Mittuniversitetet
MID SWEDEN UNIVERSITY

Performance comparison of GraalVM, Oracle JDK and OpenJDK for optimization of test suite execution time

Fredric Fong, Mustafa Raed

Final Project
Main field of study: Computer Engineering BA (C)
Credits: 15
Semester/Year: VT 2021
Supervisor: Ali Hassan Sodhro
Examiner: Felix Dobslaw
Course code/registration number: DT133G
Degree programme: Programvaruteknik

Performance comparison of GraalVM, Oracle JDK and OpenJDK for optimization of test suite execution time

Fredric Fong, Mustafa Raed
BSc. Thesis, Programvaruteknik
Institute of Computer and Systems Sciences
Mid Sweden University
Östersund, Sweden
{frfo0300, mura1601}@student.miun.se

Abstract **-** Testing, when done correctly, is an important part of software development since it is a measure of the quality of a software in question. Most of the highly rated software projects therefore have test suites implemented that include unit tests, integration tests, and other types of tests. However, a challenge regarding the test suite is that it needs to run each time new code changes are proposed. From the developer's perspective, it might not always be necessary to run the whole test suite for small code changes. Previous studies have tried to tackle this problem e.g., by only running a subset of the test suite. This research investigates running the whole test suite of Java projects faster, by testing the Java Development Kits (JDKs) GraalVM Enterprise Edition (EE) and Community Edition (CE) against Oracle JDK and OpenJDK for Java 8 and 11. The research used the test suite execution time as a metric to compare the JDKs. Another metric that was considered was the test suites number of test cases, used to try and find a breaking point for when GraalVM becomes beneficial. The tests were performed on two test machines, where the first used 20 out of 48 tested projects and the second used 11 out of 43 projects tested. When looking at the average of five runs, GraalVM EE 11 performed best in 11 out of 18 projects on the first test machine, compared to its closest competitor, and in 7 out of 11 projects on the second test machine both for JDK 8 and 11. However GraalVM EE 8 did not give any benefits to the first test machine compared to its competitors, which might indicate that the hardware plays a vital role in the performance of GraalVM EE 8. Number of test cases could not be used to determine a breaking point for when GraalVM is beneficial, but it was observed that GraalVM did not show any benefits for projects with an execution time of fewer than 39 seconds. It is observed that GraalVM CE, does not perform well as compared to the other JDKs, and in all cases, its performance is not countable due to less non-satisfied and inefficient behavior.

Index Terms – *GraalVM; Software Testing; Test Optimization; Test Performance; Java.*

I. Introduction

Software companies want to shorten their time to market for obtaining a competitive edge, and to achieve this, many of them have introduced the Continuous Integration/Continuous Delivery (CI/CD) pipeline. As part of the CI/CD pipeline is the test suite which for most software runs each time new code is proposed. When this happens, the whole test suite is executed, which might take a long time, and it might be unnecessary for small code changes. Therefore, some researchers have proposed frameworks that reduce test execution time by only running the most important tests (test case selection), chosen by their test code coverage [1]. Other researchers have proposed to prioritize the tests that have the highest test failure history (history-based selection) [2]. And yet other researchers have proposed an approach based on reinforcement learning and neural networks, where the method learns from its experience of the execution environment [3]. What the mentioned approaches have in common is that they all focus on reducing the execution time of running the test suite by only considering a subset of it.

Instead of looking at test selection, this research will examine whether it is possible to run the full test suite faster. Some previous studies have tried to do this by reducing the number of instructions and cache misses [4]–[7], however this research will have a look at whether new high-performance runtime GraalVM [8] can improve the performance of running the whole test suite. GraalVM is a JDK that can run Java and other languages that can run on the Java Virtual Machine (JVM), e.g., Clojure. Other features of GraalVM are to create native images through its Ahead-Of-Time (AOT) compilation, and its polyglot support which makes it able to compile multiple different languages in the same project with the truffle framework.

Different research conducted on GraalVM has shown that it outperforms other JDK in terms of performance [9]–[12]. However, the focus of that research has not been to improve the performance of running a test suite. Some of the studies focus more on the performance of GraalVM native images, which uses AOT compilation [11], [12]. Other studies [9], [10] are doing research on the DaCapo benchmark suite, while this research will broaden the scope by performing the measurements on a collection of open-source projects.

The contribution of this work is thus threefold, first, it will show whether, and by how much, GraalVM will improve the performance of executing the test suite of open-source projects. Second, to explain in what context GraalVM would or would not improve the performance of running the test suite. Lastly, it will show whether any drawbacks or compatibility issues are encountered by adopting GraalVM compared to the other JDKs.

The rest of the paper is divided into the following sections: section II explains the problem statement, section III explains the background section, section IV shows the related work, section V explains the research methodology, section VI explains the artifact that has been used to perform the test measurement, section VII shows the results, section VIII shows the discussion and section IX concludes the paper.

II. Problem Statement

Testing in the software development life cycle is important. Past research has tried to address the problem with the growing complexity of modern software and the number of test cases needed for effective validation [1]–[3], [13]. Running large test suites is both demanding in terms of time and energy consumption. This results in increased costs and is not ethically sustainable for the environment. In this study, we aim to address the growing cost and requirement of running more tests by evaluating new technology that could speed up the verification step of software development. GraalVM is a new high-performance runtime that has shown promising performance gains in past research for applications [9]–[12], but none have studied or presented the impact on the test performance.

In our research, we want to investigate the following research questions (RQs):

RQ1: Could we speed up test execution by using GraalVM? If so by what degree?

RQ2: Is there a (or what is the) breaking point for which GraalVM does (not) give any benefits?

RQ3: Are there any drawbacks or compatibility issues with GraalVM on open-source projects compared to other JDKs?

III. Background

GraalVM is a JDK which provides a complete package of tools for developing and running Java-based software [8]. The JDK is a portable Java development environment which makes it possible to develop and run Java applications. Main components of the JDK are the JVM, the Java interpreter, the *javac* compiler, and the Java archiver (JAR) [14], [15]. The JVM is a subset of the JDK, and it is what makes Java able to run on different platforms. The JVM is ported to the different platforms and executes the Java bytecode that has been compiled by the *javac* compiler. The JVM is thus responsible for loading and running the Java programs [15], [16].

GraalVM started out as a research project at Oracle and has since then evolved into a commercial product. The first production-ready version was officially announced and released in 2019 [17]. Some of the key features that it states to provide are high performance by generating fast and lean code, executable native binaries, and polyglot support [18]. Twitter was one of the companies that quickly adopted their *Scala*[1] based services to use GraalVM in production and stated a significant performance gain in using it, thus reducing the cost and being more environmentally friendly by reducing the number of machines required in their data centers [19]. GraalVM comes in two flavors, CE and EE. The EE has some performance optimization which the CE does not, for example auto-vectorization which makes it possible to analyze loops in code and run them faster by executing them using the target machines vector registers and instructions [20]–[22].

Two compilation technologies[2] that GraalVM provides are Ahead-Of-Time (AOT) and Just-In-Time (JIT) compilation. The AOT compiler in GraalVM provides a way to compile the code beforehand into a standalone natively executable called *native image*[3]. The intention with AOT and native images is to improve the warm-up period of applications as libraries and dependencies are already provided and loaded ahead of time instead of at runtime. This makes Java application start instantaneously compared to the same Java application compiled JIT. The JIT compiler compared to the AOT compiler in GraalVM compiles the code at runtime and does this dynamically. The code is continuously analyzed during runtime and optimized thereafter. The reason for compiling the code at runtime is to have the code platform independent. In Java, the code is converted into byte code and at execution in JVM it is compiled into machine code that is platform dependent.

[1] https://www.scala-lang.org/

[2] https://www.graalvm.org/reference-manual/compiler/

[3] https://www.graalvm.org/reference-manual/native-image/

Part of the performance gains of GraalVM comes from a more aggressive compiler analysis and optimization to remove costly object allocations. Graal which is the compiler of GraalVM uses the *Partial Escape Analysis* method to optimize its JIT compilation [23]. *Escape Analysis* is a way for compilers to determine the dynamic scope and the lifetime of allocated objects. The compiler can with this information perform different optimization techniques on operations. Partial Escape Analysis is a new way of performing Escape Analysis. The main difference is that it performs flow-sensitive analysis which means that branches are analyzed individually and can later be optimized with the *Scalar Replacement* [24] optimization technique to improve the performance.

Maven and Gradle are two widely used build tools for Java. The execution of tests is standardized and uniformed which makes it suitable to limit to projects that utilizes either of these tools in this research. Maven is used for building and managing Java-based projects. The purpose of Maven is to provide a uniform build system that makes the build process easy. Maven builds a project using a Project Object Model (POM) file and a set of plugins. Maven can also provide information about the project such as change logs, information about the dependencies used in the project, and unit test report and coverage [25]. While Maven is used for building Java-based projects, Gradle is a build automation tool that is used for building almost any type of software project. The core model of Gradle is built on tasks, which are units of work. Gradle build lifecycle consists of three phases: initialization, configuration, and execution. The initialization phase sets up the build environment and determines the projects that will take part in the build. The configuration phase creates the task graph for the build and sorts the tasks in the order they will be run. The execution phase runs the tasks [26].

Locality of reference in computer science is the tendency of a software program to access recently used memory or memory with proximity. The phenomena of locality of reference have a big impact on the software performance and a component where this occurs is in the cache. A cache is used to store data that will be used for future requests. A *cache miss* is when the data is not found in the faster cache memory and needs to be fetched from a slower part of the system, thus impacting the performance.

IV. Related Work

This section presents the related work of this study. First, we present related research that has been done in test optimization. Finally, we present related research work that has contributed to or evaluated GraalVM from a performance perspective.

