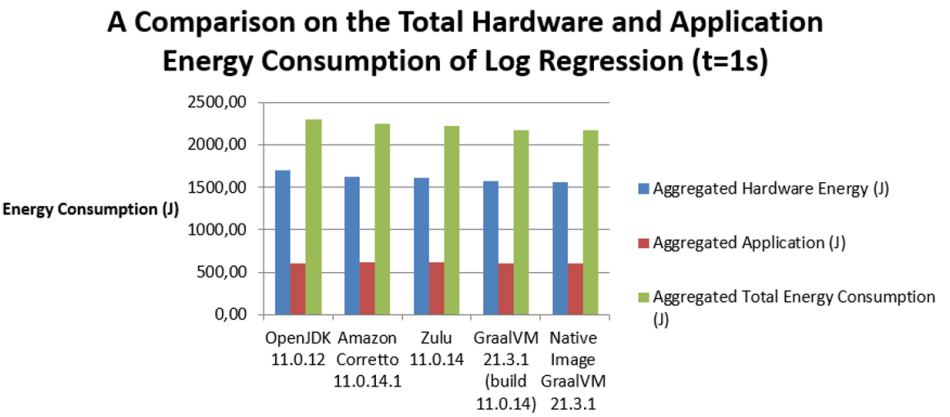


Figure 25. A comparison of hardware energy consumption on log regression

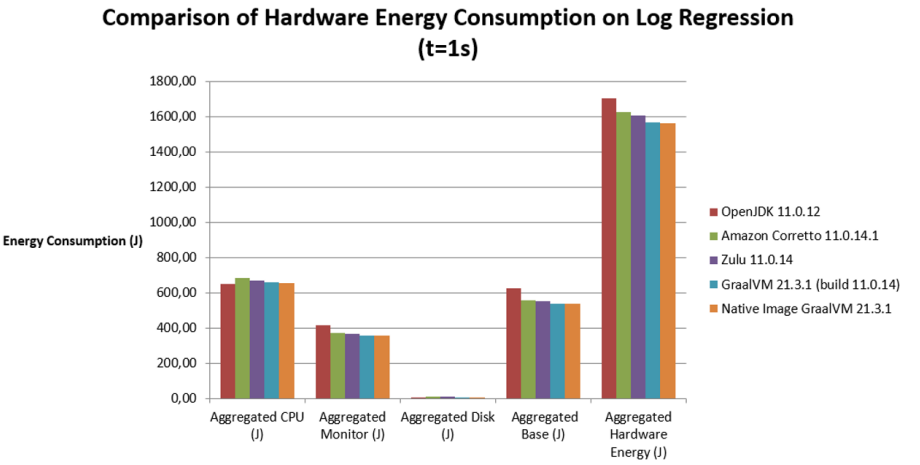


Figure 26. Total hardware and application energy consumption of movie lens

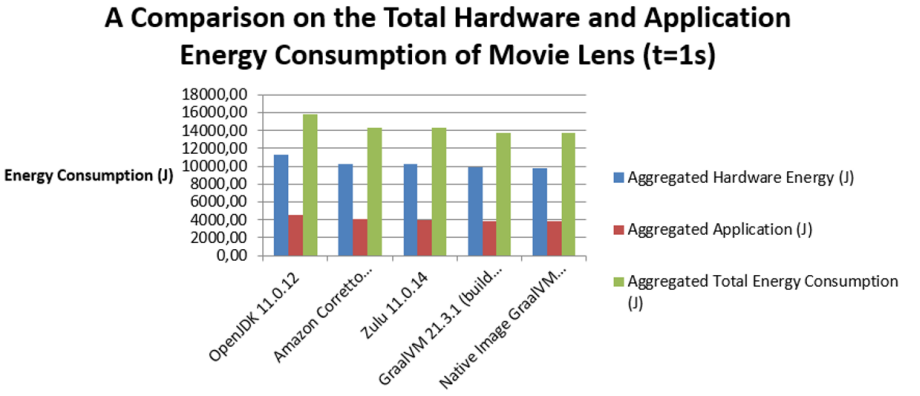


Figure 27. A Comparison of hardware energy consumption on movie lens

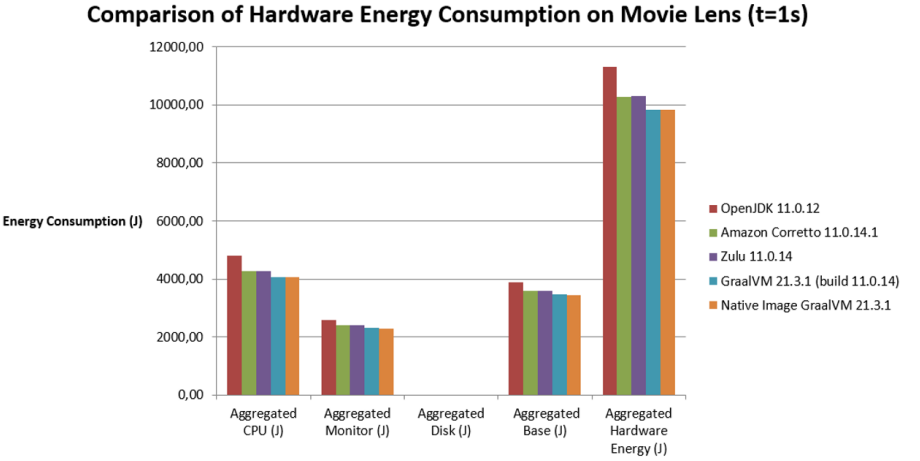


