CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

A Framework for Minimising Java Microbenchmark Suites: A Metric-Based Approach

Master's thesis in Computer science and engineering

Filip Darin Nordqvist
Rasmus Otterlind

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2024

MASTER'S THESIS 2024

A Framework for Minimising Java Microbenchmark Suites: A Metric-Based Approach

Filip Darin Nordqvist
Rasmus Otterlind

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2024

A Framework for Minimising Java Microbenchmark Suites: A Metric-Based Approach
Filip Darin Nordqvist
Rasmus Otterlind

Supervisor: Philipp Leitner, Computer Science and Engineering
Examiner: Gregory Gay, Computer Science and Engineering

Master's Thesis 2024
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2024

A Framework for Minimising Java Microbenchmark Suites: A Metric-Based Approach

FIlip Darin Nordqvist & Rasmus Otterlind
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Performance is an important non-functional requirement. The success of software projects is highly dependent on the adequate performance of the software. However, one fundamental problem with performance testing is that it can become immensely time-consuming to conduct. In this paper, several different code metrics were explored and used for conducting test case minimization (TCM) of Java Microbenchmark Harness (JMH) suites. The microbenchmark suites were created with a tool called *ju2jmh*, which creates microbenchmark suites based on already existing unit testing suites of a project. The metrics used for conducting TCM were lines of code (LOC), loop count, cyclomatic complexity, and the combination of these metrics. The results indicate that suites derived using LOC and loop count performed worse than randomly sampled suites, while suites derived from the cyclomatic metric and the combination of LOC, loop count, and cyclomatic complexity performed similar or slightly worse than randomly sampled suites. Furthermore, it is not infeasible to assume that if these metrics were refined by utilising dynamic techniques, some of them could potentially outperform randomly sampled suites consistently.

Keywords: Performance testing, Java Microbenchmark Harness, test case minimisation, artificial slowdowns, code coverage, computational complexity, cyclomatic complexity, static code analysis.

Acknowledgements

Special thanks to Philipp Leitner, our supervisor, for providing valuable feedback and showing support throughout the scope of this thesis.

Filip Darin Nordqvist & Rasmus Otterlind, Gothenburg, June 2024

Contents

List of Figures

List of Tables

1

Introduction

Performance testing is a programming technique that strives to determine the performance of a system regarding aspects such as stability, response time and scalability. Furthermore, for innumerable software systems, performance plays a crucial role in the usability and quality of the system [1]. So, the overall success or failure of a software engineering project might be heavily influenced by the non-functional requirement performance [1].

Java Mircrobenchmarking Harness (JMH) [2] is a commonly used tool for conducting performance benchmarking targeting programming languages that utilise the Java virtual machine (JVM) [3]. Examples of programming languages that use the JVM besides Java are Kotlin, Scala, and Groovy. A JMH microbenchmark repeatedly runs a small amount of code and collects data related to the performance of that specific code. One fundamental problem with JMH microbenchmarking is that it can become tremendously costly to execute full microbenchmarking suites. Execution time for a single suite can vary between several hours to several days making it impractical to integrate in tools such as a continuous integration (CI) pipeline [4] [5] [6].

1.1 Problem Statement

Although performance testing is vitally important for many software systems, it is not a common practice. Software projects often utilise unit tests, but it is not common for projects to engage in software microbenchmarking. Evidence supporting this can be seen by inspecting the Maven repository usage count for JMH and JUnit, respectively [7][8]. And in cases where performance testing is being done with JMH, it is sometimes the case that the configuration and construction of microbenchmarks are compromised in terms of bad practices [9]. Alexanderson [10] has created a tool that can create JMH benchmarks from JUnit tests called *ju2jmh*. This means that software projects with JUnit suites can use the suites to generate JMH microbenchmark suites.

In this study, a tool was created to try to evaluate which JUnit tests are more suitable to be used to create microbenchmarks. The tool is able to create a minimised microbenchmark suite from the JUnit tests present in a project. This tries to build upon the tool *ju2jmh*. More specifically, projects that utilise JUnit often have several thousand test cases, and generating microbenchmarks from all of these would create suites that could take several weeks to execute. Additionally, some generated microbenchmarks display poor stability. Stability in terms of performance microbenchmarking with JMH refers to the variability in the results of the microbenchmark. If the results have high variance, the microbenchmark is much more likely to report false positives in terms of detecting performance degradation. In this study, the applicability of prioritising JUnit tests for the generation of microbenchmarks is investigated.

The new tool that has been built for this study tries to derive from statically gained metrics which JUnit tests are most suitable to be used to generate JMH microbenchmarks with the help of *ju2jmh*. Using all the JUnit test would render the suites infeasible to use effectively in many situations, such as in CI pipelines [4].

1.2 Purpose of the Study

The purpose of this study was to develop a framework for generating minimised Java Microbenchmark Harness (JMH) suites using a metric-based approach. The goal was to create minimised microbenchmark suites by generating them from prioritised JUnit tests using ju2jmh. The framework is built to methodically evaluate the suitability of individual JUnit tests to be converted to microbenchmarks, based on metrics derived from static code analysis. Static code analysis was chosen because it is simpler to implement, and it was of interest to investigate whether measuring simple metrics such as Lines Of Code (LOC) through static analysis would be sufficient to generate adequate suites. Moreover, static analysis does not entail as much overhead as dynamic approaches, where, for example, entire JUnit test suites need to be executed to gather information about code coverage. In situations where time is limited and the test suite is very large, static analysis could be a more practical approach. The ambition was to create microbenchmark suites that are more suitable for use in time-constrained environments, such as continuous integration or continuous deployment (CI/CD) pipelines. This, in turn, could greatly benefit developers and organisations by increasing the efficiency of development pipelines while fostering performance integrity in their software systems.

1.3 Research questions

- RQ 1: *How does a generated minimised benchmark suite based on code coverage calculated through static analysis compare to randomly generated minimised suites in terms of the ability to detect performance regressions?*

  This research question served to see if code coverage could be used to generate efficient microbenchmark suites that are able to detect performance regressions. It was also interesting to investigate if prioritising code coverage would come at the cost of stability or the ability to detect smaller performance regressions. The results gained from this work indicate that micorbenchmark suites derived using the static code analysis metric lines of code (LOC), which served as a proxy for code code coverage, perform worse than randomly sampled suites. This is most likely due to the difference in the number of LOCs being executed dynamically versus what is estimated through static analysis.

- RQ 2: *How does a generated minimised benchmark suite based on computational complexity calculated through static analysis compare to randomly generated minimised suites in terms of the ability to detect performance regressions?*

  With this research question, the aim was to see if microbenchmark suites generated using metrics related to computational complexity were good at detecting performance regressions. Computational complexity refers to the runtime of algorithms and programmes; this can be calculated and estimated in some different ways. One common way to estimate computational complexity is to calculate the Big O notation, or estimate the number of loops that are executed. The reason that computational complexity was interesting to look at was because the hypothesis was that in areas of the code where computational complexity is high, small changes can lead to big performance regressions, meaning that those areas could be more important to test. The proxy metric that was used to estimate computational complexity was loop count. If the JUnit tests had a high number of loops that could be executed, they would be prioritised. The results showed that the loop-count-derived suites were poor at detecting performance regressions and also contained microbenchmarks that had worse stability in one of the projects tested.

- RQ 3: *How does a generated minimised benchmark suite based on code complexity calculated through static analysis compare to randomly generated minimised suites in terms of the ability to detect performance regressions?*

  Code complexity in this case refers to how complex the code is in terms of the number of branching execution paths and statements. If the code complexity of a function or programme is high, it may be more likely to have bugs or performance regressions introduced. So, the endeavour with RQ 3 was to see if microbenchmark suites generated using a metric related to code complexity resulted in suites that could reliably detect performance regressions, as these could be the areas of the code that are more important to test in terms of performance. Furthermore, investigating if there is a relationship between code complexity and overall suite stability was something that was interesting to explore as well. For conducting test case minimization with regard to code complexity, a metric based on cyclomatic complexity was implemented. This metric performed better than the previously mentioned metrics, and its performance was similar or slightly worse compared to the randomly sampled suites. Through refinements and further work with dynamic analysis implementations, it is not infeasible that the cyclomatic metric could consistently outperform the randomly sampled suites.

- RQ 4: *How does a generated minimised benchmark suite based on a combination of the code characteristics mentioned in RQ 1-3 compare to randomly generated minimised suites in terms of ability to detect performance regressions?*

  In order to fully utilise the metrics that were implemented and used to estimate the code characteristics in research questions 1–3, a combination of all three proxy metrics was used to create a combined metric. This combined metric was also evaluated in terms of its ability to detect performance regressions and overall microbenchmark stability. Overall, the combined metric results seem to indicate that it is somewhat worse than the randomly sampled suites at discovering and reporting performance regressions.

In order to answer the research questions outlined above, suites with 50 microbenchmarks were generated using the metrics, as well as suites that contained randomly selected JUnit tests that were converted to microbenchmarks. The metric-derived microbenchmark suites were evaluated and compared against the randomly sampled suites, which were used as a baseline that the metric derived suites aimed to beat. In the end, all the approaches were worse or slightly worse than generating randomly sampled suites. The poor results are likely connected to the fact that the Java source code and underlying metrics were parsed in a purely static environment, which likely affected the quality of the estimations significantly. So, moving from a static context to a dynamic context could possibly yield better results and should most likely be used over a static context as the benefits of a dynamic context should outweigh the increased overhead associated with a dynamic context.

1.4 Significance of the Study

The significance of this thesis was partly theoretical and partly practical. Theoretically, the aim was to advance the understanding of performance testing by looking into what metrics could be found through static code analysis to help evaluate which JUnit tests could be used as JMH microbenchmarks. Practically, the results can help to change how performance testing can be implemented in CI/CD pipelines, allowing for quick development cycles while preserving the software's performance quality. The thesis, therefore, aimed to deliver contributions by enhancing knowledge and efficiency in performance testing.

2

Background

The following chapter will provide essential background knowledge that is required to get a proper understanding of the study.

2.1 Performance Testing with JMH In Java

In the following sections, some of the relevant information needed to understand performance testing in Java is explained. This includes an explanation of the role of the Java virtual machine (JVM) and the Java microbenchmarking harness (JMH) and how they interplay.

2.1.1 Performance Testing

Performance testing relates to testing for the non-functional requirement referred to as performance [1]. The way that performance is measured and tested depends on the context, it is primarily dependant on programming language and hardware. In the context of performance testing for Java JMH is the primary tool used, and the main measurements used are measuring throughput, or average time for an execution of a benchmark method. Performance testing for software systems is often overlooked, although it can, as stated by E. Weyuker et al. [1], greatly affect the outcomes of a software product. Y. Zhao et al. [11] conducted a large-scale empirical study of performance issues in several large open source projects using Java, Python C, and C++. They found 570 issues in 13 large open source projects and inferred that practitioners likely lack a formal process to follow when verifying performance optimisations in regression testing.

2.1.2 Java Virtual Machine (JVM)

The Java Virtual Machine (JVM) [12] is a virtual machine that loads, verifies, and executes Java bytecode. More specifically, when a programming language is targeting the JVM, such as Java or Kotlin, its code gets compiled into Java bytecode by a compiler. This Java bytecode is then used by the JVM to create machine-specific code. This has some implications in terms of performance testing, as underlying hardware and the JVM can affect the results of benchmarks. One important feature of the JVM is that it uses just-in-Time (JIT) compilation [13]. Simply put, this means that the Java bytecode gets converted to machine-specific code just before the execution of that bytecode. This JIT compilation plays an important aspect in performance testing and is something that needs to be accounted for, as performance is highly dependent on what gets compiled using the JIT compilation.

2.1.3 Java Microbenchmark Harness (JMH)

JMH is used to evaluate and detect performance regressions for software running on the JVM. The way that this is done is by creating JMH microbenchmarks. Microbenchmarks run small pieces of code as many times as they can in several iterations over a time span. The time span and number of iterations depend on the configuration for the specific microbenchmark [2]. Given that JMH microbenchmarks evaluate code running on the JVM, some key aspects have to be taken into account. One of the main things that has to be accounted for is the just-in-time compiler (JIT). JIT performs optimisations to the code running on the JVM, so in order to get stable and useful results from JMH microbenchmarks, some of the iterations in a JMH microbenchmark are used to 'warm up' the JVM so that later iterations can get more correct data when the JVM has had more time to reach a steady state. In addition to this, different microbenchmarks display different characteristics; some cause the JVM to take more time to reach a steady state, and some cause the JVM to never reach a steady state [14].

The stability of the JVM when recording the results of microbenchmarks is important, as high variability in the results can lead to false positives in terms of detecting performance degradation. The stability of a microbenchmark refers to the variability between the results recorded from the non-warm-up iterations. Therefore, microbenchmarks with good stability are preferred as they help evaluate performance correctly and are less likely to report false performance degradations, also referred to as false positives.

For each microbenchmark there are several attributes that can be changed or added to make the JMH microbenchmark behave differently. Some attributes can be seen in figure 2.1.

```java
@Benchmark
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.MICROSECONDS)
@Warmup(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)
@Measurement(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)
public void testMethod(MyState state) {
    // your benchmarked code here
    int result = state.a + 42;
}
```

Figure 2.1: Generic JMH Benchmark.