A. Test Compilation and Execution Optimization

Many studies that try to address the growing and costly test execution step have mainly been using test optimization techniques focusing on the reduction of the number of test cases needed to be executed with the potential drawback of reducing the test scope [1]–[3], [13]. Reduction techniques such as test case prioritization and smarter test case selection based on failure rate history are examples of proposed methods. Stratis et al. have in recent studies [4]–[7] presented research that tries to address this from a different perspective than reducing or modifying the scope by proposing ways of optimizing the test suite compilation and test suite execution time. They proposed a data transformation method to reduce the number of instructions in test code and thus improving the test compilation time. For optimizing the executions of test cases, a way of measuring the different instructions between test cases was proposed to reduce cache misses. The proposed method means that you can keep larger test scope as test execution is faster and can as well be used in combination with the reduction techniques available. In relation to these studies, we study GraalVM as a way of speeding up test compilation and execution time.

B. GraalVM Performance

The Graal compiler uses Partial Escape Analysis to optimize its JIT code compilation. The first paper that presented this new method of Escape Analysis was in a research paper from 2014 by Stadler et al. [23] where this algorithm was implemented on top of the Graal open-source JIT compiler at that time. A performance comparison with Java HotSpot JIT compiler was done by performing measurements with a variety of open-source benchmarks such as the DaCapo [27] benchmark. The results showed that Graal compiled code in some situations consumed up to 55% less memory and improved the performance by up to 33%.

The first production-ready version of GraalVM was announced and released in May 2019. The same year Spicek et al. [10] presented a study of the architecture and the basic features and functionality of GraalVM. The same research also performed an initial evaluation of its performance using DaCapo benchmarking. The result showed that although GraalVM was still under development it had a bright future ahead with its polyglot support, Ahead-Of-Time and Just-In-Time compilation strategies, and its aggressive compilation optimizations.

GraalVM was further examined by Spicek et al. [11] in 2020 whether it could enhance the performance of cloud-based software applications. The researchers performed the test on 4 benchmarks from the *Renaissance*[4] benchmark suite, and by testing a self-created REST API with a product database. They ran the test with OpenJDK 8, 13, and GraalVM CE. The metrics they focused on were the performance and the

[4] https://renaissance.dev/

memory usage (Resident Set Size (RSS)). The outtakes from this study were that GraalVM CE performed the best and used the least memory RSS. Contrary to this research, Spicek et al. focused on GraalVM native images and AOT compilation, while this research will focus on the JIT compiler. This research will also extend the JDKs that will be tested, including Oracle JDK 8, 11, and GraalVM EE.

In 2019 several Oracle developers performed different tests and explained the concepts of GraalVM native images and AOT compilation [12]. They conducted the research by measuring application startup performance and memory footprint of 'Hello World' projects (written in C and GO), case studies, and Java microservice frameworks (Quarkos, Micronaut, and Helidon). They performed the tests with GraalVM EE and compared the results to those of Java Hotspot VM with JDK 8 and 12. The research concludes that GraalVM EE native image has better startup performance and the same peak performance as JDK 8 Java Hotspot VM. GraalVM EE also makes it possible to have a better startup performance than GO and Javascript V8 VM. Like [11] the contradiction to this research is that the focus of [12] is on native images and AOT compilation and that our study will test an extended list of JDKs.

The two compilation strategies AOT and JIT were studied in 2020 by Björk [28] where it was evaluated by performing benchmarking of serverless functions written in Kotlin language in a cloud-based environment. The benchmarking suites were a selection of test suites from different open-source benchmark suites. The AOT compiler of the GraalVM was matched against the JIT compiler of the Java Hotspot VM. This work specifically studied the AOT aspects of GraalVMs compiler infrastructure and in this case executable native images. The study indicated that AOT compiled code results in faster total runtimes for serverless functions that needed cold starts and had less memory requirements to run. This study had an influence on our work of selecting the type of compiler strategy to use. Although AOT showed faster total runtimes, it has drawbacks such as longer compilation time and no further optimization at runtime. Contrary to this study we will only evaluate the JIT part of GraalVM.

Additional study of GraalVMs performance was done the same year by Larsson [9] where the performance of GraalVM for Java programs was evaluated with the aspect of the JIT compilation optimization. The study showed performance gains for the commercial versions of GraalVM compared to other JDKs, but it also showed a high variation of the results where GraalVM was not the top performer in all cases. In contrast to this study, we solely focus on evaluating the test performance of GraalVM and if the test execution can be improved.

V. Research Methodology

In this section, we describe the projects selection step where criteria were defined and the reasoning around it. We also describe the metrics, data collection and the analysis performed. A visualization of the steps is presented in Figure 1 and Figure 2. Figure 1 shows a high level overview of the method steps in this study whereas Figure 2 is an in-depth illustration of all of the steps.

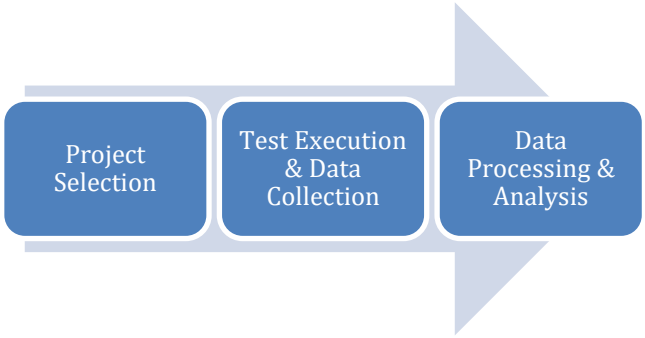


Figure 1. Research methodology step overview. First, the projects were selected, then the test performed and data collected from log files, and finally, the data was processed and analyzed, and were represented as graphs/tables.

A. Project Selection

To conduct a test performance measurement between GraalVM and different JDKs, a set of suitable projects were needed to be identified as test objects. GitHub is well known in the software industry for having the world's largest collection of open-source software projects. To get a good representation of the community and the current state, popular modern open-source projects, both small and large with a mixed number of test cases were considered. The number of forks and stars were used as metric for a project's usage and popularity. A great number of forks and stars indicates that the project is widely used, popular and appreciated in the community [29], [30].

Projects were selected based on the following criteria:

- **Project must use Java as programming language.**
  GraalVM is polyglot capable, but other programming languages are not in the scope of this study.
- **Project must have a minimum of 1000 stars and 150 forks.**
  This ensures that we have most popular projects.
- **Project must be compatible with Java 8 or 11.**
  This is the current support and the limitation of GraalVM [8]. Java JDK 11 version are normally compatible with Java 8 projects, but it is not

guaranteed as some incompatible changes were introduced compared to earlier versions[5].

- **Maven or Gradle as build system**.
  This is a limitation of our current tooling infrastructure.
- **Must have a test suite implemented.**
  This to be able to perform test execution measurements.
- **Projects must build successfully or fail at the same step**
  To be able to compare the results between the different JDK.

The base dataset[6] used originates from a study on factors that impacts popularity of GitHub repositories [31]. This study presented a manual classification of the application domain of the 5,000 most popular GitHub repositories (high number of stars and/or forks). Each project was classified in one of the following six categories: *Application software*, *System software*, *Web libraries and frameworks*, *Non-web libraries and frameworks*, *Software tools* and *Documentation*. By using this dataset instead of sourcing directly from GitHub enables the possibility to select projects that represents several different types of software domains.

The listed criteria were applied to this dataset of open-source projects. Filtering the first source over the most popular GitHub projects by Java resulted in 521 projects to choose from. From these projects, further filtering was done manually, projects were selected randomly and from different categories provided that the projects fulfilled the specified criteria. Criteria such as compatibility with Java 8 or 11, and the number of test cases are hard to identify without cloning the repository and executing the test suites. Some projects did state the required Java version on the GitHub project overview page documentation.

B. Metrics

The metrics used in this study were the *execution time*, *total execution time*, and *the number of test cases*. The execution times were used to try to answer whether GraalVM could improve the performance of running test suites (RQ1). The execution time was defined as the time it takes to compile and run a project's test suite. The reason why it was defined as both the compile time and the execution time of the test suite was because that is what would happen when running the test suite in a real-life scenario. When a developer develops new code and the CI/CD process is started, the build automation tool would compile and build the project and then execute the test suite. The second metric *total execution time* was calculated by summing all execution times of all projects for a specific JDK to show how it performs overall for a set of projects of different types in comparison to each other.

The metric *the number of test cases* was used in correlation with the *execution time*. The intention with the number of test cases in correlation with execution time was to find and decide on the dividing line where GraalVM does not improve the test suite execution (RQ2). The hope was to find approximately what number of test cases GraalVM does not give any more improvements in the execution time. The *number of test cases* however does not indicate the complexity of the test suite. Another metric that might have been a better metric would have been the *lines of test code,* however, with the number of test suites that were going to be tested in this research, the *lines of test code* were out of scope.

The mean time and the standard deviation of all projects were also calculated and presented. The following equations were used for calculating the mean and standard deviation values.

The *sample mean* $\bar{x}$ equation:

$$\bar{x} = \frac{\sum_{i=1}^{n} x_i}{n} \quad (1)$$

The *standard deviation* s equation:

$$s = \sqrt{\frac{\sum_{i=1}^{n} (x_i - \bar{x})^2}{n-1}} \quad (2)$$

C. Performing Measurement and Data Collection

The performance measurement was performed by cloning and running the test suite of each project with each Java runtime. Maven and Gradle projects have a general way of starting test suites by running commands *mvn test* and *gradlew test*, respectively. It was decided to run each runtime 6 times on each project to see the span of variation in the test execution time. The first run was however excluded in the results since this may include the download of various project dependencies. All subsequent runs were performing a clean (*mvn clean* or *gradlew clean*) which does not remove these dependencies and should thus not impact the deviation.