Figure 28. Total hardware and application energy consumption of Naïve Bayes

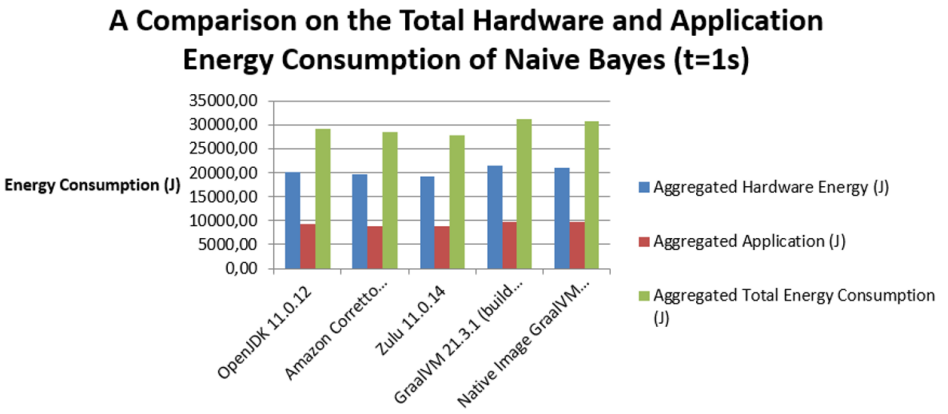


Figure 29. Comparison of hardware energy consumption on Naïve Bayes

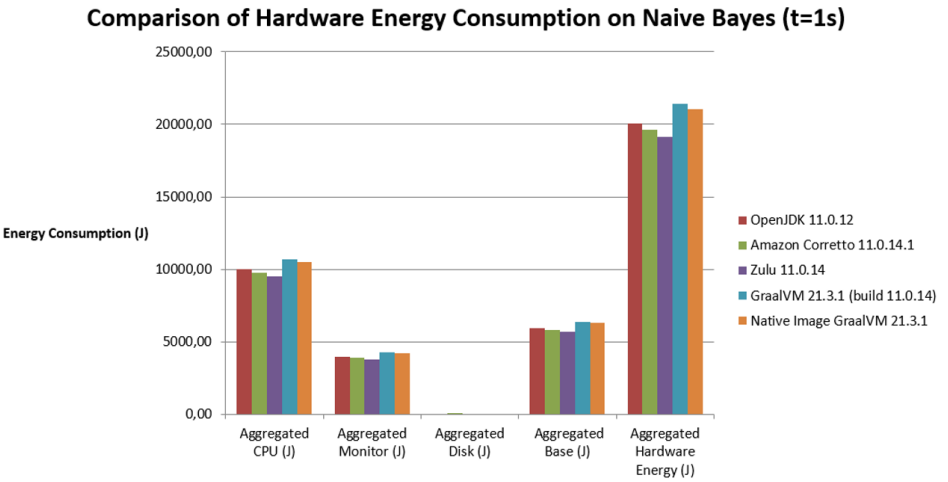


Figure 30. Total hardware and application energy consumption of page rank

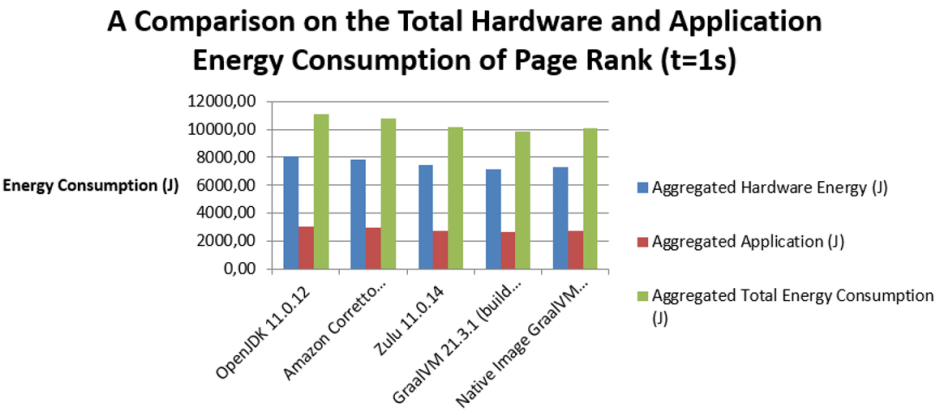
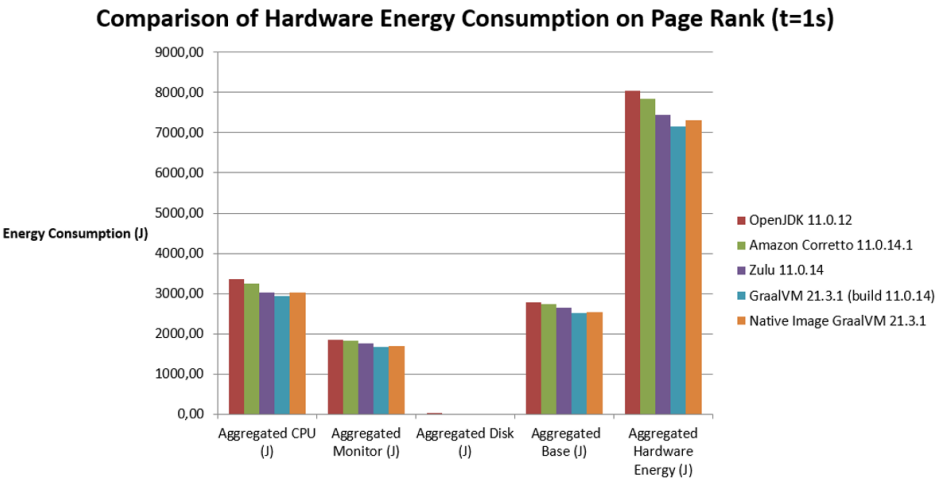


Figure 31. Comparison of hardware energy consumption on page rank



programs. Moreover, future work can also focus on other performance factors of the JVM such as VM startup time, heap size, and stack size to provide better optimizations for all JVM distributions.

ACKNOWLEDGEMENT

This research and the APC are funded by European Commission grant number 610619-EPP-1-2019-1-FR- EPPKA1-JMD-MOB (EMJMD Genial Project).