The attribute *@BenchmarkMode(Mode.AverageTime)* is used to decide what is being measured. *Mode.AverageTime* means that the average time to execute the benchmarks method is going to be measured. Another mode is *Mode.Throughput*, which measures the number of times per second that the benchmark method could be executed. There are two more modes that can be used: *Sample Time*, which measures how long it takes for the benchmark method to execute, including maximum and minimum time. *Single Shot Time* that measures how long it takes to execute the benchmark method one time. This can be used to see how the benchmark behaves with a cold start with no warmup iterations. It is also possible to measure all of the modes described at the same time. However, usage of several modes leads to longer execution times as the benchmark results for each mode is gathered separately. The attribute *@OutputTimeUnit(TimeUnit.MICROSECONDS)* is used to specify in which time unit the results should be reported. The attributes *@Warmup(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)* and *@Measurement(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)* specify how many warmup iterations and measurement iterations are going to be used and how long each of these iterations are going to be. An important note is that these attributes can be overridden by the command line arguments supplied while launching the JMH microbenchmark suite. JMH can also report on the standard deviation of the measurement iteration score's and 99.9% error margins of these scores, which helps provide insight into how stable the benchmarks were. Both the standard deviation and the 99.9% score error margins are calculated from the results of the measurement iterations.

2.1.4 Artificial Slowdowns

In order to evaluate microbenchmark suites capabilities in terms of finding performance regressions, artificial slowdowns can and have been used [6]. Injection of artificial slowdowns was also used in this thesis project to evaluate microbenchmark suites. By running a microbenchmark suite before and after inserting artificial slowdowns, the number of benchmarks that detect slowdowns can be estimated, and if several microbenchmarks encounter slowdowns, it likely means that the microbenchmark suite can detect more real performance regressions. This has some similarities to the methodology called mutation testing, which can be used to evaluate unit test suites capabilities [15]. The size of the artificial slowdowns that are inserted has a large effect on the results and can be used to find different things. Smaller slowdowns can be used to evaluate how efficient benchmarks are at detecting performance regressions that are of a smaller size, while larger slowdowns can be used as a rough estimate of a whole microbenchmark suite's coverage capabilities as larger slowdowns are easier to detect. Artificial slowdowns are used in this thesis in order to evaluate and compare microbenchmark suites derived from static code analysis with randomly sampled suites.

2.2 JUnit-To-JMH

In this section *ju2jmh* is presented along with a brief introduction to JUnit.

2.2.1 JUnit

JUnit [16] is a popular open-source framework designed for writing and running tests for projects that use the JVM. JUnit allows developers to methodically write and execute tests to evaluate the functionality of their code. As a result, JUnit can help ensure the reliability and correctness of the code. In Figure 2.2, an example of a JUnit test case can be seen. Here, the functionality of the Calculator class is tested.

```java
import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class CalculatorTest {
    @Test
    public void testAddition() {
        Calculator calculator = new Calculator();
        assertEquals(15, calculator.add(10, 5));
    }
}

public class Calculator {
    public int add(int a, int b) {
        return a + b;
    }
}
```

Figure 2.2: Example JUnit test.

2.2.2 ju2jmh

This thesis project utilises a tool created during a study conducted by Alexanderson [10]. The tool named *ju2jmh* generates JMH microbenchmarks from existing JUnit tests. In the study by Alexanderson [10] they evaluated the quality of the generated benchmarks and found that the generated microbenchmarks are comparable to human written microbenchmarks in terms of stability. *ju2jmh* takes 4 arguments as input in order to run and create microbenchmarks, a list of these arguments can be seen below in table 2.1.

Table 2.1: *ju2jmh* launch arguments.

ju2jmh Launch Arguments
1. The root directory of the JUnit files.
2. The root directory of the compiled class files for the JUnit tests.
3. The root directory where the resulting benchmark source files should be generated.
4. A list of fully-qualified test class names to generate benchmarks from.

As seen from the list above, the arguments are not complicated, but something to take note of is that *ju2jmh* cannot convert individual JUnit tests into microbenchmarks; instead, whole test classes and all underlying tests are converted into microbenchmarks. This is, however, not a large issue, as JMH allows the user to specify which microbenchmarks to execute. In figure 2.3, a microbenchmark that was created from the JUnit test shown in the previous section with *ju2jmh* can be seen. This microbenchmark will run the JUnit method several times and provide a performance score.

```
@org.openjdk.jmh.annotations.Benchmark
    public void benchmark_testAddition() throws
    java.lang.Throwable {
        this.createImplementation();
        this.runBenchmark(this.implementation()::
        testAddition,
        this.description("testAddition"));
    }
```

Figure 2.3: Example *ju2jmh* benchmark.

2.3 Software Code Metrics

In the following sections some of the metrics and code characteristics that are used in this thesis are briefly introduced.

2.3.1 Code Coverage

There are several types of code coverage that can be calculated, including metrics such as line coverage, statement coverage, and branch coverage, to mention a few. In this work line coverage is the type of coverage being estimated and used. Code coverage is one of the most common metrics that is used for determining the quality of test suites [17]. The code coverage metric checks which parts of the source code a test suite runs. It helps to find parts of the source code that are not well covered by the test cases. Code coverage is most often calculated dynamically by executing the test suite and monitoring which parts of the code get executed. For instance, JaCoCo [18], a popular code coverage tool for Java, injects small pieces of code that are called instrumentations into the Java bytecode of the programme as it executes in order to log which parts of the source code get executed from a test suite.

2.3.2 Lines of Code (LOC)

In software engineering, lines of code (LOC) is a fundamental metric and commonly cited metric [19]. While being a seemingly simple and straightforward metric, LOC can provide valuable insight regarding several aspects about software development and maintenance. Research has shown that the number of faults per line of code increases with the size of the program [20]. LOC can be measured in several different ways. Two common ways is physical line of code and logical lines of code. Physical line of code counts every line in the code including white spaces, brackets and comments. Logical lines of code count only executable statements which means that elements such as white spaces and comments are not counted. Although LOC is a commonly used metric, it has a couple of known flaws, such as being bad at predicting code complexity as well as being a bad predictor of software quality. Furthermore, using LOC to predict code coverage is an optimistic approach when the LOC is estimated in a static context, as the actual dynamic execution path will likely execute a significantly smaller amount of physical LOC. For instance, if a test case is statically estimated to cover a specific amount of LOC of the underlying source code, the amount of LOC being executed during runtime will, in most cases, be significantly less than the statically estimated value.

2.3.3 Computational Complexity

Computational complexity is a metric for how much computing time is required to solve different problems. It can be measured in different ways. One of the more common ways to measure the complexity of code and algorithms is to calculate the Big O Notation, which is a measure of the worst-case compute time for algorithms. Big O quantifies algorithms' efficiency by describing how the runtime of an algorithm grows based on the input size given to the algorithm. Some common Big O measurements are O(1), O(n), and O(n^2). O(1) means that the algorithm is not affected by the input size. O(n) means that the time complexity is directly proportional to the input size. O(n^2) indicates that the time complexity grows quadratically with the input size. A concrete example of O(n^2) time complexity is an algorithm with two nested loops, each iterating over the input list.

2.3.4 Cyclomatic Complexity

Cyclomatic complexity is a software metric that can be used to try to quantify the complexity of code. It is most commonly calculated by counting the number of linearly independent paths of a programme's code using a control flow graph. It was introduced in 1976 by Thomas J. McCabe, Sr. [21]. A higher cyclomatic complexity indicates that the code is likely harder to understand and maintain, therefore being more error-prone [22].

2.4 Optimising Test Suites

In this section, two common approaches for optimising test suites are described. In this study, test case minimisation has been utilised. However, it is important to know the difference between these two.

2.4.1 Test Case Prioritisation

Test Case Prioritisation (TCP) is a software testing technique that aims to enhance the effectiveness of a test suite. This is done by methodically reordering the execution order of specific test cases. An example prioritisation strategy is Risk-Based Prioritisation meaning that High-risk features, which could be those that are newly developed, have had significant changes, or are crucial to the application's operation, are tested first. Another prioritisation technique is to use historical data in order to execute the tests that historically have been more prone to fail. There are many ways to prioritise and the examples mentioned are just two of them. The aim with TCP is to execute tests that are more likely to detect faults early, which can enable faster feedback and better allocation of testing resources. In other words, TCP can help meet important constraints in software testing, such as time and budget [23].

2.4.2 Test Case Minimisation

Test case minimisation (TCM) is another software testing technique that also strives to mitigate the effects of different constraints, such as time. The purpose of TCM is to generate a subset of test cases that still match the requirements of the full test suite with the smallest number of test cases possible [24]. As a result, unlike TCP, TCM discards tests that are not deemed valuable enough to execute. Due to this, TCM has lesser capabilities to detect faults compared to TCP as it does not execute full suites [25]. TCM uses methods like coverage analysis to intensify the minimal set of tests that cover the code paths, conditions, or requirements as the full suite does. Furthermore, tests that might not be valuable to execute can, for instance, be test cases that cover code that has not changed in a long time.

3

Related work

The literature regarding JMH microbenchmarking is relatively sparse. Several papers have explored problematic aspects of JMH. However, none has explicitly focused on generating minimised microbenchmarking suites using *ju2jmh* and statically derived metrics. Some papers have focused on reducing the execution time of suites using other methods. A common characteristic for many of these papers is the desire to integrate performance testing in CI/CD [5] [6] [10].

Laaber and Leitner [5], proposed a technique where benchmarks are dynamically stopped when their results have reached adequate level of stability. As a result, the authors managed to significantly reduce execution time of several suites (48.4% - 86.0%) while maintaining the same level of quality for the large majority of the benchmarks (78.8% - 87.6%). This is quite different from the aim of this thesis project which was to minimise the amount of benchmarks that are executed not minimise individual benchmarks execution time which was what Laaber and Leitner [5] focused on.

One of the main features of JMH is the option to specify user-defined parameters to specific microbenchmarks. For instance, in a microbenchmark the developer might find value in running the benchmark several times with different variable values. Samoaa and Leitner [26] lift one problematic aspect of this parameterisation. For each parameter and the different values assigned to these the benchmark need to be executed m^n times where m is the different parameters values and n is the number of parameters. As a result, even minor amount of parameterisation can lead to a immensely increased combinatorial workload. At the same time, the authors also show that 40% of these user-defined parameters does not have a correlation with the results of the microbenchmarks. Furthermore, the authors also quantified the importance of different parameters. The results from this was that one parameter often dominates in terms of importance.

Traini et al. [14] concluded that a lot of developers that use JMH for benchmarking Java fail to estimate how long it takes for the Java Virtual Machine to reach a steady state. This relates to the fact that the JVM performs optimisations while the application is running, and these optimisations can greatly affect the results of the benchmarks, making the results have high variability [13]. In addition to this, Traini et al. [14] found that some microbenchmarks never reach a steady state, meaning that the results from those benchmarks are always more likely to report false positives and be less useful. The main goal for Traini et al. [14] was to investigate the stability of JMH benchmarks overall whereas the goal of this thesis project was more aimed at investigating if smaller generated suites can be used to detect performance regressions reliably.

Laaber et al. [27], used machine learning utilising 58 different statically-computed source code features to predict if a benchmark is going to be stable. They used 11 different binary classification models. Their Random Forest classifier performed best with a prediction performance between 0.79 and 0.90, concluding that static source code features can be used to predict benchmark quality. Laaber et al. [27] also performed a feature importance analysis, that found 7 features to be especially important to their predictions, and these features can be evaluated as possible metrics to evaluate benchmark quality. Additionally, Laaber et al. evaluated the results of applying test case prioritisation techniques to software microbenchmark suites [28]. They concluded that applying test case prioritisation can work and recommended dynamic techniques, but in cases where time is sparse, such as in a CI pipeline, static prioritisation techniques may be preferred as the overhead is significantly lower [28].

Laaber et al. [6] also investigated several open-source software (OSS) microbenchmark suites in terms of their capability in providing feedback for continuous integration. In the study, 10 OSS projects using Java and Go are investigated. They find great variability in benchmark suite capability between these projects in terms of the ability to detect performance regressions. They utilised artificial slowdowns to evaluate the capabilities of the microbenchmark suites [6].

Path [29] applied different optimisation approaches to *ju2jmh* in order to generate minimised microbenchmark suites. No generalised tool was, however, created, which makes it hard to evaluate the results presented in the study. In the paper, it is suggested that random selection for microbenchmark generation from JUnit tests consistently produced suites that had high code and branch coverage. Path also concluded that the best optimisation approach was prioritising the highest branch and code coverage, as this produced suites with good stability based on the results presented in the thesis [29]. Path mentions that static analysis was used but this is not the case as it is mentioned several times that JUnit tests were executed and tools such as Jacoco [18] were used.

Costa et al. [9] studied 123 open source projects and identified five common bad practices, such as not consuming the product of a computation and incorrect usage of JMH forks. They conclude that these types of bad practices are common in many Java-based open source systems, and they also showed that the five bad practices described have a large effect on the results of the microbenchmarks. They state that the results of their study indicate that many open-source developers struggle to understand the intricacies of benchmarking for Java.