Table 1 shows the Java runtimes that were tested in this research and their Java/JRE versions. The decision of choosing these versions relied on the fact that GraalVM EE and CE are based on either Java 8 or 11. It was decided to choose the latest GraalVM EE/CE which has version 21.0.0.2 to be able to test the most up to date runtime. The choice of the OpenJDK and Oracle JDK versions was then based on the Java versions which the GraalVM runtimes are based on.

[5] https://www.oracle.com/java/technologies/javase/jdk11-readme.html

[6] http://doi.org/10.5281/zenodo.804474

Table 1. Java runtimes and their Java/JRE versions.

Name	Java version	JRE version
GraalVM EE 21.0.0.2 (11)	Java version "11.0.10" 2021-01-19 LTS	Java(TM) SE Runtime Environment GraalVM EE 21.0.0.2 (build 11.0.10+8-LTS-jvmci-21.0-b06)
GraalVM CE 21.0.0.2 (11)	openjdk version "11.0.10" 2021-01-19	OpenJDK Runtime Environment GraalVM CE 21.0.0.2 (build 11.0.10+8-jvmci-21.0-b06)
Oracle JDK 11.0.10	java version "11.0.10" 2021-01-19 LTS	Java(TM) SE Runtime Environment 18.9 (build 11.0.10+8-LTS-162)
OpenJDK 11.0.10	openjdk version "11.0.10" 2021-01-19	OpenJDK Runtime Environment (build 11.0.10+9-Ubuntu-0ubuntu1.20.04)
GraalVM EE 21.0.0.2 (8)	java version "1.8.0_281"	Java(TM) SE Runtime Environment (build 1.8.0_281-b09)
GraalVM CE 21.0.0.2 (8)	openjdk version "1.8.0_282"	OpenJDK Runtime Environment (build 1.8.0_282-b07)
Oracle JDK 1.8.0_281	java version "1.8.0_281"	Java(TM) SE Runtime Environment (build 1.8.0_281-b09)
OpenJDK 1.8.0_282	openjdk version "1.8.0_282"	OpenJDK Runtime Environment (build 1.8.0_282-8u282-b08-0ubuntu1~20.04-b08)

The test was performed on two test machines and Table 2 lists their characteristics and software. Test Machine 1 (TM1) ran Ubuntu operating system natively, while Test Machine 2 (TM2) ran Ubuntu in a Virtual Machine instance in Windows.

Table 2. Characteristics of test machines.

Test Machine 1 (TM1)	Test Machine 2 (TM2)
Processor: AMD A6-7310 APU, 4-cores and 4-threads, 2.0GHz RAM: 8 GB DDR3 OS: Ubuntu 20.04.2 LTS 64-bit	Processor: AMD Ryzen 5 3600, 6-cores and 12-threads, 3.6GHz RAM: 32 GB DDR4 OS: Windows 10 Professional 64-bit Oracle VirtualBox VM: 5C/10T, 8GB VM OS: Ubuntu 20.04.2 LTS 64-bit

D. Data Analysis

The artifact outputs log information of the test suite execution for each project in the test logs folder. This information includes the execution time of the test suite and the number of test cases. Our metrics were thus extracted from the log files and placed into spreadsheets. The results were plotted in tables, and bar graphs including the standard deviation which represents the variations in test execution time.

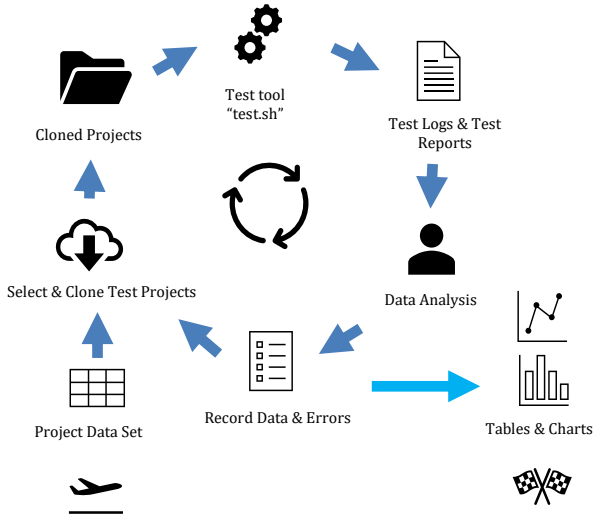


Figure 2. Detailed flow chart of the research methodology. The flowchart shows the test was performed iteratively. First, the projects were selected, then the test process started, obtained data analyzed, and based on the results the projects might be refined and the process re-executes.

VI. Artifacts

The purpose of the artifact[7] illustrated in Figure 3, is to run Maven and Gradle projects with different runtimes. The main components of the artifact are the *clone-projects.sh* and the *test.sh* scripts. The *clone-projects.sh* script:

1. Reads the projects URLs from files.
2. Clones each project to a folder using Git.

The *test.sh* script takes over the result of the *clone-projects.sh* script and:

3. Runs the test suite of each project in the cloned-projects folder, with every runtime in the JDKS folder.
4. Outputs test execution logs to the terminal and to logfiles in the test-logs folder.

[7] https://github.com/mustafaRaed/miun-jdk-test-tool

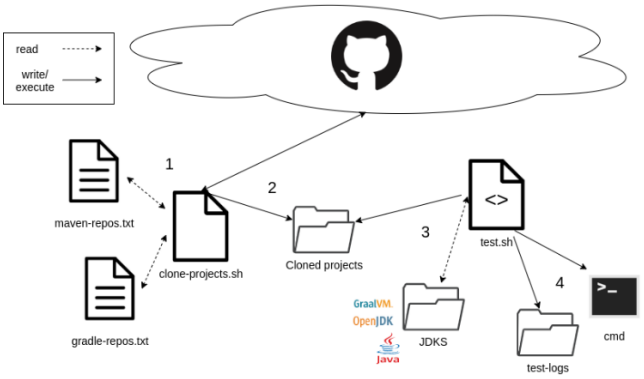


Figure 3. Test tool diagram. (1) 'clone-projects.sh' reads Git URLs from the files 'maven-repos.txt' and 'gradle-repos.txt', and clones the projects (2). 'test.sh' sets the JDK that should run the cloned projects test suites (3), starts the test, and outputs log to file and cmd (4).

VII. Results

A total number of 42 projects on TM2 and 48 projects on TM1 were chosen at the initial project selection step. They were all downloaded and tested to see if they fulfilled the requirements and if there were any complications during the test suite execution step. For TM2, 28 projects out of the 42 initial project selections were excluded as they failed with build error or exception and for TM1, the same number of projects were excluded from advancing any further in the process. Figure 4 illustrates the failure and success rate of the initial project selection step.

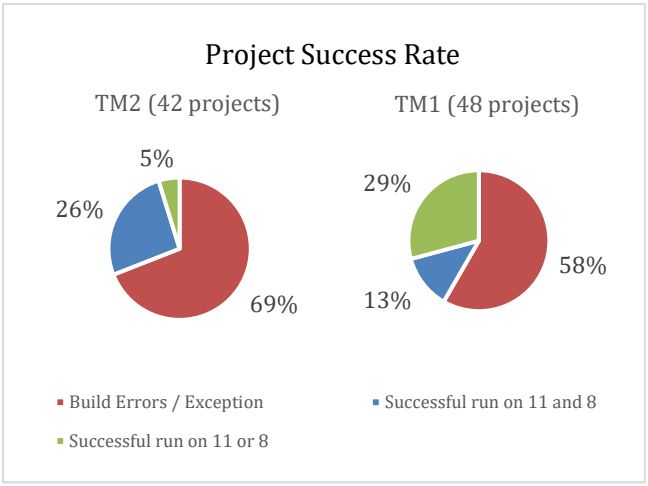


Figure 4. Project build and test suite success rate for both TM2 and TM1.

A. Projects Included in Measurement Data

Table 3 lists the total number of projects that fulfilled our criteria, and which also successfully ran test suites without halting on build exceptions or errors. The table shows a total number of 22 projects and their characteristics which are the project name, the build system used, the compatible Java version for the project, the number of test cases that were executed by the build script, the type of software category classification, and the test machine that the measurement was performed on.

Table 3. List of selected open-source projects for measurement in this study and their characteristics.

Project	Build System	Comp. Java 8/11	#Test Cases	Software Category	Tested on TM
reactiveX/ rxJava	gradle	8	13044	System software	1,2
google/ guava	maven	8	858390	Non-web libraries and frameworks	1,2
zxing/ zxing	maven	8	474	Non-web libraries and frameworks	1,2
clojure/ clojure	maven	8	651	System software	1,2
junit-team/ junit4	maven	8	1108	Non-web libraries and frameworks	1,2
eclipse-vertx/ vert.x	maven	8	4576	System software	1,2
openzipkin/ zipkin	maven	8	1055	Software tools	2
google/ guice	maven	8	7200	Non-web libraries and frameworks	1,2
mybatis/ mybatis-3	maven	8	1675	Non-web libraries and frameworks	1,2
code4craft/ webmagic	maven	8	94	Web libraries and frameworks	1,2
Atmosphere/ atmosphere	maven	8	264	System software	1,2
iluwatar/ java-design-patterns	maven	11	4665	Documentation	1,2*
dropwizard/ dropwizard	maven	8	2016	Web libraries and frameworks	1,2*
Bukkit/ Bukkit	maven	8		Web libraries and frameworks	1
jhy/ jsoup	maven	8		Non-web libraries and frameworks	1
shwenzhang/ AndResGuard	gradle	8		Software tools	1
dropwizard/metrics	maven	8		Software tools	1
flyway/ flyway	maven	11		Software tools	1
square/ javapoet	maven	8		Software tools	1
spring-projects/ spring-mvc-showcase	maven	8		Documentation	1
eclipse/ che	maven	11		Software tools	1
google/ error-prone	maven	8		Software tools	1

NOTE: The * denotes that the project was not included in the comparison for TM2.