REFERENCES

Arushanyan, Y., Ekener-Petersen, E., & Finnveden, G. (2014). Lessons learned – Review of LCAs for ICT products and services. *Computers in Industry*, *65*(2), 211–234. doi:10.1016/j.compind.2013.10.003

Avom, D., Nkengfack, H., Fotio, H. K., & Totouom, A. (2020). ICT and environmental quality in sub-Saharan Africa: Effects and transmission channels. *Technological Forecasting and Social Change*, *155*, 120028. doi:10.1016/j.techfore.2020.120028

Bastida, L., Cohen, J. J., Kollmann, A., Moya, A., & Reichl, J. (2019). Exploring the role of ICT on household behavioural energy efficiency to mitigate global warming. *Renewable & Sustainable Energy Reviews*, *103*, 455–462. doi:10.1016/j.rser.2019.01.004

Bourdon, A., Noureddine, A., Rouvoy, R., & Seinturier, L. (2013). PowerAPI: A Software Library to Monitor the Energy Consumed at the Process-Level. *ERCIM News*, *2013*(92).

Deitel, H., & Deitel, P. (2020). Java how to program (late objects) (11th edition). Pearson UK.

Department for Business. Energy & Industrial Strategy. (2021*). Greenhouse Gas Reporting: Conversion Factors 2021*. Gov.uk. https://www.gov.uk/government/publications/greenhouse-gas-reporting-conversion-factors-2021/

Ergasheva, S., Khomyakov, I., Kruglov, A., & Succil, G. (2020). Metrics of energy consumption in software systems: A systematic literature review. *IOP Conference Series. Earth and Environmental Science*, *431*(1), 012051. Advance online publication. doi:10.1088/1755-1315/431/1/012051

Evans, B. (2015). *Java, the legend: Past, present, and future*. O'Reilly Media.

Floyer, D. (2020). *Java for mid-sized enterprises: On-premises & in the cloud.* Oracle.com. https://www.oracle.com/de/a/ocom/resources/java_for_mid-size_enterprises.pdf

GeeksForGeeks. (2019). *Disadvantages of Java language*. Geeks for Geeks.com. https://www.geeksforgeeks.org/disadvantages-of-java-language/

Gilliard, M. (2020). *Using SDKMAN! to work with multiple versions of Java*. Twilio.com. https://www.twilio.com/blog/sdkman-work-with-multiple-versions-java

Gosling, J., Joy, B., Steele, G., Bracha, G., Buckley, A., & Smith, D. (2018). The Java language specification: Java SE (11th edition). Addison-Wesley Professional.

Graalvm.org. (n. d.). GraalVM architecture overview. https://www.graalvm.org/22.0/docs/introduction/

Houghton, J. (2009). ICT and the environment in developing countries: An overview of opportunities and developments. *Communications & Stratégies*, *1*, 39–60.

Ikedilo, O., Osisikankwu, P., & Madubuike, C. (2015). A critical evaluation of Java as a good choice for introductory course. *International Journal of Research*, *2*(12), 847–853.

ITU-T, International Telecommunication Union. (2012). Methodology for the assessment of the environmental impact of information and communication technology goods, networks and services. https://www.itu.int/rec/T-REC-L.1410

Kansal, A., Zhao, F., Liu, J., Kothari, N., & Bhattacharya, A. (2009). *Joulemeter: Virtual machine power measurement and management.* https://www.microsoft.com/en-us/research/publication/joulemeter-virtual-machine-power-measurement-and-management/

Kor, A. L., Pattinson, C., Imam, I., AlSaleemi, I., & Omotosho, O. (2015). Applications, energy consumption, and measurement. In *2015 International Conference on Information and Digital Technologies*, (pp. 161-171). IEEE. doi:10.1109/DT.2015.7222967

Kumar, A. (2021a). *Supercharge your applications with GraalVM: Hands-on examples to optimize and extend your code using GraalVM's high performance and polyglot capabilities*. Packt Publishing.