Unlike all the related work mentioned above, the aim of this thesis project was to see if small suites generated from JUnit tests based on statically derived metrics related to the characteristics mentioned in the research questions can reliably detect performance regressions. Investigating if tools such as *ju2jmh* can be used to make performance testing more accessible and effective for software developers. This thesis also builds upon the work by Alexanderson [10] as the feasibility of generating minimised suites from JUnit tests was evaluated. Additionally, the evaluation process used also provided some insight into the coverage of the generated microbenchmark suites while also taking into account the stability of the overall suites.

4

Methods

In the sections below the methods used are explained in more detail. This study is best described as a laboratory experiment as the variables are to be considered very controlled and the setting is highly artificial [30].

4.1 Overview

In this section a brief overview of the method is presented. In Figure 4.1, an overview of the method can be seen. The first step refers to the process of selecting projects with JUnit tests. After selecting projects, the JUnit tests were estimated and measured in terms of the code characteristics mentioned in the research questions using proxy metrics that were calculated by recursively parsing the source code methods being called in the test method. The reasons why these metrics were chosen were primarily due to their overall popularity in software engineering and their implications for a programme's behaviour. After estimating for one of the approaches using a metric, all the JUnit tests in a project are ranked according to the metric, and the highest-ranked JUnit tests are used to generate a microbenchmark suite with the JMH minimisation tool. The generated microbenchmark suite was then executed to gather results, but in order to be able to evaluate the results, five other randomly generated suites of the same size were executed as well. The evaluation process itself also involved the injection of artificial slowdowns to evaluate the suite's capabilities in terms of detecting performance regressions and overall stability. This means that the metric-based suite and the randomly sampled suites were executed before and after artificial slowdowns were injected. The data gathering process was repeated with three different sets of artificial slowdowns combined with five new randomly sampled suites for each configuration, meaning that it was repeated three times for each project and slowdown length. The main reason that five randomly generated suites were used for each set of artificial slowdowns and that data gathering was done three times per configuration was primarily because of time constraints. Given more time, more data could have been collected.

4.1.1 Sensitivity Analysis

In order to analyse how the results vary depending on different parameters and settings, a sensitivity analysis was conducted. The primary parameters and factors that

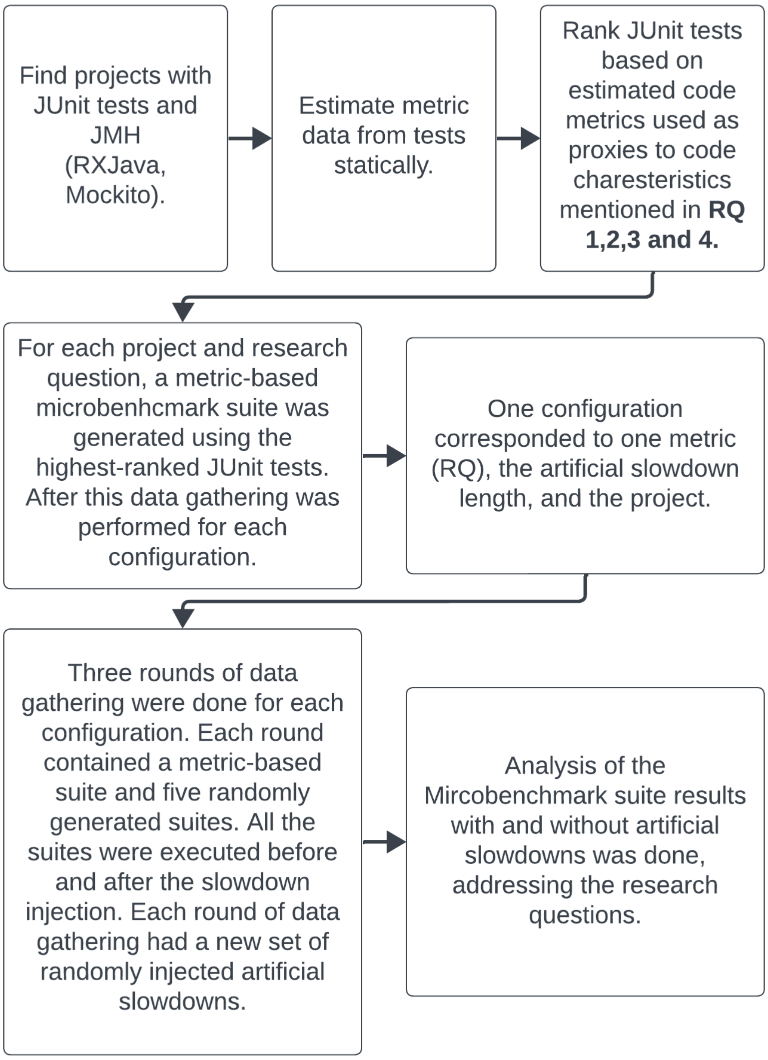

Figure 4.1: Overview of the method.

were considered were artificial slowdown duration and artificial slowdown locations. These will be described in more detail in the following two sections. It is important to note that, due to the extensive time it takes to run a single microbenchmark suite, sensitivity analysis quickly becomes infeasible. Due to this, the number of parameters and corresponding values that are tested is limited.

4.1.1.1 Artificial Slowdown Duration

Artificial slowdown duration is the performance regression of a single slowdown. More especially, the amount of time that the slowdown forces the thread to sleep. This parameter was interesting to analyse deeper, as one can simulate various degrees of performance degradation and see how these affect the end result of the generated benchmark suites. For this parameter, two different approaches were chosen: an extreme value and a smaller value. In the first approach, slowdown regression will inject an extreme performance regression (10 ms). This was useful, as it provided insight into how much coverage the suites provided. If a microbenchmark encountered a slowdown that was severe, then it would show in the results. This could then be used to get an overview of a suite's potential coverage capabilities and also help estimate how many slowdowns should be injected. The second approach to artificial slowdown duration was smaller slowdowns (1 ms). This was an approach that was aimed to be more representative of the performance regressions that occur during the development of software systems. The smaller slowdown value was used to evaluate the actual applicability of the microbenchmark suites in a more realistic setting. 1 ms of artificial slowdown is quite a large slowdown, given that execution times can be a lot lower for a simple Java test method. This means that the main takeaway from the findings will likely be more indicative of coverage. However, by evaluating the stability of the generated suites in combination with their coverage capabilities, a more clear result of how capable the suites are at discovering "real" performance regressions is created.

Execution times for the JUnit test methods used to derive microbenchmarks in this thesis varied between 300 nanoseconds and 19000 milliseconds, where simple methods lie on the lower end and more complex methods are on the upper end. In Table 4.1, the minimum, maximum, median, and average execution times for 15000+ microbenchmarks used during this thesis are presented. Here, one can see that there is a significant difference in magnitude between the minimum and maximum execution times. Furthermore, the difference between the median and average execution times is substantial. This indicates that several outliers exist, which drastically affect the average execution time.

Table 4.1: Execution times for microbenchmarks derived from JUnit tests (RxJava and Mockito).

Minimum execution time	**Maximum execution time**	**Median execution time**	**Average execution time**
0.000334 ms	19076.491970 ms	0.020446 ms	55.610603 ms

4.1.1.2 Artificial Slowdown Locations

Since the artificial slowdowns were injected into randomly selected methods, several runs with the same parameters but with varying slowdowns were conducted. The purpose of this was to investigate how the results differed depending on which methods were injected with artificial slowdowns.

4.2 Microbenchmark Optimisation through Static Analysis

In this thesis, minimised microbenchmark suites containing 50 microbenchmarks were generated based on four different code characteristics: code coverage, computational complexity, code complexity, and finally, a combination of the three aforementioned code characteristics. In order to generate suites based on these three approaches in a static code analysis context, they have to be measurable. To do this, a separate metric that could be calculated during static analysis was used for each code characteristic. The primary reason why static code analysis was conducted was simply because it was estimated that dynamic analysis would lead to much more overhead, and given that the problem statement of this thesis revolves around the fact that time is a precious commodity, saving time in terms of the parsing of the source project was important as well.

The measurements were done by utilising metrics that were either estimations or proxy data related to the approaches mentioned. The main reason for this was that the specific dynamic execution path of code is generally hard to derive without executing the code, as the initial input to a programme can change the execution path drastically. This means that the code coverage, computational complexity, and code complexity calculated for a JUnit test method are likely rough estimations, as it is generally impossible to derive which code is executed. Therefore, estimations or proxy data for these were calculated instead. The kinds of metrics used are discussed in the sections below.

4.2.1 Shared Aspects Between Metrics

The implementation of all metrics described in this study shares several implementation details. The significant difference between these metrics is how they evaluate individual source code methods. Other than that, they are all implemented the exact same way. In Figure 4.2 below, one can see an example of a JUnit test. Each metric would start at the test method and see which source code methods are being called. After this, they would go into each source code method and see what methods were being called there, and they would repeat the process until they reached the end of the method chain. External methods belonging to third-party libraries (e.g., java.util) are not taken into account. The data gathered is then used to form the aggregated value for the specific Junit test and metric being used. This aggregated value is then used during the ranking of JUnit test cases.

```java
public class ExampleTestClass {
    ...
    @Test
    public static void test_calculate() {
        processedData = ExampleClassA.calculate();
        ...
    }
}
public class ExampleClassA {
    ...
    public static List<Data> calculate() {
        myData = ExampleClassB.getData();
        ...
    }
}
public class ExampleClassB {
    ...
    public static List<Data> getData() {
        ExternalClass.doSomething(); // Method not counted.
        ...
    }
}
```

Figure 4.2: Example JUnit test case and underlying methods being called.

4.2.2 Code Coverage

Code coverage in terms of performance microbenchmarking relates to a benchmark's ability to execute areas of the code containing performance regressions. The hypothesis was that if a benchmark is estimated to have high code coverage, then it will be more likely to cover areas of the code where performance regressions can be introduced. The LOC metric was used as a proxy for code coverage (RQ1), as the idea was that a method that has a high statically calculated LOC would indicate that more code would be executed during execution. This is now known to have been a very naive approach in that it is highly dependent on the source code, and the calculated LOC will almost always be drastically higher than what is actually executed. With more prior knowledge another approach to estimating code coverage would likely have been used.

LOC was the first metric used to create a minimised microbenchmark suite. The reason that this was the first metric was because it is one of the metrics that are easier to estimate through static code analysis. The hypothesis is that executing and prioritising microbenchmarks that are estimated to be able to execute the most LOC would allow for wider detection of performance regressions than a random sample of the JUnit test converted to microbenchmarks. The LOC value for each method was calculated by counting the physical lines of code, excluding any documentation. Furthermore, Laaber et al. used test case prioritisation techniques related to code coverage and applied them to JMH microbenchmark suites. They found techniques related to code coverage to be effective, and they also concluded that greedy total coverage techniques that do not account for overlapping coverage between microbenchmarks performed better than additional coverage techniques that do account for overlapping coverage between microbenchmarks when optimising microbenchmark suites [28]. Using LOC estimated from static analysis as a proxy for code coverage was and is a naive approach because any estimations made are likely to be way off in terms of how much LOC is actually executed, the reasons for this are discussed at the start of Section 4.2. However, static analysis provided a method that would provide overall less overhead in terms of calculation and execution time needed, and time was one of the primary constraints to consider during the planning of the data gathering process.

With the help of Figure 4.3, one can visualise how the loc metric would form the aggregated value for these methods. In essence, the metric counts physical lines of code, excluding any documentation. So, in this case, the first method would be valued at three (method signature, code block, and closing bracket). Any documentation will not be counted. Furthermore, since the first method calls an internal method (belonging to the same project), this method will also be counted. This method will be evaluated at four, just like the previous method. As a result, the loc metric would form an aggregated value of seven in this case.

```
public static int do_something(int a, int b) {
    // Comments are not counted.
    return do_someting_else(a + 10, b + 10)
}
public static int do_something_else(int a, int b) {
    int c = 10
    return (a + b + c)
}
```

Figure 4.3: A simple method used for visualising loc metric.

4.2.3 Computational Complexity

Computational complexity is related to the amount of resources required to run specific algorithms or parts of code. The reason that the computational complexity was interesting to investigate was because the hypothesis was that microbenchmarks that execute code with larger worst-case execution times are the places that are more vulnerable to performance regressions. In addition to this, it was also interesting to investigate the relationship between computational complexity and microbenchmark stability.

The loop count metric was used as a proxy for computational complexity (RQ2). In order to calculate and estimate the computational complexity of a JUnit test's underlying code, the approach used was to count the number of loops that the test method could execute. As it is often hard to know the exact execution path, this, in the end, is just an estimation of the actual number of loops that will ultimately be executed. Regardless of this, if the estimated number of loops that a JUnit test can execute is high, then it has potential for the execution of more loops. The reason that loop count is used to estimate computational complexity is because algorithms that have a large number of loops have a potential for high computational complexity.

In Figure 4.4, two methods containing loops can be seen. The first method calls the second method inside its nested loop. Here, the metric would classify this as having a aggregated value of three. Two from the first loops in the first method and one for the loop in the second method.

```
public static void do_something(int a, int b) {
    for (int i = 0; i < a; i++) {
        for (int j = 0; j < b; j++) {
            do_something_else(a, b);
        }
    }
}
public static void do_something_else(int a, int b) {
    for (int i = 0; i < a; i++) {
        ...
    }
}
```

Figure 4.4: A simple method used for visualising loop metric.