B. Total Mean Test Execution Time

This section presents the total mean test execution time measurement performed on the two machines. Firstly, we present the results from TM2 and finally the results from TM1.

Test Machine 2

A total of 11 projects that ran successfully for all JDKs on TM2 were chosen for the calculation of the total mean test execution time and the standard deviation. Figure 5 illustrates

the total execution time of all selected projects for Java 8 & 11 JDKs that were tested on TM2. The staples represent the mean time in minutes and seconds in the format *mm:ss* with standard deviation bars. The result indicates that GraalVM EE was the fastest overall performer among the Java 11 JDKs with 1.09% faster than the runner-up Oracle JDK, 2.60% faster than GraalVM CE, and finally 2.69% faster than the slowest performer OpenJDK. For Java 8 JDKs the GraalVM EE was also the best performer with a greater impact as GraalVM EE is ahead of Oracle JDK with 2.17%, and 2.94% faster than GraalVM CE, finally 3.04% faster than the OpenJDK.

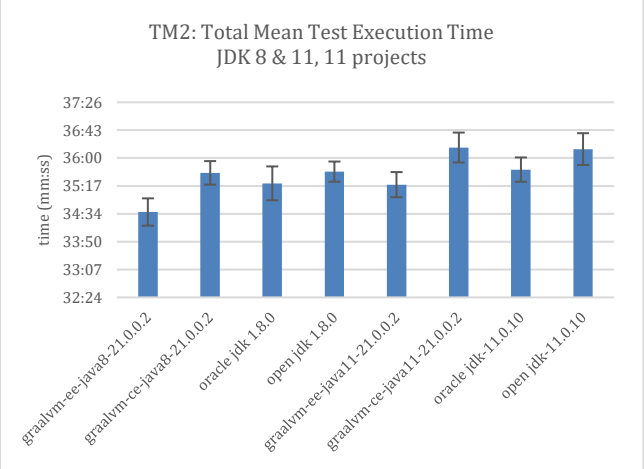


Figure 5. Total mean test execution time chart for Java 8 & 11 JDKs (TM2). Shows that GraalVM EE with Java 8 performs best overall, followed by GraalVM EE with Java 11.

Test Machine 1

A total of 20 projects on TM1 that, either ran successfully[8] on all JDKs or one Java version, were chosen for the calculation of the total mean test execution time and the standard deviation. Figure 6 shows the results obtained from TM1 for the six projects that were successful for all JDKs. The results also showed that GraalVM EE was the best performer for Java 11 JDK with 3.73% faster than the runner up Oracle JDK 11, 4.68% faster than OpenJDK, and 5.09% faster than GraalVM CE 11. However, TM1 gave different results for JDK 8 than TM2. GraalVM CE was the slowest performer, and GraalVM EE was runner up, while OpenJDK performed the best. GraalVM EE was 12.57% faster than GraalVM CE, but 6.66% slower than Oracle JDK and 7.21% slower than OpenJDK.

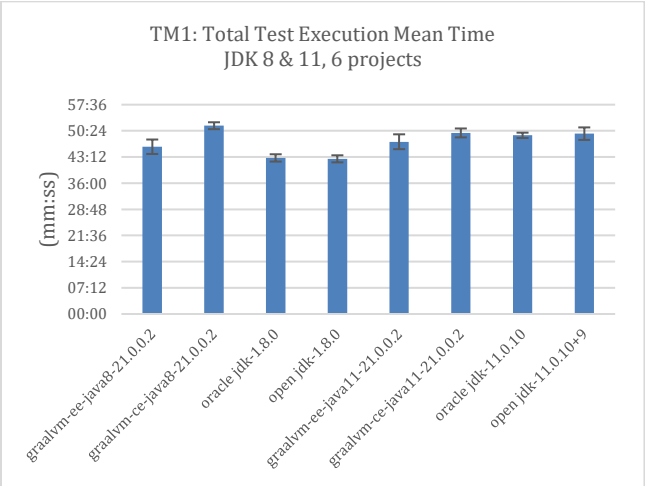


Figure 6. Total mean test execution time chart for Java 8 & 11 JDKs (TM1). Oracle JDK and OpenJDK perform the best overall, while GraalVM EE with Java 11 performs best for the Java 11 JDKs.

Figure 7 takes a closer look at the eight projects that were successful for the Java 8 JDKs, and Figure 8 takes a closer look at the 18 projects that were successful on the Java 11 JDKs on TM1. Both graphs give a similar picture to that of Figure 5 but for a greater number of projects.

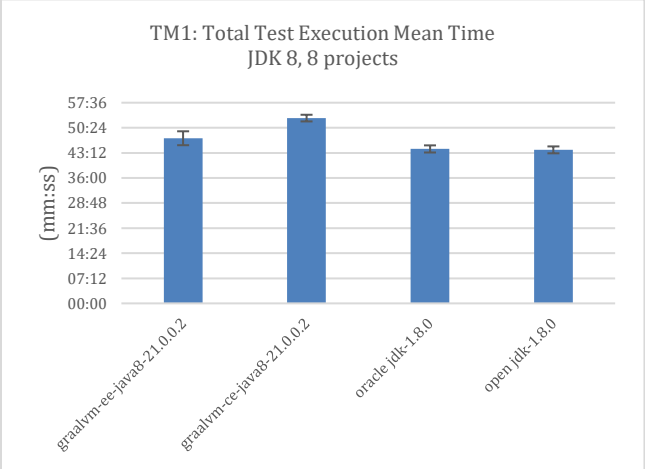


Figure 7. Total mean test execution time chart for Java 8 JDKs (TM1). The figure summarizes the results of 8 projects and still shows that Oracle JDK and OpenJDK performed the best on average.

Looking at the 18 projects that were successful for JDK 11, GraalVM is the best performer, followed by Oracle JDK, closely followed by GraalVM CE, and lastly OpenJDK. Here, GraalVM EE was on average 2.99% faster than Oracle JDK, 3.02% faster than GraalVM CE, and 3.50% faster than OpenJDK. Compared to Figure 6 with the six projects, GraalVM CE performed better here than OpenJDK.

[8] Successfully in this context is considered as either successfully built, or failing at the same step, since in that case the JDKs can too be compared.

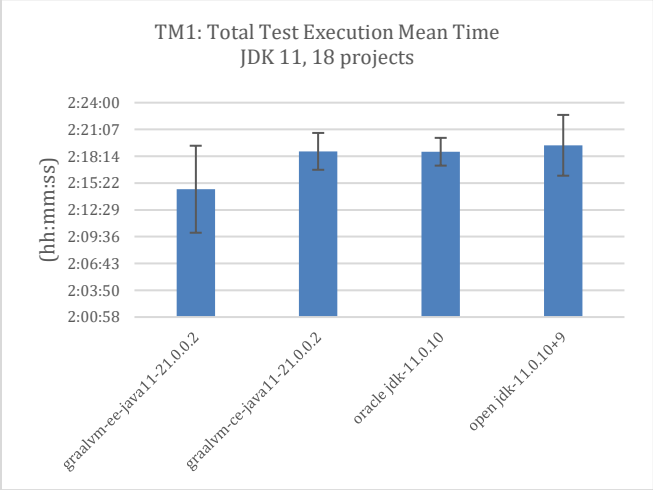


Figure 8. Total mean test execution time chart for Java 11 JDKs (TM1). For 18 projects that were tested for JDK 11, GraalVM EE 11 performed best on average, however it also has a higher standard deviation.

C. Mean Test Execution Time Per Project

This section presents the mean test execution time per project for both test machines. First, we present the results for TM2, then we present the results for TM1.

Test Machine 2

Table 4 shows the individual mean test execution time for the projects on TM2 and the result indicates that the gains were more visible for projects that have greater than a minute of execution time.

Table 4. Mean test executions time per project summary (TM2).

Project	EE11	CE11	Oracle11	Open11	EE 8	CE 8	Oracle 8	Open 8
rxJava	06:17	06:18	06:20	06:22	05:58	06:04	05:58	06:00
guava	10:31	11:05	10:52	11:11	10:22	10:59	10:53	10:58
zxing	00:48	00:51	00:51	00:51	00:47	00:49	00:49	00:49
clojure	01:10	01:13	01:15	01:15	01:09	01:11	01:13	01:14
junit4	00:12	00:12	00:10	00:11	00:11	00:11	00:10	00:10
vert.x	09:12	09:29	09:24	09:26	09:12	09:23	09:42	09:36
zipkin	02:39	02:35	02:33	02:35	02:21	02:20	02:11	02:15
guice	01:15	01:16	01:04	01:05	01:22	01:23	01:11	01:12
mybatis-3	01:55	01:56	01:54	01:57	01:53	01:57	01:55	02:05
webmagic	00:26	00:26	00:26	00:26	00:26	00:26	00:25	00:26
atmosphere	00:53	00:54	00:54	00:54	00:53	00:54	00:54	00:54

In Table 5 we see a comparison of the best and the worst performer for Java 8 JDKs. The data indicates a noticeable variation between the projects where seven out of the eleven projects indicated a faster execution on GraalVM EE.

Table 5. Comparison between best and worst performer for Java 8 JDKs (TM2).