Kumar, A. (2021b). *GraalVM — Episode 2: The Holy Grail*. Faun.pub. https://faun.pub/episode-2-the-holy-grail-graalvm-building-super-optimum-microservices-architecture-series-c068b72735a1

Lestal, J. (2020, August 5). *History of programming languages*. DevSkiller.com. https://devskiller.com/history-of-programming-languages/

Long, D. (2017). *Programming languages' milestones: An overview from 1960 - present (the last part).* Viblo.asia. https://viblo.asia/p/programming-languages-milestones-an-overview-from-1960-present-the-last-part-924lJr7XlPM

Mahdavi, S., & Sojoodi, S. (2021). Impact of ICT on environment. 10.21203/rs.3.rs-1020622/v1

Morales, A. (2019). Meet the team that built GraalVM, an energy-saving multilingual compiler written entirely in Java. *Forbes.com*. https://www.forbes.com/sites/oracle/2019/05/08/meet-the-team-that-built-graalvm-an-energy-saving-multilingual-compiler-written-entirely-in-java/?sh=1a4425784ee6

Noureddine, A., Rouvoy, R., & Seinturier, L. (2013). A review of energy measurement approaches. *Operating Systems Review*, *47*(3), 42–49. doi:10.1145/2553070.2553077

Ournani, Z., Belgaid, M. C., Rouvoy, R., Rust, P., & Penhoat, J. (2021). Evaluating the impact of Java virtual machines on energy consumption. In *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)* (pp. 1-11). IEEE. doi:10.1145/3475716.3475774

Prokopec, A., Rosà, A., Leopoldseder, D., Duboscq, G., Tuþma, P., Studener, M., Bulej, L., Zheng, Y., Villazón, A., Simon, D., Wu¨rthinger, T., & Binder, W. (2019). Renaissance: Benchmarking suite for parallel applications on the JVM. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (pp. 31-47). IEEE. doi:10.1145/3314221.3314637

Renaissance. (n. d.). *Renaissance-benchmarks/renaissance: The Renaissance benchmark suite Architecture Overview.* Github.com. https://github.com/renaissance-benchmarks/renaissance

Sehgal, R., Mehrotra, D., Nagpal, R., & Sharma, R. (2022). Green software: Refactoring approach. *Journal of King Saud University - Computer and Information Sciences, 34*(7), 4635-4643. 10.1016/j.jksuci.2020.10.022

Singh, B., & Gupta, G. (2019). *Analyzing windows subsystem for Linux metadata to detect timestamp forgery*, 159-182.

Sipek, M., Muharemagic, D., Mihaljevic, B., & Radovan, A. (2020). Enhancing performance of cloud-based software applications with GraalVM and Quarkus. In *2020 43rd International Convention on Information, Communication and Electronic Technology (MIPRO)* (pp. 1746-1751). IEEE. doi:10.23919/MIPRO48935.2020.9245290

Stephens, A., & Didden, M. (2013). The development of ICT sector guidance: Rationale, development and outcomes. In *ICT4S 2013: Proceedings of the First International Conference on Information and Communication Technologies for Sustainability, ETH Zurich*, (pp. 8-11).

TIOBE Software. (2023). *TIOBE index for February 2023*. https://www.tiobe.com/tiobe-index/

United Nations. (2021). *Technology and innovation report 2021: United Nations Conference on Trade and Development (UNCTAD) Technology and Innovation Report (TIR).* UN. https://www.un-ilibrary.org/content/books/9789210056588

Vermeer, B. (2020). *JVM ecosystem report 2020.* Snyk.io. https://snyk.io/wp-content/uploads/jvm_2020.pdf

Thalita Vergilio is a Senior Lecturer and independent researcher at Leeds Beckett University. With a strong industry background of over 10 years in software engineering, her research interests include stream big data frameworks, containers and container orchestration technology, web development best practices, systems architecture, and DevOps practices.

Long is a sponsored graduate of the Erasmus Mundus Joint Master's Program GENIAL. With a solid background of over five years in Information Technology and Software Engineering, Long is interested in big data processing, high throughput distributed systems, container technologies and cloud computing.

Ah-Lian Kor is a Professor in Sustainable and Intelligent Computing at Leeds Beckett University. She has been involved in several EU projects for Green Computing, Innovative Training Model for Social Enterprises Professional Qualifications, and Integrated System for Learning and Education Services. She has published work on Ontology, Semantics Web, Web Services, Portal and semantics for GIS.