4.2.4 Code Complexity

The reason that code complexity was interesting to investigate is because if the code in function or programme is very complex, it suggests that the code is likely harder to maintain and more error-prone [22]. Additionally, complex code may suggest that the code contains many for and while loops, which can greatly impact the execution time.

Code complexity can be estimated in numerous ways, some of them can be estimated statically. This includes metrics such as cyclomatic complexity and Halstead complexity, among other ways to calculate code complexity. Halstead complexity was considered, but cyclomatic complexity was chosen as it is a commonly used and more straightforward metric to implement. While the previously mentioned computational complexity metric could be considered a subset of the cyclomatic complexity metric, there is an important aspect to consider. For instance, a test case can have a high estimated cyclomatic complexity while only containing one or no loops, while some test cases can have a high loop count but overall low cyclomatic complexity. If you were only to use the cyclomatic complexity metric, there is a risk that test cases with a relatively high loop count and a low amount of conditional statements could have lower cyclomatic complexity than test cases with a high amount of conditional statements, and hence the effects of a high loop count could be missed.

The cyclomatic complexity metric was calculated by calculating the JUnit test's total cyclomatic complexity, meaning it took into account the underlying source code methods that could be called while executing the JUnit test. Traditionally Cyclomatic complexity is calculated using a control flow graph and the formula below:

$$M = E - N + 2P$$

.

- M: Represents cyclomatic complexity.
- E: E corresponds to edges.
- N: Number of nodes in the flow graph.
- P: Number of connected components.

The formula presented above is not entirely straightforward to implement a calculation for. Consequently, a more simplified version was used for the purposes of this project. The formula that was utilised for calculating the cyclomatic complexity of a Java method was:

$$M = D + 1$$

An explanation of the variables can be seen below.

- M: Represents cyclomatic complexity.
- D: Is the number of decision points within the method. Decision points include `if, else if, for loops, while loops, switch case statements, and occurrences of logical operators such as && and ||.`

Both formulas basically calculate an integer representing the cyclomatic complexity but the latter one is the one that was used in the implementation of the cyclomatic complexity metric. It is also important to remember that the implementation sums up all the source code cyclomatic complexity that could be called from the JUnit method being calculated for meaning that the formula is used to create a total estimated cyclomatic complexity. The reason that cyclomatic complexity was used to estimate code complexity is because it is one of the common metrics used to estimate the complexity of code [31].

In Figure 4.5, an example of cyclomatic complexity can be seen. Here, the metric would classify this as having a complexity of four. The first if-statement provides one complexity. The second if statement provides two, one from the if statement and one from the and statement. So, in total, the method has four plus one complexity, where one is the base value. Furthermore, if the method would have called another internal method the complexity of this method would have been counted as well in order to form the aggregated value.

```
public static void do_something(int a, int b) {
    if (a == b) {
        ...
    }
    if(a > b && a > 0) {
        ...
    }
}
```

Figure 4.5: A simple method used for visualising cyclomatic complexity.

4.2.5 Combining the Metrics

In addition to ranking the JUnit tests that are to be used for creating microbenchmarks using a single metric, a combination of the metrics for code coverage, computational complexity, and code complexity was used. This was done by calculating the values for each metric and then normalising them. The normalised metric values could then be summed up into one value. The sum of the normalised metric values was then used to rank individual Junit tests. All of the metrics had the same weight, meaning that each metric had as much effect on the ranking as the other metrics. The reason that the same weights were picked for all the metrics is because there was a lack of time for doing sensitivity analysis to see which weights performed best.

4.3 Evaluation of Minimised JMH Suites

In this section, the process used to evaluate the quality of the minimised microbenchmark suites is explained. Each time a JMH microbenchmark suite is executed, a report is created containing the score of the individual microbenchmarks and a 99.9% error score that can be used to evaluate the stability of the microbenchmarks. The report data is what was used to evaluate and compare the results from different suites. All of the suites discussed were generated using *ju2jmh* and contained 50 microbenchmarks.

4.3.1 Environment

In order to ensure non-volatile results for the microbenchmarks, two virtual machines (VMs) were utilised. These machines were hosted through Google Cloud [32]. More specifically, two identical VMs were used. The hardware for each was two dedicated CPU cores running on an Intel Broadwell CPU with 16 gigabytes of memory and 50 gigabytes of storage. Furthermore, Debian 12 [33] was used as the operating system of the VMs. The reason why two VMs were used is because comparing the effects of artificial slowdowns using metric-based test case minimisation and random sampling of microbenchmarks can take several hours up to several days, depending on the parameters used, rendering local execution infeasible.

4.3.2 Study Objects

In the following subsections, the two GitHub repositories used for test case minimisation will be described. These repositories were selected through purposive sampling [34]. Purposive sampling is a method used to select specific, useful and interesting research participants, which in this case were OSS github projects. This was done to get clearer and more useful results. During the sampling for repositories, several factors were considered. This includes popularity, occurrence in other scientific papers, quality of the JUnit test suite, and the usage of JMH. More specifically, the popularity of a repository was measured by the number of stars it had on GitHub and the quality of the test suite was measured by checking code coverage and the number of tests. To find an initial set of repositories, two primary methods were used. The first method was to look at repositories used in other papers related to JMH. The second method was using the search functionality and searching for code specifically used in build tools such as Maven or Gradle to resolve the dependency of JMH which made it easy to find projects that utilised JMH. From this process, an initial set of 20 repositories was collected. From this set two final repositories were derived. These were derived using the previously mentioned aspects. During the process of selecting two final repositories the aim was to try select two repositories were these aspects were balanced. The reason that only two were selected is because time is the main limiting factor for this thesis project, and committing execution time to more than two projects seemed unfeasible given the resources that were available. The projects picked are mentioned briefly in the sections below.

4.3.2.1 RxJava

RxJava [35] is a library created for handling asynchronous tasks and events for programming languages that target the JVM. The project has over 45 thousand stars on GitHub. Furthermore, the test suite contains several thousand JUnit tests. Additionally, it has a microbenchmark suite that utilises JMH. RxJava has over 8 thousand source code methods and over 12 thousand JUnit test cases.

4.3.2.2 Mockito

The second repository used is the mocking framework Mockito [36]. Mockito offers mocking capabilities for unit tests in Java. Mocking is the process of creating simulations of classes in programming languages such as Java that can be used to test how the objects behave in various scenarios without having to modify the actual implementation of the class. For instance, Mockito can help track how many times methods within a class are being called and what they should return if they get called, which allows for extensive control of the behaviour of the class. Mockito has over 2 thousand source code methods and also more than 2 thousand Junit tests. Furthermore, Mockito has over 14 thousand stars on Github.

4.3.3 Detecting Performance Regressions

To determine whether a microbenchmark was likely to have encountered a slowdown, a mathematical approach was employed. This approach was designed to account for the stability and results of a benchmark both before and after the injection of artificial slowdowns. A straightforward yet effective formula was developed to quantify the impact of injected slowdowns on benchmark results. The score of a benchmark for this project is recorded as a throughput score, which is a count of the number of times the benchmark method could execute per second. This throughput score could then be used in combination with the 99.9% score error which is calculated from the measurement iterations of the benchmark. The initial step involved calculating the total difference in throughput scores (Δ_{Score}), termed Score Delta, before and after the injection of slowdowns, as shown below.

$$\Delta_{\text{Score}} = T_{\text{Score}_{\text{before}}} - T_{\text{Score}_{\text{after}}}$$

The next step was to account for the possibility that the benchmark may be unstable. Therefore, the combined stability before and after the slowdown injection was considered. This is done by summing up the total 99.9 % score error confidence interval of the benchmark results into a combined error (CE) from before and after the injection of artificial slowdowns. The formula for the combined error score can be seen below.

$$CE_{\text{Combined 99.9\% Score Error}} = E_{\text{99.9\% score Error CInt before}} + E_{\text{99.9\% score Error CInt after}}$$

The last step is to use the combined error (CE) and the score delta in combination. If the combined error from the benchmark is high, it indicates that the benchmark is unstable, but if the score delta is much larger than this combined error, then it is likely that the benchmark encountered one or more artificial slowdowns. Furthermore, to further decrease the risk of false positives when assessing if a microbenchmark was affected by one or more slowdowns, the combined error (CE) was multiplied by a factor of 5. This factor was derived by experimenting with different values. This process is described in detail in Section 4.3.3.1. So, a benchmark is assumed to have encountered a slowdown if the total difference in score (Score Delta) from before and after artificial slowdown injection is greater than the combined error multiplied by 5, as seen below.

$$\Delta_{\text{Score}} > CE_{\text{Combined 99.9\% Score Error} * 5}$$

The main reasoning behind this formula is related to the fact that it accounts for the stability of the benchmark both before and after injection. Using the combined error means that the microbenchmarks that are assumed to have encountered a slowdown have accounted for the stability of both before and after artificial slowdowns, meaning that the assumption should be conservative. Even so, the formula above is not perfect as it cannot be trusted to classify each individual microbenchmark correctly, but it does allow assessment on a whole-suite basis, which is going to be the use case for this particular formula.

4.3.3.1 Mathematical Model Sensitivity Analysis

Sensitivity analysis was conducted on the formula presented in Figure 4.3.3. The primary reason was to investigate how to limit the number of false positives. False positives are the number of microbenchmarks where the difference before and after slowdowns was most likely caused by the instability of the microbenchmarks and not due to artificial slowdowns. One problem is that running the same benchmark suite twice can lead to quite varying results. There are two primary factors that contribute to this. These factors are mainly hardware and operative system factors on the machine where the microbenchmarks run. In Figure 4.6, the results from executing the same benchmark suite can be seen. Here, one can see that most of the microbenchmarks have very similar results. However, some of them vary substantially. This variation is most likely caused by the instability of microbenchmarks or factors related to the virtual machine and underlying hardware.

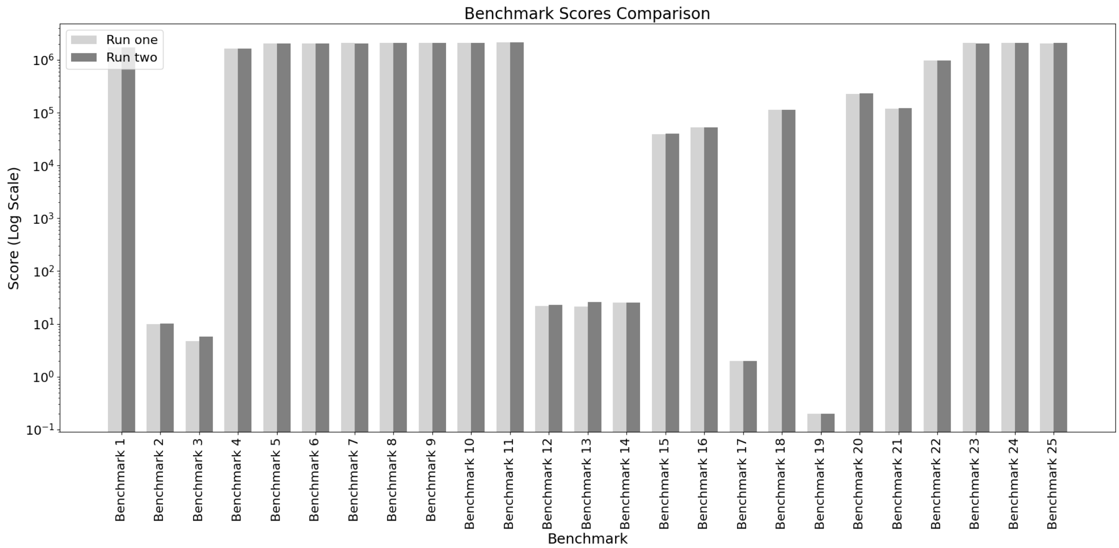

Figure 4.6: Same benchmark suite executed two times.

In Figure 4.7, the results from running one benchmark suite with and without artificial slowdowns. Here, it is easy to see that the changes compared to 4.6 are more drastic. It is these more drastic changes that the formula is aimed to capture.