Project	Test Cases Run	GraalVM EE 8	OpenJDK 8	EE 8 vs. Open 8
guava	858390	10:22	10:58	5.79%
vert.x	4576	09:12	09:36	4.35%
rxJava	13044	05:58	06:00	0.56%
zipkin	1055	02:21	02:15	-4.26%
mybatis-3	1675	01:53	02:05	10.62%
guice	7200	01:22	01:12	-12.20%
clojure	651	01:09	01:14	7.25%
atmosphere	264	00:53	00:54	1.89%
zxing	474	00:47	00:49	4.26%
webmagic	94	00:26	00:26	0.00%
junit4	1108	00:11	00:10	-9.09%

The same comparison between GraalVM EE 11 and OpenJDK 11 is presented in Table 6 with similar data except for the *mybatis/mybatis-3* project that showed significantly less gain.

Table 6. Additional comparison between Java 11 JDKs (TM2).

Project	Test Cases Run	GraalVM EE 11	OpenJDK 11	EE 11 vs. Open 11
guava	858390	10:31	11:11	6.34%
vert.x	4576	09:12	09:26	2.54%
rxJava	13044	06:17	06:22	1.33%
zipkin	1055	02:39	02:35	-2.52%
mybatis-3	1675	01:55	01:57	1.74%
guice	7200	01:15	01:05	-13.33%
clojure	651	01:10	01:15	7.14%
atmosphere	264	00:53	00:54	1.89%
zxing	474	00:48	00:51	6.25%
webmagic	94	00:26	00:26	0.00%
junit4	1108	00:12	00:11	-8.33%

Figure 9 and Figure 10 show the average of each JDK on each project that ran on TM2. The projects are sorted from shortest to longest (left to right) execution time. The graph shows that in most cases GraalVM EE has the lowest execution time on average, however the difference becomes more visible for the projects with the longest running test suites; *eclipse-vert.x/vert.x* and *google/guava*.

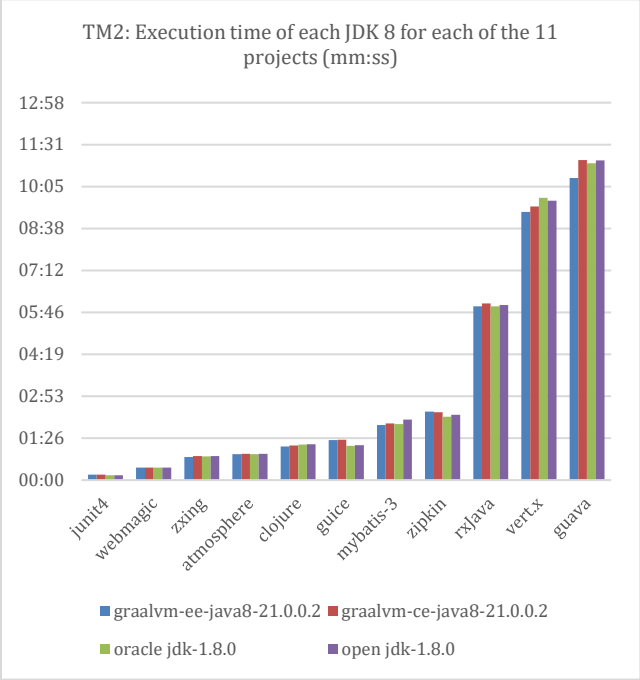


Figure 9. Mean execution time per project for JDK 8 (TM2). GraalVM EE performs best (on average) in most cases, but the difference in performance is more obvious for projects with a long running test suite.

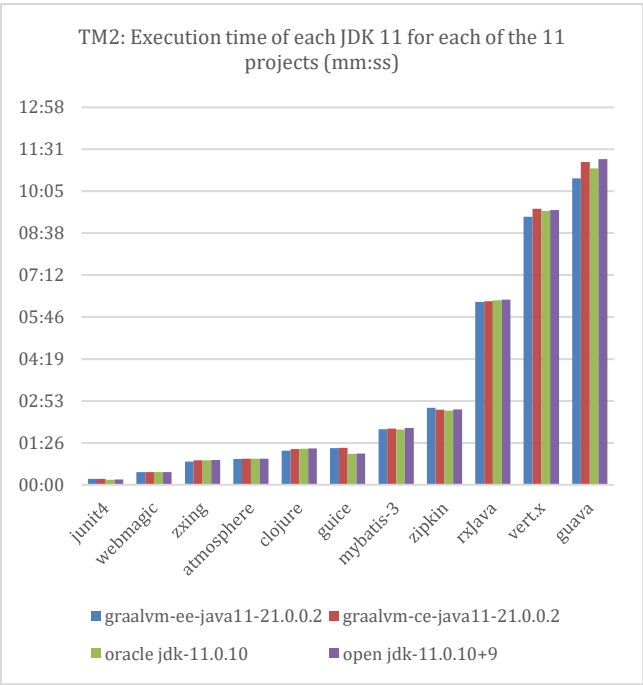


Figure 10. Mean execution time per project for JDK 11 (TM2). Gives a similar picture to Figure 9.

Test Machine 1

Table 7 shows a similar table as Table 4 for mean test execution time, but for the projects that were executed on TM1.

Table 7. Mean test executions time per project summary (TM). The '-' indicates that no data was obtained, mainly due to execution errors.

Project	EE11	CE11	Oracle11	Open11	EE8	CE8	Oracle8	Open8
javapoet	-	-	-	-	00:45	00:45	00:45	00:44
spring-mvc-showcase	-	-	-	-	00:38	00:37	00:38	00:38
AndResGuard	00:09	00:10	00:09	00:09	00:10	00:08	00:08	-
Bukkit	00:19	00:21	00:19	00:19	00:19	00:19	00:17	00:18
Jsoup	00:39	00:41	00:41	00:42	00:36	00:35	00:35	-
junit4	00:42	00:44	00:41	00:41	00:36	00:37	00:35	00:35
flyway	00:45	00:46	00:46	00:56	-	-	-	-
webmagic	00:56	00:50	00:51	00:53	00:45	00:45	00:45	-
atmosphere	02:10	02:10	02:13	02:14	02:07	02:07	02:05	-
zxing	03:03	03:07	03:04	03:03	03:09	03:01	02:51	02:54
clojure	03:56	04:12	04:07	04:08	03:53	04:02	03:49	03:47
guice	05:27	05:34	05:35	05:31	06:20	06:21	06:05	06:04
error-prone	06:24	06:56	06:56	07:03	07:14	-	06:47	06:46
metrics	06:25	06:20	06:23	06:21	09:30	09:16	09:03	-
mybatis-3	09:24	09:26	08:50	09:08	09:48	11:27	08:28	-
che	11:05	11:06	11:19	11:22	-	-	-	-
RxJava	12:28	12:01	12:10	12:15	11:18	10:52	10:26	-
vert.x	16:57	17:31	17:49	17:47	16:41	17:09	16:52	-
java-design-patterns	19:55	21:00	21:26	20:57	-	-	-	-
guava	33:56	35:52	35:24	35:55	31:44	37:29	29:18	29:04

The following tables are like Table 5, but for TM1 JDK 8 and 11. In none of the 8 projects in Table 8 for JDK 8 does GraalVM EE perform better than the best performng JDK, OpenJDK.

Table 8. Additional comparison between Java 8 JDKs (TM1).

Project	EE8 (mm:ss)	Open JDK 8 (mm:ss)	EE 8 vs. Open 8
guava	31:44	29:04	-8,40%
guice	06:20	06:04	-4,21%
clojure	03:53	03:47	-2,58%
zxing	03:09	02:54	-7,94%
javapoet	00:45	00:44	-2,22%
spring-mvc-showcase	00:38	00:38	0,00%
junit4	00:36	00:35	-2,78%
Bukkit	00:19	00:18	-5,26%

For JDK 11 however, GraalVM EE performed better than the runner up Oracle JDK on 11 occasions out of 18. OracleJDK performed better than GraalVM on five occasions, and on two occasions they performed equally well.

Table 9. Additional comparison between Java 11 JDKs (TM1).

Project	EE11 (mm:ss)	Oracle JDK 11 (mm:ss)	EE 11 vs. Oracle 11
guava	33:56	35:24	4,32%
java-design-patterns	19:55	21:26	7,62%
vert.x	16:57	17:49	5,11%
RxJava	12:28	12:10	-2,41%
che	11:05	11:19	2,11%
mybatis-3	09:24	08:50	-6,03%
metrics	06:25	06:23	-0,52%
error-prone	06:24	06:56	8,33%
guice	05:27	05:35	2,45%
clojure	03:56	04:07	4,66%
zxing	03:03	03:04	0,55%
atmosphere	02:10	02:13	2,31%
webmagic	00:56	00:51	-8,93%
flyway	00:45	00:46	2,22%
junit4	00:42	00:41	-2,38%
jsoup	00:39	00:41	5,13%
Bukkit	00:19	00:19	0,00%
AndResGuard	00:09	00:09	0,00%

Figure 11 and Figure 12 show the average of each JDK on each project that ran on TM1. Figure 11 and Figure 12 show a similar result to that of Figure 9 and Figure 10. However, both the CE and EE versions of GraalVM perform poorly in the results shown in Figure 11, especially at the *google/guava* project.

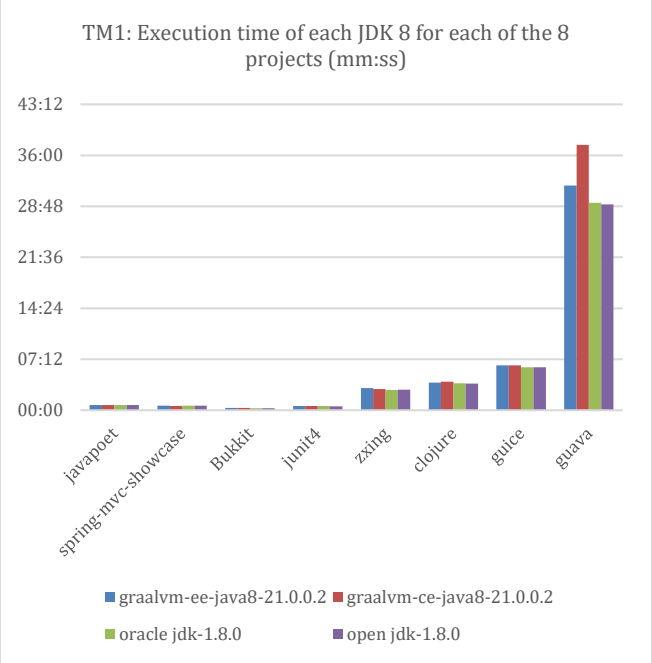


Figure 11. Mean execution time per project for JDK 8 (TM1). This figure does not show that there are performance gains of using GraalVM.