In order to only capture the more drastic changes between microbenchmarks, several different factors were applied to the final formula seen in section 4.3.3 before choosing 5. This value means that the results between two microbenchmarks before and after artificial slowdowns must differ more than five times their combined score error in order to say that one microbenchmark was affected by slowdowns. Different values

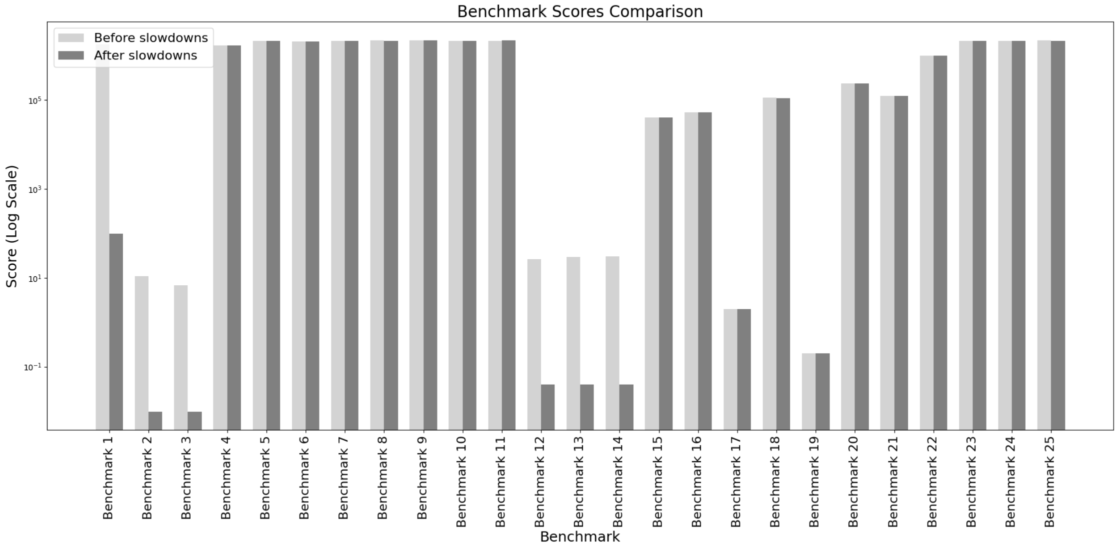

Figure 4.7: Before and after slowdowns.

of the factor were tested, and the results were thoroughly analysed. This was done by executing several suites, like in Figures 4.6 and 4.7, and seeing what factors led to the fewest number of false positives of artificial slowdowns. The counting of false positives was done by plotting these suites, manually deriving the number of drastic changes, and comparing this with the results obtained by applying the formula to the microbenchmarks. The values tested were 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, and 20. From this analysis, from 5 and up, the results were stable. Meaning that values greater than 5 provided the same results as 5. With the value 5, only the more extreme changes between microbenchmarks were captured, not the changes that could have been caused by the instability of the benchmark or factors related to the virtual machine and underlying hardware.

4.3.4 Minimised and Random Microbenchmark Suites

Two types of microbenchmark suites were used to collect and analyse the results from test case minimisation using the previously described metrics. The suites generated are in the sections below.

4.3.4.1 Minimised Microbenchmark Suite

The minimised suite is the suite generated by utilising the JMH minimisation tool in combination with a metric. For each OSS repository a minimised suite was created per metric. The suite's capabilities was then compared to the random sample microbenchmark suites described below.

4.3.4.2 Randomly Sampled Microbenchmark Suites

To assess the quality of a metric-based microbenchmark suite, it was compared to several sets of randomly sampled microbenchmark suites generated from the JUnit tests present in the OSS projects used. Each time a metric-based microbenchmark suite was evaluated, it was evaluated against five randomly sampled microbenchmark suites of the same size from the same OSS project. The reason why five randomly sampled suites were collected was the result of trying to balance the time constraints of this project and the need to collect data from several suites in order to more accurately capture how the results vary between suites. It was deemed that five suites provided an adequate amount of data, while the computational time needed to collect this data was feasible given the time constraints. This was done by adding all the JUnit tests present in the OSS project to a list and picking out random unique tests from that list until the suites matched the metric-based minimised suite size. The reasoning for using randomly generated microbenchmark suites as baselines was that it was closely related to the context of the thesis, which is generating JMH benchmarks from JUnit tests. Using already existing JMH microbenchmarks in manually created microbenchmark suites could have been done. However, using manually created JMH suites is a lot more complicated in terms of data collection, as these suites vary in size and can be quite large, and some benchmarks have very long execution times depending on their setup. Therefore, randomly sampling JUnit tests to use as JMH microbenchmarks for baseline was much more feasible and less time-consuming.

4.3.5 Artificial Slowdowns

In order to evaluate the abilities of the minimised microbenchmark suite and random sampling suites, artificial slowdowns were introduced into the OSS projects that were used. These artificial slowdowns were injected using JavaParser. The slowdowns were injected into random methods present in the source code of OSS projects. The injected slowdowns utilised the Java *Thread.sleep* method [37]. This had been used before in literature and is one way to introduce artificial slowdowns [6].

The number of artificial slowdowns that are randomly injected into the source code naturally affects the results as well. It was found in the early stages that injecting around one percent of the source code methods with artificial slowdowns was a good, balanced amount of slowdowns to inject. The reason for this is that the injection of too many slowdowns makes it hard to interpret the results as too many microbenchmarks execute artificial slowdowns. And on the other hand, injecting far too few slowdowns has the inverse effect, where none of the microbenchmarks may encounter an artificial slowdown, making the results more or less the same before and after injecting slowdowns. In short, one percent artificial slowdowns were good as they showed how the different microbenchmarking suites were characteristically different from one another and allowed for more nuanced comparisons.

4.3.6 Suite Evaluation process

In order to evaluate the capabilities of the metrics used for RQ1, RQ2, RQ3, and their corresponding microbenchmark suites', a thorough process was used. For each metric, project and slowdown length, three rounds of data gathering were conducted, with a different set of slowdowns for each round as well as five randomly sampled microbenchmark suites to compare against in each of these rounds. In each of these rounds, the first step is to generate all of the executables containing the microbenchmark suites, a list of them can be seen in Table 4.2.

Table 4.2: Executables needed for one round of data collection for one metric and set of slowdowns.

Before Slowdown Injection	**After Slowdown Injection**
Five randomly sampled microbenchmark suites without artificial slowdowns	Same five microbenchmark suites with artificial slowdowns
One microbenchmark suite based on a metric without artificial slowdowns	Same microbenchmark suite based on a metric with artificial slowdowns

As seen in Table 4.2, a number of different suites were needed for each round of data gathering for each metric. The execution time for each suite was roughly 180 minutes when it managed to execute correctly, which means a total execution time of around 36 hours for one round of data gathering when everything worked. If something did go wrong, the suites that failed had to be executed once again. Sometimes the execution time of a suite was increased significantly because of some edge-case microbenchmarks for which the origin JUnit tests handled concurrent functionality, which was prone to more errors, especially after artificial slowdowns had been injected. To try to remedy this some JUnit tests were not included while calculating metrics and generating microbenchmarks, this was done by removing specific JUnit tests that refereed to concurrency testing.

In the end, the three rounds of data gathering needed for one metric took around 108 hours for one project. Furthermore this was multiplied by two as two different slowdown duration's were tested during sensitivity analysis. So, after sensitivity analysis had been done the time had increased to 216 hours for one metric and one project with sensitivity analysis. Two projects were used so the total execution time for all the data gathering needed for one metric took around 432 hours or 18 days. However, it is paramount to note that this calculation was strictly theoretical and not very realistic as it only accounted for the expected execution times of the microbenchmark suites and it assumes that nothing went wrong. In practice, several benchmark suites failed to fully execute which in turn needed to be rerun which caused even more time overhead. For instance, in many cases the suites failed to execute near completion which resulted in severe time lose.

Before introducing the artificial slowdowns, the different microbenchmark suites were executed without artificial slowdowns to get a baseline result. After that, the microbenchmarks were executed once more, but with the artificial slowdowns introduced into the OSS projects. The next step was to evaluate the abilities of the different microbenchmark suites (Minimised and Random Sampling). They were evaluated and compared in terms of their ability to detect the injected artificial slowdowns and their overall microbenchmark stability. The evaluation process involved comparing the baseline results gathered before introducing slowdowns with the results gathered after the slowdowns were introduced. If a microbenchmark has a throughput score that has remained largely unchanged, then it was assumed that it had not detected an artificial slowdown. If a microbenchmark throughput score had changed substantially and the stability of the microbenchmark was adequate, it was assumed that the microbenchmark covered an artificial slowdown. Before assuming that the microbenchmarks ability to detect performance regressions is good, both the baseline results and the results from executing with artificial slowdowns injected were investigated in terms of stability. The reason for this is that if the microbenchmark is not sufficiently stable, it may be the case that the microbenchmark results are just unstable, and the microbenchmark cannot reliably be used to detect performance regressions. Doing this evaluation on a individual benchmark basis was cumbersome and therefore a more streamlined process was used to evaluate individual microbenchmarks more detailed information about this can be found in Section 4.3.3.

As a result of the excessive amount of time needed to collect data, it was not the aim of this study to perform statistical testing on the collected data. This is simply because it would take several months to get a valid sample size. However, the aim was still to get as much data as possible, but instead of performing statistical testing, the plan was to assess more simple values such as mean and confidence intervals without drawing any statistical conclusions and instead see if the result could indicate that using the metrics described in this paper for test case minimisation could provide better microbenchmark suites compared to randomly sampled suites. Since the data samples contain fewer than 30 entries, when calculating the confidence intervals, a t-distribution was used.

For each microbenchmark configuration (e.g., RxJava, LOC, 1 milisecond artificial slowdowns), two suites were created. Both of these suites contained 50 identical microbenchmarks, where the only difference was that one suite had been injected with artificial slowdowns. One important aspect was that during the execution of these suites, sometimes not all microbenchmarks were able to fully execute. As a result, some suites used for data analysis contained less than 50 microbenchmarks. It was common that 1–3 microbenchmarks did not execute successfully.

4.3.7 Evaluating Stability

In order to evaluate the stability of the results from the microbenchmark suites that were executed, the mean coefficient of variation for the microbenchmark 99.9% error scores was used. This 99.9% score error is the +- 99.9% confidence interval of the throughput score calculated by JMH from the 10 measurement iterations. For each suite, the coefficient of variation was calculated based on the underlying microbenchmark results. The stability calculation for one microbenchmark can be seen below.

$$(\text{Microbenchmark 99.9\% Score Error interval / score}) * 100$$

This was done for each microbenchmark in a suite and was used to calculate the mean and median stability of the whole suite. The reasoning for this is that it provides a good overview of the overall stability of different suites and allows for comparative analysis as the mean coefficient of variation for a suite can be expressed as a percentage, and a lower percentage likely indicates that a microbenchmark suite is stable.

5

Implementation

In this chapter, the implementation of the tool created for test case minimisation of JUnit tests that are to be used in the generation of microbenchmarks is described in detail. This tool is available on GitHub [38]. Figure 5.1 illustrates a simple overview of the structure of the JMH Minimisation tool. From this, the seven main classes that help to build the tool can be seen. Several of these will be described in more detail in the following sections.

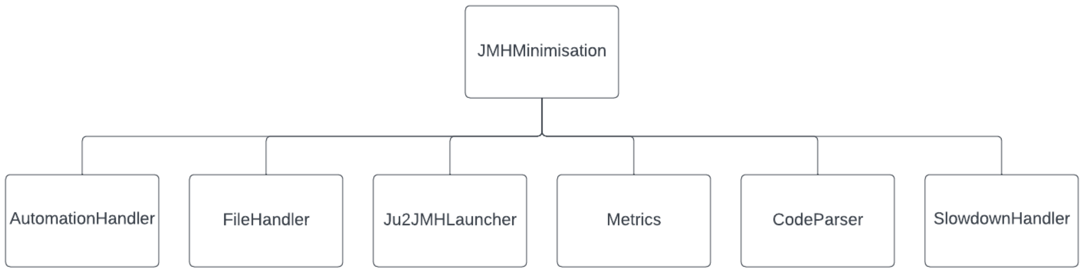

Figure 5.1: Overview of the JMH Minimisation tool.

5.1 Tools and Libraries used

In this section several tools and libraries used during the implementation of the JMH Minimisation is described.

5.1.1 JavaParser

JavaParser [39] [40] is a library that allows developers to interact with Java source code. This is done by object representation in a Java environment. This is also known as object representation as an abstract syntax tree (AST). Furthermore, JavParser offers several methods for traversing the source code tree, which allows developers to easily analyze the code without having to manually write extensive code. Additionally, JavaParser allows for symbol resolving. This means that JavaParser can find how different methods, classes, and packages relate, which can, for instance, be used when deriving types.

5.2 JMH Minimisation Tool

The JMH minimisation tool that was created for this project makes heavy use of the JavaParser library. The application needs to be configured prior to use. This configuration includes the path of the source code, the path of the JUnit test folder, paths to JAR files needed by JavaParser to do symbol resolving and lastly configuration for *ju2jmh*.

The application contains several classes that help form the core of the application. These classes are AutomationHandler, FileHandler, Ju2JMHLauncer, CodeParser, SlowDownHandler, and classes related to metrics such as the LOCMetric class. These classes will be described in greater detail in the text below.

On a higher level, the application works as follows. Firstly, the program starts with parsing the source code and JUnit test cases of the project. Secondly it performs Test Case Minimisation (TCM) on the JUnit test cases. This TCM is based on a single metric or the combination of metrics. Thirdly, it generates with the help of *ju2jmh* a benchmark suite of the JUnit test cases that were derived during the TCM. Additionally, in order to conduct evaluation of the benchmark suite the application can also inject artificial slowdown into the source code and create randomly sampled benchmark suites.

5.2.1 FileHandler

FileHandler is a simple class. It simply gets a start directory and traverses all files in that directory and any subdirectory. If a Java file is found, it adds the absolute path of the file to a list. As a result, it returns a list containing the absolute paths to different Java files.

5.2.2 Ju2JMHLauncher

Ju2JMHLauncher is a helper class that helps to connect the JMH minimisation tool created during this thesis and the *ju2jmh* tool created by N.Alexanderson et al. [10]. This class takes a list of JUnit test cases that are to be converted to benchmarks with the help of *ju2jmh* and creates a jar containing the specific benchmarks. Furthermore, it also returns a string that contains the relevant commands and arguments. This string can be copied and pasted into a terminal to execute the correct benchmarks with the same settings.

In figure 5.2 below, an example command that was created by the Ju2JMHLauncher can be seen. The first part of the command is basic Java syntax to execute a specific jar file. Followed by that, the JMH arguments are specified, and lastly, which benchmarks to execute have been derived using the different metrics.

```
java -jar
/path/to/jar/benchmarks.jar <JMH ARGUMENTS>
"example_benchmark_to_exectute1|example_benchmark_to_exectute2|..."
```

Figure 5.2: Execute JMH jar command.

In Table 5.1, the values used for different arguments can be seen. All of these values are very commonly used. All other arguments are using default values.

Table 5.1: Used JMH Arguments

Argument	**Value**
Number of forks	1
Warmup iterations	5
Minimum warmup iteration time (seconds)	10
Measurement iterations	10
Minimum measurement iteration time (seconds)	10

5.2.3 CodeParser

The CodeParser class is responsible for parsing both the source code and the JUnit tests of a project. The parsing is done with the help of the JavaParser library. Each source code and JUnit test method is parsed and stored in a data class called ParentMethodInfo. Some of the important variables in this class can be seen in Table 5.2 below. As one can see from the table, the ParentMethodInfo class holds a list called "childMethodInfoList." This list holds information about any method that is called in the current method that is being parsed. For instance, if a JUnit test method is being parsed and within this method a source code method is being called, the package name, class name, and method signature of the source code method are added as an object to the "childMethodInfoList".

Table 5.2: *ParentMethodInfo* Attributes

Attribute	**Description**
filePath	Absolute path to the class with the method.
methodName	The method name.
parameters	A list of parameters for the method.
blockStatement	The method body.
qualifiedName	The fully qualified name of the class or interface with the method.
childMethodInfoList	A list of methods being called in the method body.

5.2.4 Metrics

In this section the implementation of the different metrics will be described in detail. Furthermore, before describing the implementation of specific metrics shared aspects between these metrics will be described.

5.2.4.1 LOCMetric

The LOCMetric class is responsible for handling the logic for ranking JUnit test cases based on the number of lines of code (LOC) the test case covers. The LOC of a single method is calculated by counting each physical line of code, excluding any documentation. The LOC of all of the source code methods that the JUnit test covers is then summed up to give the aggregated value for that JUnit test case.

5.2.4.2 LoopMetric

The LoopMetric class handles the calculation and ranking of JUnit test cases based on the number of loops the test case covers. More specifically, it counts how many "while", "for", and "foreach" loops the JUnit test case covers. So, if the test case covers two source code methods that both have two kinds of loops, the aggregated value for the test case will be four.

5.2.4.3 CyclomaticMetric

The CyclomaticMetric handles the calculation of Cyclomatic complexity. The formula that was used is explained in Section 4.2.4. Simply put, it counts the number of decision points in the methods that are called by the test method and their underlying methods and sums all the decision point into a integer value representing Cyclomatic complexity. It’s worth noting that, just like the other metrics, only source code methods are included in the calculations, and methods are only counted once, even if they are called several times.

5.2.4.4 CombinedMetric

The combined metrics rank JUnit tests based on all of the previously mentioned metrics. This is done by calculating the normalized value for each metric and combining these values to form the total value for a specific JUnit test. All of the metrics are valued equally, meaning that there is no emphasis on a single metric.

5.2.5 SlowdownHandler

In order to compare the performance of the subset of benchmarks generated based on metric data with a random sample of the same size, one needs to be able to inject artificial slowdowns into the source code. This logic is handled by the SlowdownHandler class. This class allows for primarily two arguments, which are the number of slowdowns that are to be injected into the source code and the desired performance regressions in milliseconds for each slowdown. When slowdowns are being injected into the source code, it randomly selects methods and injects the slowdown at the start of the method. The reason why the slowdown is injected at the start of the method is due to if, else if, else and return statements. Putting the slowdown at the start of the methods guarantees that the slowdown is executed, as one cannot statically derive the exact execution of the method.

In the figure 5.3 below, a slowdown can be seen. This slowdown forces the thread that executes the method to sleep for a specified amount of time in milliseconds. As one can see from the figure, it forces the thread to sleep for 1 milliseconds, or 0.001 seconds, before continuing. Additionally, if something goes wrong when trying to force the thread to sleep, an error message will be shown in the log. Furthermore, as the method contains a return statement within an if statement, if the slowdown had been placed at the end of the method, it would not have always been executed.

```java
@Override
public void onComplete() {
    try {
        Thread.sleep(1);
    } catch (InterruptedException injectionError) {
        System.err.println("Injection error: "
        + injectionError.getMessage());
    }
    if (done) {
        return;
    }
    done = true;
    C c = collection;
    collection = null;
    complete(c);
}
```

Figure 5.3: Injected slowdown.

5.2.6 AutomationHandler

The primary purpose of this class is to automate the process of getting data that can be used to compare the effects of artificial slowdowns on metric-based suites and randomly sampled suites of the same size. Firstly, with the help of CodeParser, it parses the source code and JUnit test cases. After this, it generates several benchmark suites using Ju2JMHLauncher. It generates several pairs of suites where each pair contains the same microbenchmarks, but one contains artificial slowdowns and one does not. Furthermore, all of the pairs contain the same artificial slowdowns in the source code. Each benchmark suite is saved to a compressed folder, which contains two files: the jar file with the benchmark suite and a text file with the command to execute the jar. An example of this can be seen in Figure 5.4. In this figure, one can see 14 compressed folders. 12 of these folders correspond to six randomly sampled suites before and after artificial slowdowns, and the last two are suites derived from metrics before and after slowdowns. The reason why six randomly sampled suites were created and not only five is because, in some cases, a suite contained errors and could not be executed. Generating one suite more than needed helps mitigate this. Nevertheless, only five randomly sampled suites were executed. After these compressed folders are generated, each folder is copied and decompressed, and each jar is executed with its corresponding command. The results of executing each jar are saved into several json files, which can be used to analyse the performance of the suite and the effects of the artificial slowdowns.

```
random_no_slowdowns_1_1709035349998.zip  random_slowdowns_3_1709036449123.zip
random_no_slowdowns_2_1709035462289.zip  random_slowdowns_4_1709036550833.zip
random_no_slowdowns_3_1709035575129.zip  random_slowdowns_5_1709036648447.zip
random_no_slowdowns_4_1709035689211.zip  random_slowdowns_6_1709036743690.zip
random_no_slowdowns_5_1709035836949.zip  ranked_no_slowdowns1709035232487.zip
random_no_slowdowns_6_1709036000963.zip  ranked_slowdowns1709036123044.zip
random_slowdowns_1_1709036250605.zip
random_slowdowns_2_1709036345539.zip
```

Figure 5.4: Example structure of compressed folders containing microbenchmark suites.

5.2.7 Code Quality and Testing

In order to ensure code quality and the correct implementation of the JMH minimisation tool, several guidelines were derived. Some of these guidelines can be seen in Table 5.3. For instance, each commit was reviewed by the other thesis partner before being pushed to the main branch of the project. This review process included looking at the code to check for errors as well as checking that proper documentation and tests had been created. It was important that relevant unit tests were created to test the functionality and correctness of developed features and algorithms. Furthermore, the code coverage tool JaCoCo [18] was used to ensure sufficient code coverage of all of the classes in the JMH minimisation tool. It is important to note that throughout the development of the JMH minimisation tool, there has always been a need to balance code quality and testing aspects with the actual development of new features, as the time scope for the project was a limiting factor.

Table 5.3: Example guidelines followed during the creation of the JMH minimisation tool

Guidelines
Code Reviews - Each commit should be reviewed prior to commit.
Unit testing - Each commit should have relevant unit tests.
S.O.L.I.D - Each commit should follow the S.O.L.I.D principle.
Documentation - All classes and method should have relevant JavaDoc.

6

Results

In the sections below the results from this thesis project are presented. All the data used and presented in the results section is available online [41].

6.1 RQ1: Code Coverage - LOC Metric

In the following sections the results from using the LOC metric to derive minimised suites on both RXJava and Mockito is presented and interpreted. The artificial slowdowns used were 1 millisecond and 10 milliseconds. TCM was done with three rounds of artificial slowdowns for each slowdown duration.

6.1.1 Number of Detected Slowdowns

In Table 6.1 the results from suites generated using the LOC metric are presented. The table displays the results from one configuration (project and slowdown time) per row and shows the difference in detection rates between the metric suites and the randomly sampled baseline suites. The most significant column to investigate is the rightmost column. The reason for this is that a negative difference in the rightmost column indicates that the metric had a worse detection rate when compared to the randomly sampled suites used as a baseline. furthermore if the value falls outside of the mean 95% confidence interval it may indicate that there is a significant difference.

Overall, the LOC-derived microbenchmark suites are worse in terms of being able to detect performance regressions when compared to the randomly derived microbenchmark suites used as a baseline. The most probable reason for LOC being really poor at detecting performance regressions is related to the fact that the metric is calculated in a static environment. The estimation of how much code could be executed is likely a lot larger than the amount of code that is actually executed. The hypothesis for the LOC-based metric was that if a JUnit was estimated to cover a lot of code statically, it would likely execute a good amount of code dynamically as well. This hypothesis, however, does not seem to hold, at least for RXJava, where the suite that was generated using the LOC metric was significantly worse at detecting performance regressions when compared to the randomly sampled suites.

The LOC-derived suites for RXJava had mean of 20.83% (1ms and 10ms) of microbenchmarks reliably detecting performance regressions when compared to the corresponding randomly sampled suites that have around 35.90% ± 3.90% and 28.26% ± 4.38% detecting artificial slowdowns, as seen in Table 6.1. Both the LOC-derived suites are lower and outside of the 95% mean confidence interval calculated for the randomly sampled suites, meaning that a LOC-derived suite in RXJava is very likely to be worse in terms of being able to detect performance degradations when compared to the randomly sampled baseline suites.

For Mockito, the results are not as clear, with the LOC-derived suites being relatively close to the randomly sampled suites in terms of the percentage of microbenchmarks that report encountering artificial slowdowns. For the 1 millisecond configuration in Mockito, the LOC-derived suite mean detection rate is within the mean 95% confidence interval calculated for the randomly sampled suites. For the Mockito 10 ms setup the LOC suite is worse and outside of the randomly sampled suites confidence interval which indicates that the LOC derived suite is likely worse at detecting performance regressions. Additionally, it is worth mentioning that Mockito has fewer source code methods and JUnit tests than RXJava. This seems to affect the results quite a bit, even if the amount of artificial slowdowns injected into the source code of Mockito was lower than that of RXJava (100 for RXJava and 25 for Mockito). This could explain the large gap between the number of microbenchmarks that encounter one or more artificial slowdowns in RXJava compared to Mockito.

Table 6.1: The % of microbenchmarks that detected performance regressions after artificial slowdown injection for LOC based suites and corresponding randomly sampled suites.

Project	**Slowdown**	**Detection Rates (%)**	**Suite Mean (%)**	**Random Suite 95% CI (%)**	**Diff. From Random Mean (%)**
RXJava	1 ms	14.58, 22.92, 25.00	20.83	35.90 ± 3.90	-15.07
RXJava	10 ms	10.42, 25.00, 27.08	20.83	28.26 ± 4.38	-7.43
Mockito	1 ms	76.09, 80.43, 82.22	79.58	81.03 ± 2.75	-1.45
Mockito	10 ms	54.35, 60.00, 80.85	65.07	70.99 ± 5.66	-5.92

6.1.2 Stability of Suites, RQ1

In Table 6.2, the stability of the LOC-based suites and their corresponding randomly sampled suites before artificial slowdowns are presented (negative difference better). Overall, the stability of the LOC-based suites and randomly sampled suites is good in RXJava and suggests no tangible difference between the randomly sampled suites and the LOC-metric-derived suites. For Mockito, the LOC-derived suites were much more stable than their randomly sampled counterparts. However, this does not indicate that LOC-derived suites are generally more stable, as the LOC-derived suites for RXJava are similar in terms of stability to the randomly sampled suites, which suggests that the stability is highly dependent on the project.

Table 6.2: The stability of the LOC metric suites as well as the randomly sampled suites used as baseline **before** artificial slowdown injection. (Negative difference from baseline is good)

Project	**Before Slowdown**	**Stability (%)**	**Suite Mean (%)**	**Random Suite 95% CI (%)**	**Diff. From Baseline Mean (%)**
RXJava	1 ms	0.68, 0.88, 2.06	1.21	1.49 ± 0.33	-0.28
RXJava	10 ms	0.86, 1.05, 2.10	1.34	1.30 ± 0.49	0.04
Mockito	1 ms	2.39, 2.69, 3.23	2.77	10.36 ± 2.36	-7.59
Mockito	10 ms	2.39, 2.56, 2.72	2.56	10.78 ± 3.23	-8.22

In Table 6.3, the stability of the LOC-derived suites after the injection of slowdowns is presented. Overall, stability was better with artificial slowdowns. Additionally, for three out of four suites derived from the LOC metric, the stability was better compared to the randomly sampled suites.