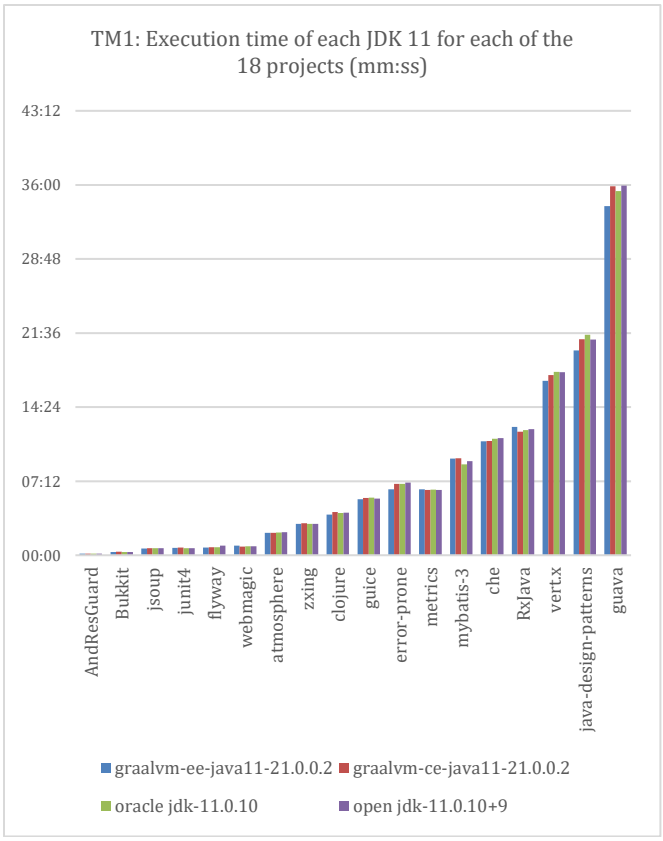


Figure 12. Mean execution time per project for JDK 11 (TM1). Gives a similar picture as Figure 9 and Figure 10, but for the 18 projects that were tested on TM1.

D. Execution Time Summary

The following tables and figures summarize the execution time result sections. Figure 13, Figure 14, and Figure 15 show pie charts comparing the average of running GraalVM EE 5 times on each project, against the best performing JDK. For TM2 (Figure 13), GraalVM EE was superior to the other best performing JDK (OpenJDK 8/11) in 64% of the tests (7/11 projects). They performed equally in 9% of the tests (1/11 projects) and performed worse in 27% of the tests (3/11 projects).

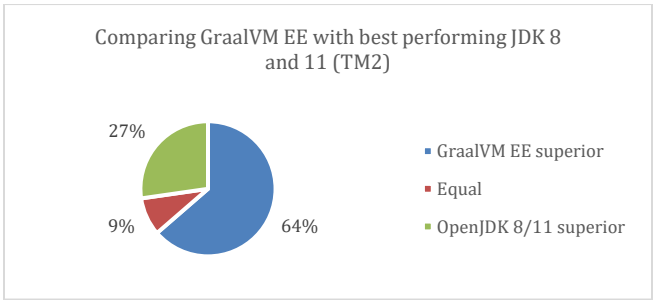


Figure 13. Overview of when the best performing JDKs were superior to each other. The results are based on the average of the JDK 8 & 11 tests that were performed on TM2. The results show that GraalVM EE is superior to the closest performing JDK in most cases.

For the tests on TM1 JDK 11 (Figure 14) GraalVM EE performed best in 61% of the cases (11/18 projects) against Oracle JDK 11, equally as well in 28% of the cases (5/18 projects) and worse in 11% of the cases (2/18 projects).

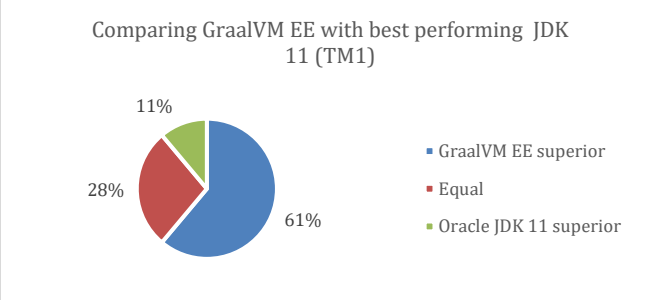


Figure 14. Overview of when the best performing JDKs were superior to each other) The results are based on the average of the JDK 11 tests that were performed on TM1. The results show that GraalVM EE is superior to the closest performing JDK in most cases.

Figure 15 shows results performed on TM1 JDK 8 for GraalVM EE and the other best performing JDK (OpenJDK 8). Here GraalVM EE was not the best performer in any projects (0/8) and once did equally as good as OpenJDK 8 (1/8). In 88% of the cases, OpenJDK 8 performed better than GraalVM EE 8 (7/8 projects).

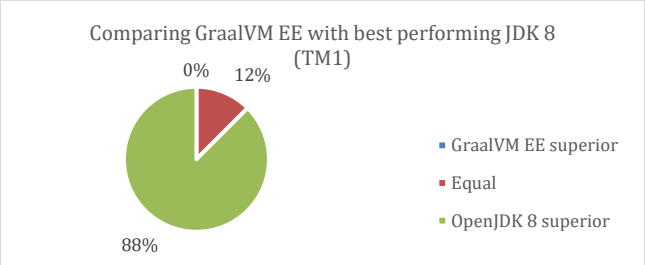


Figure 15. Overview of when the best performing JDKs were superior to each other. The results are based on the average of the JDK 8 tests that were performed on TM1. The results do not show that GraalVM EE is superior to the closest performing JDK.

Table 10 summarizes the data collected in this section in terms of when GraalVM EE did not benefit the execution time compared to the other JDKs, when it did, and when it gave mixed results.

Table 10. Time periods where GraalVM EE was beneficial for TM1 and TM2.

	No benefits	Mixed results	Only benefits
TM1 JDK 8	0–31:44	-	-
TM1 JDK 11	<= 00:39	00:40-12:28	12:29-33:56
TM2 JDK 8	<=00:47	00:47-02:21	02:21-10:22
TM2 JDK 11	<=00:48	00:48-02:39	02:39-10:31

E. Test Execution Time and Test Case Relation

Figure 16 shows the individual test execution time for each project in relation to the number of test cases for GraalVM EE 8, 11, and OpenJDK 8, 11.

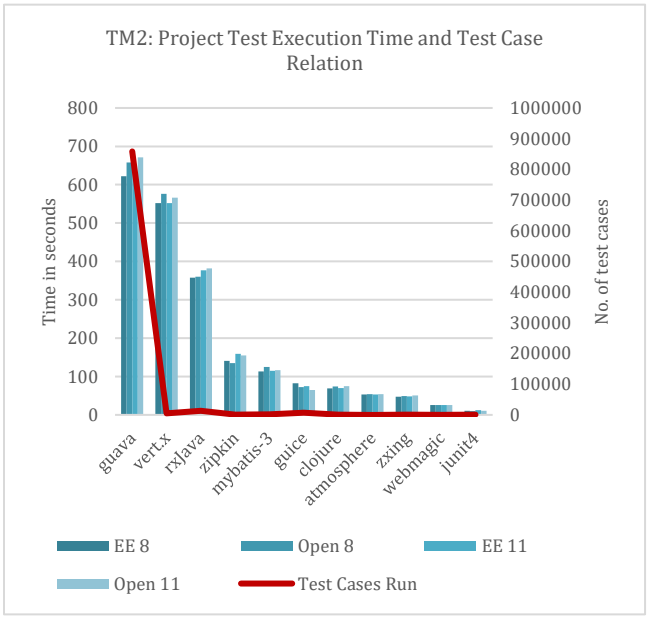


Figure 16. Project execution time and test case chart (TM2). The figure shows that there is not necessarily a correlation between the number of test cases and the length of the execution time. This is especially obvious in the comparison of the 'vert.x' (4576 test cases) and 'rxjava' (13044 test cases) projects, where 'rxjava' has a lower execution time despite having a higher number of test cases.

F. Validity Threats

As there was no aligned way of presenting the test results and the total number of test cases executed for Maven and Gradle projects, a lot of manual processing of test logs were necessary which could have introduced some inaccuracy in the amount of total test cases executed.

An additional threat to the validity is the number of projects tested. TM2 tested 11 projects and TM1 tested 20 projects, TM1 however could not run all JDKs on all 20 projects. Only 6 projects were used to compare all JDKs, 8 projects were used to compare Java 8 JDKs, while Java 11 JDKs were well represented with 18 projects. One reason why only 8 Java 8 projects were tested is because of situations where OpenJDK was the only JDK with an error. Another aspect that could threaten the validity is the results in Table 10. Here, the columns "No benefits" and "Only benefits" are based on data from 2-3 projects.

Another thing that could threaten the validity are the pie charts that try to summarize the results (Figure 13, Figure 14, and Figure 15). These pie charts show the number of times one JDK was superior to another, however it does so by looking at the average of running each JDK 5 times on each

project. The average might not give the right indication, because a JDK might for example have one bad run which affects its average negatively.

VIII. Discussion

In this section, we discuss the results. First, we discuss the project success rate observed, then we answer the research questions, and lastly, we discuss ethical aspects and the limitations in this study.

A. Project Success Rate

During the data collection step, we stumbled upon failures in the build step and issues where test suites did not pass. Build failures due to missing dependency were a common problem when trying to compile and run tests from open-source Java projects. A quantitative study of the buildability of Java open-source software by Sulir et al. [32] has shown that around 38% of the open-source Java projects end up in a failure at the build step and often due to dependency related problems. We experienced an even greater failure rate where 69% for TM2 and 58% for TM1 of the Java projects failed in different ways during the build step on our test machines. This forced us to exclude these projects from the time measurement. The extra layer of running test suites, and the fact that the projects needed to build successfully on multiple JDKs, most likely contributed to the higher failure rate.

B. Answering the Research Questions

To answer the question:

> *RQ1: Could we speed up test execution by using GraalVM? If so by what degree?*

The results show that it is possible to speed up test execution by using GraalVM EE. GraalVM EE performed the best with JDK 8 and 11 for TM2 and with JDK 11 for TM1. GraalVM EE had better execution time compared to the runner up in 7 out of the 11 projects that were tested on TM2 (both JDK 8 and 11), and once it performed equally as well. GraalVM EE performed better than the runner up in 11 out of 18 projects with TM1 JDK 11. In two of the projects, it performed equally as well, and in five projects it performed worse than its closest competitor. These results are aligned with the previous results of Larsson [9] and Sipek et al. [10] which also showed that GraalVM EE performed the best. However, as stated in *Validity Threats*, the comparisons are based on the averages of running each JDK 5 times on each project.

Although the overall gain seen was in the lower single digit percentages it could impact the operation cost in the development phase and in the regression testing. This would reduce the number of resources needed and thus reduce the cost and impact on the environment.

GraalVM EE for JDK 8 on TM1 however did not perform better than the Oracle and OpenJDK 8 counterparts. It is unclear why GraalVM performs worse than OpenJDK and Oracle JDK on TM1, however a possible explanation might be that GraalVM is more dependent on the hardware it runs on than the other JDKs, since that is the main difference between TM1 and TM2. TM1 and TM2 have the same version of the same operating system (however TM2 ran it as a virtual machine), but different hardware. This observation is further proved by the work of Sipek et al. [10] which observed that GraalVM EE performed better when it ran multithreaded programs. Multithreaded programs run better when having more resources (cores and threads), and this might explain why GraalVM EE 8 performed better on TM2 than on TM1, since TM2s virtual machine is given 5 cores compared to TM1s four cores, and TM2 uses ten threads compared to TM1s four.
If this observation is proven to be general, then this result can be used to state that GraalVM improves the execution time of running the test suite on a server, since a server normally consists of powerful hardware. More studies however need to be conducted to examine if this is a general observation.

GraalVM EE benefitted JDK 8 and 11 for TM2 and JDK 11 for TM1 in a range of 0.55%-10.62%. GraalVM CE 8 and 11 however did not show any improvements for neither TM1 nor TM2, and in most cases it performed either worst or second to worst. This result is also aligned with the previous result of Larsson [9] where GraalVM CE did not perform so well. A part of the explanation for why GraalVM EE performs better than GraalVM CE might be that the CE version does not have auto-vectorization [20]–[22].

To answer the question:

> *RQ2: Is there a (or what is the) breaking point for which GraalVM does (not) give any benefits?*

The result of the number of test cases and the execution time proved that it was not a suitable metric for identifying the breaking point where GraalVM would give benefits. The execution time and the number of test cases did not correlate. A high amount of test case runs does not indicate that it would take a long time to run. Some projects had around 850 000 test case runs which took a similar time as a project with around 4500 test case runs.

An observation is that GraalVM EE does not give any benefits for the lower time-consuming projects. Table 10 shows that JDK 11 for TM1 did not show any benefits of using GraalVM for test suites with execution time less than or equal to 39 seconds. For TM2 that number was 47 seconds for JDK 8 and 48 seconds for JDK 11.

The same table also showed that GraalVM performed better than its competitors on the last 2-3 longest running projects,

which might indicate that the benefits of GraalVM become clearer as the execution time becomes higher.

More studies need to be conducted that examine test suites of projects that take an even longer time to execute. As written in *Validity Threats*, however, this result is based on the performance of 2-3 projects.

To answer the question:

> *RQ3: Are there any drawbacks or compatibility issues with GraalVM on open-source projects compared to other JDKs?*

In some instances, the GraalVM performed the worst. One of the projects that proved a slower execution was *google/guice* which is a lightweight dependency injection framework. In this project, GraalVM EE was over ~12% slower than OpenJDK. Compatibility issues were also observed as the project *dropwizard/dropwizard* on both TM1 and TM2 ended up in build failure for the GraalVM EE and CE 11 versions. All other JDKs passed for this project.

C. Ethical Aspects

Test execution in software development requires resources and resources, in turn, requires energy, we have had the cost and environmental aspect in mind. By the hope of reducing the resources needed for software development, we would reduce the cost and the environmental impact. Both open-source and commercially licensed versions of JDKs were included in this study to exclude any exclusivity.

D. Limitations

Unit test, integration test, and *functional test* have been a common classification of tests in the software development industry. In this study, we do not emphasize or make any distinction of the types of tests in the repositories during project selection. A recent study has also indicated that in modern software development that the types of defects found for unit test and integration test are similar and stated that this classification is no longer suitable in the modern software development context [33].

IX. Conclusions

The results show that GraalVM EE 8 and 11, based on the average, benefitted the performance of 7 out of 11 projects that ran on TM2 and GraalVM 11 benefitted the performance of 11 out of 18 projects that ran on TM1. In one case it performed equally as well as its competitor on TM2 and for TM1 that was the case 5 times. The overall improvement was in the lower single-digit percentages for the time taken to run eleven (TM2) and twenty (TM1) open-source project test suite runs. Although the data proved an overall gain, the data showed a noticeable variation for individual project results where GraalVM also in some cases performed worse than the overall worst performing JDK. A comparison between GraalVM EE 8 and OpenJDK 8 showed a range between ~11% to ~(-12)% for GraalVM EE 8.

However, for JDK 8 and 11 TM2, and JDK 11 on TM1, GraalVM EE in most cases benefitted the execution time in a range of 0.55%-10.62%, compared to its closest competitor. GraalVM CE 8 and 11 however did not benefit the performance of running the test suite neither on TM1 or TM2. In all cases, GraalVM CE performed worst or second to worst, compared to the other JDKs. A reason for that might be that GraalVM CE does not use the same compilation optimization techniques as GraalVM EE, such as auto-vectorization. These results are in line with the results of some of the previous studies.

GraalVM EE 8 however did not benefit the performance on TM1. A possible explanation for that is that TM2 is superior to TM1 in terms of hardware and processing power, and this might indicate that GraalVM EE 8 is more dependent on the hardware it runs on compared to the other JDKs. A previous study has shown that GraalVM performs better on multithreaded programs, which might explain this finding. This can be examined further by testing GraalVM against other JDK on test machines that run the same software but different hardware and/or focus the tests on multithreaded programs.

The number of test cases is not the most suitable indication of how complex or time-consuming the test step is. As test cases vary in both complexity and time to run independently of the number of test cases. However, in terms of the breaking point for when GraalVM no longer is beneficial, it was observed that GraalVM EE JDK 8 did not give any benefits for projects that ran for less than 48 seconds on TM2, JDK 11 did not give any benefits for projects running for less than 49 seconds on TM2, and GraalVM EE JDK 11 did not give any benefits for projects that ran less than 39 seconds for TM1. An observation made is that the longer the execution time takes the clearer it becomes that GraalVM performs better than its competitors. This could also be studied further by focusing the tests on project test suites with long execution times.

Apart from that, it could be interesting to evaluate *Java on Truffle*. Java on Truffle[9] is a new experimental technology of GraalVM and was released in version 21.0.x Java on Truffle is a minimized version of the JVM and follows the Java Virtual Machine Specification. At the point of release, it was stated that the current version is 2-3 times slower than the Java HotSpot Virtual Machine, but this can of course change in future releases. In version 21.1.x[10] of GraalVM, further compiler optimization is introduced which could be of

[9] https://www.graalvm.org/reference-manual/java-on-truffle/

[10] https://www.graalvm.org/release-notes/21_1/

interest when it comes to performance. Also, this study focused on Java, similar research could be conducted for other programming languages that can run on the JVM, e.g., Clojure.

REFERENCES

[1] D. D. Nardo, N. Alshahwan, L. Briand, and Y. Labiche, "Coverage-based regression test case selection, minimization and prioritization: a case study on an industrial system," *Softw. Test. Verification Reliab.*, vol. 25, no. 4, pp. 371–396, 2015, doi: https://doi.org/10.1002/stvr.1572.

[2] J.-M. Kim and A. Porter, "A history-based test prioritization technique for regression testing in resource constrained environments," in *Proceedings of the 24th International Conference on Software Engineering*, New York, NY, USA, May 2002, pp. 119–129. doi: 10.1145/581339.581357.

[3] H. Spieker, A. Gotlieb, D. Marijan, and M. Mossige, "Reinforcement Learning for Automatic Test Case Prioritization and Selection in Continuous Integration," *Proc. 26th ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, pp. 12–22, Jul. 2017, doi: 10.1145/3092703.3092709.

[4] P. Stratis and A. Rajan, "Reordering tests for faster test suite execution," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, New York, NY, USA, May 2018, pp. 442–443. doi: 10.1145/3183440.3195048.