Table 6.3: The stability of the LOC metric suites as well as the randomly sampled suites used as baseline **after** artificial slowdown injection. (Negative difference from baseline is good)

Project	**Slowdown**	**Stability (%)**	**Suite Mean (%)**	**Random Suite 95% CI (%)**	**Diff. From Baseline Mean (%)**
RXJava	1 ms	0.41, 0.79, 1.14	0.78	1.39 ± 0.32	-0.71
RXJava	10 ms	0.28, 0.41, 6.71	2.47	1.21 ± 0.98	1.26
Mockito	1 ms	0.58, 0.83, 1.01	0.81	2.63 ± 1.68	-1.82
Mockito	10 ms	0.37, 0.43, 0.46	0.42	3.13 ± 2.41	-2.71

6.2 RQ2: Computational Complexity - Loop Metric

In the following sections the results from using the Loop metric to derive minimised suites on both RXJava and Mockito is presented and interpreted.

6.2.1 Number of Detected Slowdowns, RQ2

In Table 6.4, the data from the loop metric-based suites is presented. From this data, it is clear that these suites performed worse compared to their corresponding randomly sampled suites as all the metric-derived suites performed worse. There are several reasons why this might be the case. For instance, similarly to the LOC-based suites, the metric was estimated statically, so the data for the loop metric might not be representative of the actual code being executed during runtime. Furthermore, it might also be the case that the effects of the type of artificial slowdowns used during this experiment (Thread.sleep) are too large to sufficiently assess the applicability of the loop count metric. The hypothesis was that in code that has a high loop count, performance regressions could have a larger effect even if the slowdown itself is small, as it may be executed several times. Essentially, this means that it could be the case that one has to use considerably smaller artificial slowdowns in order for methods with a high loop count to perform better compared to randomly sampled methods. This is an important aspect that should be considered for any future work that plans on performing TCM and where artificial slowdowns are utilised for evaluation of the results.

Table 6.4: The % of microbenchmarks that detected performance regressions after artificial slowdown injection for loop based suites and corresponding randomly sampled suites.

Project	**Slowdown**	**Detection Rates (%)**	**Suite Mean (%)**	**Random Suite 95% CI (%)**	**Diff. From Random Mean (%)**
RXJava	1 ms	32.65, 32.65, 34.69	33.33	43.00 ± 5.23	-9.67
RXJava	10 ms	10.20, 18.37, 34.69	21.09	26.82 ± 4.63	-5.73
Mockito	1 ms	47.73, 75.00, 75.56	66.10	73.41 ± 5.66	-7.31
Mockito	10 ms	53.33, 56.82, 72.73	60.96	76.78 ± 4.57	-15.82

6.2.2 Stability of Suites, RQ2

In Table 6.5, the stability of the loop metric-derived suites is presented. Once again, it is very noticeable that overall, RXJava seems to generate more stable microbenchmarks, which suggests that the efficacy of generating microbenchmarks from JUnit tests seems to be dependent on the project. The loop metric suite generated for RX-Java were unstable compared to the randomly sampled counterparts with the mean suite stability being reliably above the randomly sampled mean 95% confidence interval by more than 1% which means that using the loop metric to generate suites may lead to less stable microbenchmarks in some projects. However, for Mockito the loop count metric once again seems to yield suites with considerably more stable microbenchmarks when compared to the randomly sampled suites. However, the poor stability of the randomly sampled suites for Mockito does not necessarily mean that the loop metric is good at generating stable suites, more importantly it highlights that the project plays a big role in how stable randomly picked JUnit tests are as microbenchmarks.

The stability of the suites after artificial slowdown injection can be seen in Table 6.6. Overall, the results from the stability of the suites after slowdown injection tell the same story as before artificial slowdown injection, meaning that for RXJava, the

Table 6.5: The stability of the loop metric suites as well as the randomly sampled suites used as baseline **before** artificial slowdown injection. (Negative difference from baseline is good)

Project	**Before Slowdown**	**Stability (%)**	**Suite Mean (%)**	**Random Suite 95% CI (%)**	**Diff. From Baseline Mean (%)**
RXJava	1 ms	1.80, 3.19, 3.44	2.81	1.43 ± 0.35	1.38
RXJava	10 ms	2.86, 3.02, 3.04	2.97	1.30 ± 0.28	1.67
Mockito	1 ms	2.31, 2.44, 2.52	2.42	11.74 ± 3.16	-9.32
Mockito	10 ms	2.64, 2.88, 2.96	2.83	11.44 ± 4.18	-8.61

stability was worse than the randomly sampled counterparts, and for Mockito, the stability was better than the randomly sampled suites, which indicates that stability is highly dependant on the project.

Table 6.6: The stability of the loop metric suites as well as the randomly sampled suites used as baseline **after** artificial slowdown injection. (Negative difference from baseline is good)

Project	**Slowdown**	**Stability (%)**	**Suite Mean (%)**	**Random Suite 95% CI (%)**	**Diff. From Baseline Mean (%)**
RXJava	1 ms	2.07, 3.00, 6.99	4.02	1.72 ± 1.13	2.30
RXJava	10 ms	1.46, 2.14, 3.00	2.20	1.17 ± 0.40	1.03
Mockito	1 ms	0.53, 0.67, 0.75	0.65	2.92 ± 1.81	-2.27
Mockito	10 ms	0.39, 0.49, 0.83	0.42	2.65 ± 1.58	-2.08

6.3 RQ3: Cyclomatic Complexity - Cyclomatic Metric

In the following sections the results from using the Cyclomatic metric to derive minimised suites on both RXJava and Mockito is presented and interpreted.

6.3.1 Number of Detected Slowdowns, RQ3

In Table 6.7, the data corresponding to the detection rate of artificial slowdowns for the cyclomatic metric is presented. This data differs from the data from the previous metrics. Unlike previous metrics, the performance of the cyclomatic metric falls within the confidence intervals of three out of four randomly sampled baselines. This data indicates that there is not a meaningful difference between the metric and the randomly sampled suites. There are several reasons why this might be the case but one reason can simply be that in a static analysis context the cyclomatic metric offers some basic correlation with the amount of code being dynamically executed. However, it is still the case that generating suites using the cyclomatic-metric seems to be a bad startegy as the baseline randomly sampled suite approach is not necessarily a good strategy.

Table 6.7: The % of microbenchmarks that detected performance regressions after artificial slowdown injection for cyclomatic based suites and corresponding randomly sampled suites.

Project	Slowdown	Detection Rates (%)	Suite Mean (%)	Random Suite 95% CI (%)	Diff. From Random Mean (%)
RXJava	1 ms	16.67, 18.75, 20.83	18.75	34.05 ± 4.44	-15.30
RXJava	10 ms	8.33, 16.67, 58.33	27.78	26.79 ± 5.66	0.99
Mockito	1 ms	61.36, 72.73, 86.67	73.59	74.18 ± 3.16	-0.59
Mockito	10 ms	63.64, 68.18, 88.89	73.57	76.7 ± 3.32	-3.13

6.3.2 Stability of Suites, RQ3

In this section the stability of the suites used for evaluation of the cyclomatic based suites will be presented and interpreted.

Stability of the cyclomatic-metric-derived suites before artificial slowdown injection is presented in Table 6.8. As seen from the table, the stability of the cyclomatic-metric-derived microbenchmark suites is quite similar to the randomly sampled baseline suites in RXJava, indicating no significant difference as the stability is within the confidence intervals in RXJava. For Mockito, the randomly sampled suites are quite unstable, which aligns with previous results. This is the cause of some microbenchmarks that are very unstable, causing the average stability to rise quite a bit. However, with the number of randomly sampled suites generated for Mockito, the microbenchmarks that are unstable cannot necessarily be deemed outliers, as the average stability of the randomly sampled suites seems to fluctuate around 10–12 percent, which indicates that some very unstable microbenchmarks almost always get included. It also seems to be the case that the metric-based suites manage to avoid microbenchmarks with bad stability in Mockito, but the underlying reason for this is unclear. It may be the case that the results in terms of stability would look a lot different if the microbenchmark suites included more microbenchmarks, e.g., 100 microbenchmarks instead of 50.

The stability of the microbenchmark suites used to evaluate the cyclomatic-metric-derived suites after slowdown injection can be seen in Table 6.9. The implications of the results after the artificial slowdown injection were similar to the takeaways before the artificial slowdown injection. However, the RXJava 10 ms cyclomatic-generated

Table 6.8: The stability of the cyclomatic metric suites as well as the randomly sampled suites used as baseline **before** artificial slowdown injection. (Negative difference from baseline is good)

Project	**Before Slowdown**	**Stability (%)**	**Suite Mean (%)**	**Random Suite 95% CI (%)**	**Diff. From Baseline Mean (%)**
RXJava	1 ms	0.74, 0.88, 1.51	1.04	1.10 ± 0.23	-0.06
RXJava	10 ms	0.56, 1.02, 1.16	0.91	1.30 ± 0.49	-0.39
Mockito	1 ms	2.83, 2.97, 3.17	2.99	12.07 ± 3.85	-9.08
Mockito	10 ms	2.53, 2.73, 2.96	2.74	11.7 ± 4.28	-8.96

suite displayed stability results that were outside of the 95% mean confidence interval of the randomly sampled counterpart and better in terms of stability. But it is not possible to draw any larger conclusions from this as the 1 ms RXJava setup is still within the confidence interval. The main reason that the Mockito suites are more stable after artificial slowdown injection is related to the fact that the artificial slowdowns 'water down' the results, causing the fluctuations of the measurement iterations of individual microbenchmarks to vary less between as the artificial slowdowns are quite large. For Mocktito, the cyclomatic-metric-derived suites were better and outside of the 95% mean confidence interval after slowdown injection, which is to be expected given the stability results from before slowdown injection.

Table 6.9: The stability of the Cyclomatic metric suites as well as the randomly sampled suites used as baseline **after** artificial slowdown injection. (Negative difference from baseline is good)

Project	**Before Slowdown**	**Stability (%)**	**Suite Mean (%)**	**Random Suite 95% CI (%)**	**Diff. From Baseline Mean (%)**
RXJava	1 ms	0.75, 0.81, 0.83	0.80	1.02 ± 0.25	-0.22
RXJava	10 ms	0.46, 0.51, 0.71	0.59	1.30 ± 0.50	-0.74
Mockito	1 ms	0.68, 0.88, 1.02	0.86	3.12 ± 1.74	-2.26
Mockito	10 ms	0.39, 0.41, 0.66	0.49	3.40 ± 2.03	-2.91

6.4 RQ4: Combined Metrics

In the following sections, the results from using the combination of LOC, loop, and cyclomatic metrics to derive minimised suites on both RXJava and Mockito are presented and interpreted.

6.4.1 Number of Detected Slowdowns, RQ4

In Table 6.10, the detection rates for the combined metric and its corresponding randomly sampled suites are presented. This data is similar to the data for the cyclomatic metric, three out of four suites for the combined metric falls within the confidence interval of the randomly sampled suites. For these suites, there seems to be no relevant difference. One important part of the data is that the combined metric had several detection rates that were very high (e.g., 95.12 %) and this high rate of detection was not seen for the randomly sampled suites. However, the combined metric also provided suites with a very low detection rate (e.g., 8.33 %) which severely lowered the mean of the suites derived from the combined metrics. Overall, the results indicate that generating suites using the combined metric with equal weights seems to be a bad strategy, as it is not better than randomly sampled suites at detecting performance regressions.

Table 6.10: The % of microbenchmarks that detected performance regressions after artificial slowdown injection for combined metric based suites and corresponding randomly sampled suites.

Project	**Slowdown**	**Detection Rates (%)**	**Suite Mean (%)**	**Random Suite 95% CI (%)**	**Diff. From Random Mean (%)**
RXJava	1 ms	8.33, 22.92, 45.83	25.69	41.33 ± 9.46	-15.64
RXJava	10 ms	8.33, 20.83, 27.08	18.75	20.41 ± 3.54	-1.66
Mockito	1 ms	34.88, 73.17, 95.12	67.72	69.06 ± 5.67	-1.34
Mockito	10 ms	9.09, 75.00, 95.12	59.74	62.65 ± 16.14	-2.91

6.4.2 Stability of Suites, RQ4

In this section the results in terms of stability for the combined-metric-derived suites will be interpreted and presented.

In Table 6.11, the stability of the suites before artificial slowdown injection is presented. In terms of stability for RXJava, the combined metric one suite was worse and outside the 95% mean confidence interval of the randomly sampled suites and the other was within the interval. For Mockito, the stability is just like that of the other metric-based suites, meaning it is quite good when compared to the randomly sampled suites. However, drawing any meaningful conclusions from this is hard, as the results likely would look a lot different if a different project was used or if more microbenchmarks were included in the generated suites. Furthermore, the overall stability of most of the microbenchmarks that can be generated for Mockito using ju2jmh seem to be worse in terms of stability overall when compared to a larger project like RXJava. This is an indication that generating microbenchmarks using ju2jmh will have very different results in terms of stability depending on the project.

Table 6.11: The stability of the combined metric suites as well as the randomly sampled suites used as baseline **before** artificial slowdown injection. (Negative difference from baseline is good)

Project	Before Slowdown	Stability (%)	Suite Mean (%)	Random Suite 95% CI (%)	Diff. From Baseline Mean (%)
RXJava	1 ms	1.52, 1.87, 2.49	1.96	1.61 ± 0.47	0.35
RXJava	10 ms	0.82, 1.84, 2.14	1.60	1.09 ± 0.31	0.51
Mockito	1 ms	3.03, 3.45, 3.70	3.39	10.56 ± 4.20	-7.17
Mockito	10 ms	2.14, 2.83, 3.62	2.86	12.64 ± 3.65	-9.78

After slowdown injection the combined metric suites had a bit worse stability in the RXJava configurations when compared to that of the randomly sampled suites, while for the Mockito configurations once again the metric-derived suites were more stable and the overall stability of the randomly sampled suites became a lot better after artificial slowdown injection. This is because the artificial slowdowns made the results more stable because of their inherent size, meaning that the measurement iterations record more proportionally sized results when encountering artificial slowdowns.