[5] P. Stratis, "Software testing: test suite compilation and execution optimizations," Jun. 2020, doi: 10.7488/era/411.

[6] P. Stratis and A. Rajan, "Speeding up test execution with increased cache locality," *Softw. Test. Verification Reliab.*, vol. 28, no. 5, p. e1671, 2018, doi: https://doi.org/10.1002/stvr.1671.

[7] P. Stratis and A. Rajan, "Test case permutation to improve execution time," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Sep. 2016, pp. 45–50.

[8] "GraalVM Documentation." https://www.graalvm.org/docs/introduction/ (accessed Mar. 27, 2021).

[9] R. Larsson, *Evaluation of GraalVM Performance for Java Programs*. 2020. Accessed: Mar. 21, 2021. [Online]. Available: http://urn.kb.se/resolve?urn=urn:nbn:se:lnu:diva-97537

[10] M. Šipek, B. Mihaljević, and A. Radovan, "Exploring Aspects of Polyglot High-Performance Virtual Machine GraalVM," in *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, May 2019, pp. 1671–1676. doi: 10.23919/MIPRO.2019.8756917.

[11] M. Šipek, D. Muharemagić, B. Mihaljević, and A. Radovan, "Enhancing Performance of Cloud-based Software Applications with GraalVM and Quarkus," in *2020 43rd International Convention on Information, Communication and Electronic Technology (MIPRO)*, Sep. 2020, pp. 1746–1751. doi: 10.23919/MIPRO48935.2020.9245290.

[12] C. Wimmer *et al.*, "Initialize once, start fast: application initialization at build time," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, p. 184:1-184:29, Oct. 2019, doi: 10.1145/3360610.

[13] M. Kumar, A. Sharma, and R. Kumar, "Optimization of Test Cases using Soft Computing Techniques: A Critical Review," *Inf. Sci. Appl.*, vol. 8, no. 11, p. 13, 2011.

[14] "JDK | Java Development Kit - Javatpoint," *www.javatpoint.com*. https://www.javatpoint.com/jdk (accessed Apr. 18, 2021).

[15] "Java™ Platform Overview." https://docs.oracle.com/javase/7/docs/technotes/guides/index.html (accessed Apr. 18, 2021).

[16] "JVM | Java Virtual Machine - Javatpoint," *www.javatpoint.com*. https://www.javatpoint.com/jvm-java-virtual-machine (accessed Apr. 18, 2021).

[17] O. Šelajev, "Announcing GraalVM 19," *Medium*, May 12, 2019. https://medium.com/graalvm/announcing-graalvm-19-4590cf354df8 (accessed Apr. 17, 2021).

[18] "GraalVM." https://www.graalvm.org/ (accessed Apr. 17, 2021).

[19] "GraalVM: Clearing up the confusion and why Twitter uses it in production." https://jaxenter.com/graalvm-chris-thalinger-interview-163074.html (accessed Apr. 17, 2021).

[20] TylerMSFT, "Auto-Parallelization and Auto-Vectorization." https://docs.microsoft.com/en-us/cpp/parallel/auto-parallelization-and-auto-vectorization (accessed Jun. 05, 2021).

[21] "SIMD / Autovectorization support · Issue #892 · oracle/graal," *GitHub*. https://github.com/oracle/graal/issues/892 (accessed Jun. 05, 2021).

[22] Apete, "oj! Blog: Oracle's JVMs HotSpot, Graal CE & Graal EE," *oj! Blog*, Feb. 18, 2019. https://ojalgo.blogspot.com/2019/02/oracles-jvms-hotspot-graal-ce-graal-ee.html (accessed Jun. 05, 2021).

[23] L. Stadler, T. Würthinger, and H. Mössenböck, "Partial Escape Analysis and Scalar Replacement for Java," *F*, p. 10.

[24] T. Kotzmann and H. Mossenbock, "Run-Time Support for Optimizations Based on Escape Analysis," in *International Symposium on Code Generation and Optimization (CGO'07)*, Mar. 2007, pp. 49–60. doi: 10.1109/CGO.2007.34.

[25] "Maven – Introduction." https://maven.apache.org/what-is-maven.html (accessed Apr. 18, 2021).

[26] "What is Gradle?" https://docs.gradle.org/current/userguide/what_is_gradle.html (accessed Apr. 18, 2021).

[27] "DaCapo Benchmarks." http://dacapobench.org/ (accessed Apr. 17, 2021).

[28] K. Björk, *A comparison of compiler strategies for serverless functions written in Kotlin*. 2020. Accessed: Feb. 17, 2021. [Online]. Available: http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-273961

[29] H. Borges, A. Hora, and M. T. Valente, "Understanding the Factors That Impact the Popularity of GitHub Repositories," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Oct. 2016, pp. 334–344. doi: 10.1109/ICSME.2016.31.

[30] H. Borges and M. Tulio Valente, "What's in a GitHub Star? Understanding Repository Starring Practices in a Social Coding Platform," *J. Syst. Softw.*, vol. 146, pp. 112–129, Dec. 2018, doi: 10.1016/j.jss.2018.09.016.

[31] H. S. Borges and M. T. Valente, "Application Domain of 5,000 GitHub Repositories." Zenodo, Jun. 08, 2017. doi: 10.5281/zenodo.804474.

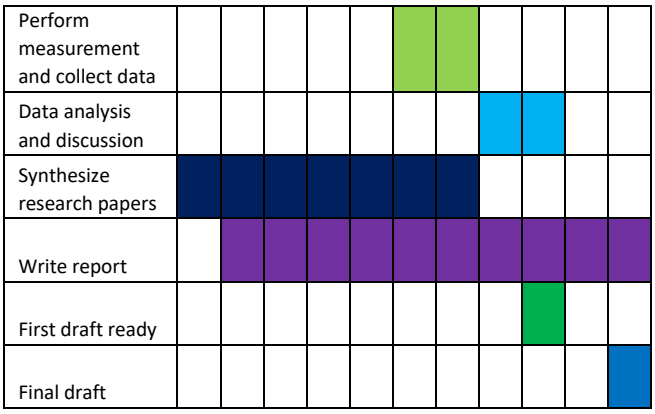
[32] M. Sulír and J. Porubän, "A quantitative study of Java software buildability," in *Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools*, New York, NY, USA, Nov. 2016, pp. 17–25. doi: 10.1145/3001878.3001882.

[33] F. Trautsch, S. Herbold, and J. Grabowski, "Are unit and integration test definitions still valid for modern Java projects? An empirical study on open-source projects," *J. Syst. Softw.*, vol. 159, p. 110421, Jan. 2020, doi: 10.1016/j.jss.2019.110421.

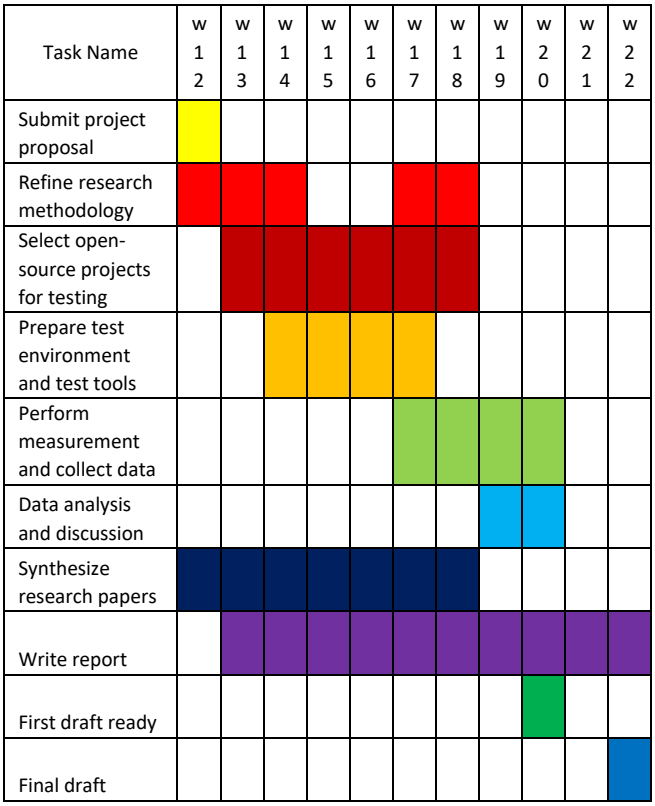
APPENDIX I: TIME PLAN

Initial Time Plan

Task Name	w12	w13	w14	w15	w16	w17	w18	w19	w20	w21	w22
Submit project proposal											
Refine research methodology											
Select open-source projects for testing											
Prepare test environment and test tools											



Actual Time Plan



Appendix II: Contributions

Fredric Fong

Thesis Sections
Abstract: 0% | Introduction: 0% | Problem Statement: 100% | Background: 65% | Related Work: 75% | Research Methodology: 50% | Artifact: 10% | Results: 50% | Discussion: 50% (Project Success Rate, Ethical Aspects, Limitations, Answering the Research Questions 50%) | Conclusion 40%

Other
Tables | Figures | Overall Formatting, Structuring & Outlining | Overall Clean-Up | References | Measurements performed on Test Machine 2 | Time Plan

Mustafa Raed

Thesis Sections
Abstract: 100% | Introduction: 100% | Problem Statement: 0% | Background: 35% | Related Work: 25% | Research Methodology: 50% | Artifact: (section + implementation) 90% | Results: 50% | Discussion: 50% | Conclusion 60%

Other
Tables | Figures | Overall Clean-Up | Measurements performed on Test Machine 1 | Time Plan

Appendix III: Project Data Set And Result

The project data set, result and calculation can be found at the following link: https://github.com/mustafaRaed/miun-jdk-test-tool/tree/master/results/.