Table 6.12: The stability of the combined metric suites as well as the randomly sampled suites used as baseline **after** artificial slowdown injection. (Negative difference from baseline is good)

Project	Before Slowdown	Stability (%)	Suite Mean (%)	Random Suite 95% CI (%)	Diff. From Baseline Mean (%)
RXJava	1 ms	1.29, 2.37, 2.61	2.09	0.93 ± 0.21	1.16
RXJava	10 ms	0.62, 1.72, 2.21	1.52	0.99 ± 0.28	0.53
Mockito	1 ms	0.51, 0.55, 1.37	0.81	4.84 ± 3.31	-4.03
Mockito	10 ms	0.15, 0.21, 1.82	0.73	4.15 ± 3.01	-3.42

7 Discussion

In the following sections, the results will be analysed and discussed. Furthermore, throughout the process of this thesis, several areas of improvement have been recognised. So, potential changes to the method will be discussed, which might have yielded better results and which can serve as an aid or starting point for future work. Additionally, the reasoning behind several methods will be discussed.

7.1 LOC Metric, RQ 1 Discussion

The LOC metric that was used to estimate code coverage performed worse than the randomly sampled suites in terms of being able to detect performance regressions. The poor results are likely because the LOC metric was calculated in a static environment. Using dynamic tools to determine what is actually executed is likely to improve the results in terms of how many performance regressions the microbenchmarks encounter. However, it is unclear how dynamically calculated LOC would affect the stability of the generated microbenchmarks and it is also unknown if it would outperform randomly sampled suites in terms detecting performance regressions. In Mockito, the LOC metric seemed to lead to favourable results in terms of the overall stability of the generated suites, but that is not the case for the LOC-derived suites for RXJava, which indicates that different projects and underlying code can greatly affect the effectiveness of microbenchmarks generated from JUnit tests using *ju2jmh*.

Furthermore, instead of focusing on maximising the LOC and disregarding all other aspects, it could be interesting to investigate if maximising LOC where performance regressions are more likely to occur (e.g., I/O operations or networking), as focusing on relevant LOC could be better than strictly counting all the LOC that a test case covers.

7.2 Loop Metric, RQ 2 Discussion

The loop metric that was used to estimate computational complexity performed worse in terms of being able to detect performance regressions when compared to the randomly sampled suites. However, the artificial slowdowns used in this thesis project are quite large, which in turn means that the artificial slowdowns are likely quite easy to detect. If considerably smaller artificial slowdowns were utilised, the results in terms of detection of performance regression could look quite different, as smaller performance regressions could become substantial if executed several times in a loop, meaning that microbenchmarks that execute loops could be better at detecting smaller performance regressions. However, this does not mean that the loop metric was good as the overall detection rates were bad and the stability seemed to suffer in RXJava as well. Additionally there is nothing to suggest that the loop metric is overall very good at finding small performance regressions but it could be better than randomly sampled suites. Furthermore, designing artificial slowdowns is a complex endeavour in and of itself, as quantifying a small slowdown is highly related to the project, the code being benchmarked, and the underlying hardware being used to run the code, which makes the process of creating small, realistic artificial slowdowns very intricate.

7.3 Cyclomatic Metric, RQ 3 Discussion

The data from the cyclomatic metric indicates that the performance of the metric is slightly worse or similar to the randomly sampled suites when it comes to detecting performance regressions. Furthermore, the stability of the metric was similar compared to the randomly sampled suites. Overall, the results from the cyclomatic metric are underwhelming, as it did not outperform randomly sampled suites in terms of finding performance regressions or overall suite stability.

7.4 Combined Metric, RQ 4 Discussion

The combined metric data shows similar results to the cyclomatic metric, both regarding detection rates and stability. However, one very interesting aspect that was previously mentioned in the result section is that while the mean detection rates were quite similar to the randomly sampled suite for three out of four configurations, the combined metric provided the highest detection rates of individual suites. However, it also had some individual suites that produced very low detection rates, which severely lowered the mean values. The reason why this is interesting is because future work could possibly take what was done in this work and improve it, which might lead to a metric that is statistically better than randomly sampled suites. Some areas of improvement include moving from static code analysis to dynamic code analysis, assigning weight to different metrics, and/or the removal of some metrics (e.g. LOC) to form a more refined combined metric. Furthermore, it could be interesting to investigate in what cases the very low detection rates typically occur and try to mitigate them.

7.5 The Effects of Randomly Injecting Artificial Slowdowns

During the data collection process, numerous artificial slowdowns were randomly injected into the source code. For each configuration (e.g., project, metric, slowdown duration), a new set of artificial slowdowns was randomly injected. This was done in order to see the effects the locations of the artificial slowdowns had on the detection rates and the overall stability of the suites. It is clear from the results that the number of benchmarks that encounter slowdowns is highly dependent on the artificial slowdown locations. However, a lot of randomly sampled suites were executed (120 for each project), and each data gathering configuration indicated that the metric-based suites were significantly worse or equal to the randomly sampled suites. This indicates that the metric-based suites are likely worse, but by gathering more data for each metric-derived microbenchmark suite, this claim could have been strengthened.

7.6 Areas of Improvements

The area of research that involves microbenchmarking with JMH is relatively new and not a common area to conduct research in. The number of papers done in previous work is extremely sparse when compared to more popular topics such as machine learning or artificial intelligence. Due to this, several design choices in the method had to be selected without being substantiated by previous work, which is simply because of the overall lack of studies regarding JMH. Furthermore, after accounting for the fact that this work uses the tool *ju2jmh* to create microbenchmarks from JUnit tests, the amount of previous work goes to nearly non-existent. As a result, several method choices that were made in the early stages of the thesis might not have been optimal. In the following sections, these method choices will be discussed, as well as how different choices might have yielded better results.

7.6.1 Static Code Analysis versus Dynamic Code Analysis

At the start of the thesis, it was decided to limit the scope to static code analysis only. The main reasoning behind this was simply that it was assumed that static code analysis would have less overhead in terms of execution time compared to dynamic code analysis. This was particularly important, as the initial plan and hope were to create a tool that could be used in CI/CD pipelines where time is often a limiting factor. It is not difficult to see that this might not have been the most optimal approach. Instead of opting out for dynamic code analysis, it would most likely have been beneficial to conduct only dynamic analysis or a combination. There are several reasons why a dynamic or a combination of dynamic and static code analysis could have yielded better results. The primary reason might simply be that the statistics gained related to lines of code, computational complexity, or code complexity are not representative of the dynamic code execution. For instance, a JUnit test might statically cover a lot of lines of code or have a high computational complexity, but the dynamic execution path of that specific test case might not cover a lot of code nor have a high computational complexity. Due to aspects such as this, it would, in hindsight, most likely be better to not only focus on static code analysis. So, any future work that aims to also conduct test case minimisation of microbenchmark suites should most likely apply the knowledge gained during this thesis and not exclude dynamic code analysis.

7.6.2 Data Gathering and Implementation Approach

Another area of improvement is to better align the data gathering process and implementation phases of the project. The metric calculations were researched and implemented one at a time. After implementing each metric, data gathering for that metric commenced. However, this approach may not have been optimal, leading to issues in comparing results across metrics. The main issue was that each data gathering round featured its own set of artificial slowdowns in random source code methods, complicating comparisons between metric-based suites. This is important because the location of these slowdowns significantly influenced the results. In hindsight, generating all the different metric-based and randomly sampled suites while using the same set of randomly placed artificial slowdowns would allow for better comparisons within the same set of slowdowns, leading to more comparable results. The approach would be to implement all the metric calculations first and then use them at the same time. However, this could greatly stagger the start of the data gathering process, which in turn could lead to less total data being gathered. Furthermore, it is paramount to underline that the implementation of individual metrics was a time-consuming process that required a fair amount of time. This process included research, implementation, automated testing, and manual verification of the functionality.

7.6.3 Challenges Implementing Artificial Slowdowns in Java Applications

Artificial slowdowns have played a crucial role for this work. However, the usage and implementation of artificial slowdowns is intricate and complex. There are several aspects that render artificial slowdowns hard to use. Java functions such as *Thread.sleep* are highly tied to the operating system that the JVM is running on, an example of this is that for Windows operating systems the overhead of *Thread.sleep* can be as high as 15 milliseconds while for Linux based systems it is often much lower around 1 millisecond [42], [43]. This makes the actual time added by a artificial slowdown utilising *Thread.sleep* somewhat unclear. However utilising other techniques such as adding computationally heavy to act as artificial slowdowns is also complex, as the JVM and it is optimisations as well as underlying hardware make it difficult to know exactly how large the artificial slowdown is. Additionally, the act of measuring the artificial slowdown itself takes time which in turn complicates this even more.

7.7 Limitations and Delimitations

In the following sections limitations, internal validity, and external validity will be discussed.

7.7.1 Limitations

One primary limitation of this study is the time scope. Executing single benchmark suites can take several hours to complete. So, if one wants to conduct a thorough comparison analysis of various suites, it can take several days just to generate the suites. Furthermore, many times the different parameters used for generating the suites need to be altered in order to conduct sensitivity analysis, which greatly increases the time it takes to get the data used for analysis. This limitation is further exacerbated by the fact that only two VMs were available during the scope of the project. If more VMs were available, the generation of suites could have been spread evenly across these in an attempt to mitigate the effects of the time-consuming tasks. Using personal machines was out of the question because, in order to get reliable results, the microbenchmarking suites needed to be executed in stable environments with as little disturbance as possible from background processes.

For each configuration (e.g., project, metric, slowdown duration), during the data collection process, only three repetitions with different artificial slowdowns were conducted. The primary reason only three repetitions were done was due to the limiting time scope. This low number of repetitions could pose a threat to the validity of the conclusion of this work, as more repetitions could possibly alter the results which in turn could affect the final conclusions.

7.7.2 Internal Validity

In this thesis project, there are some threats to internal validity. One threat to internal validity is the fact that JavaParser, which is the parser that was used for this thesis project, sometimes has limitations when handling more complex code sections that contain generic types and lambda functions. This can make the underlying calculated metrics less reliable, but it is worth noting that these parsing errors are rare. Another threat to internal validity is related to hardware. The results in this thesis project come from running the microbenchmarks on the same virtual machines, but even small background processes that the operating system may launch can affect the microbenchmark results.

Furthermore, another core threat to internal validity is the implementation of the tool used in this project. The tool uses several complex algorithms and features, and there is always a risk that some of these contain errors and are not working as intended, which could have affected the results. Several actions have been taken in attempt to mitigate the risks of faulty code. For instance, unit testing and code reviewing have been used extensively during the scope of this project.

Another factor that can pose a threat to the internal validity is the fact that some benchmarks that were converted from JUnit test cases to benchmarks by ju2jmh were unable to execute. So, these benchmarks were manually removed in order to make the full benchmark suite execute. It is possible that the removal of these benchmarks could have affected the end result. However, due to the magnitude of the difference in the number of removed benchmarks compared to the total number of executed benchmarks, the removal of these benchmarks should have little to no effect on the results.

The mathematical equation described in Section 4.3.3 used for determining if a benchmark has been affected by artificial slowdowns or not is not perfect. This means that although the results from the equation might suggest that an individual benchmark has been affected by artificial slowdowns, that might not always be the case. However, as the equation should not be used for assessing individual benchmarks with high accuracy, this should not be a problem. The goal with the equation is to capture larger patterns within entire microbenchmark suites.

7.7.3 External Validity

A threat to external validity is that JMH microbenchmarking is designed and used for Java specifically. The metrics were found to be related to good quality benchmarks may be applicable to other languages and frameworks, but the results when using those metrics in other languages could vary.

The results that were found when working on this thesis are based on OSS projects found on Github. It is debatable if the results will are valid and applicable in other contexts, such as industrial software and software that can be found on other platforms similar to Github. Another external threat to validity is that Java is a programming language that keeps getting developed upon and future changes may have a large effect on how applicable the results of this thesis are in the future as changes to the JVM could have a large impact on microbenchmarks. Furthermore, only two projects were used for data gathering and this also makes the results hard to generalise and extend to other projects outside of the two that were used.

Additionally, a threat to external validity is how artificial slowdowns are injected into the code during the comparison process. These slowdowns might not be representative of real-world scenarios where actual new features are committed, which could introduce performance regressions. So, while artificial slowdowns can allow for a controlled way to study performance impacts and microbenchmark suites ability to detect performance regressions, their ability to capture the complex changes that occur in production environments is limited. Real-world performance regressions often come from complex interactions between new features and existing code. These interactions can create complex performance issues that the artificial slowdowns cannot fully emulate.

8

Conclusion

In this study, metrics related to code coverage, code complexity, computational complexity, and a combination of the three aforementioned were estimated statically and used to prioritise JUnit tests to be made into microbenchmarks using *ju2jmh*. The metric-based suites consistently had poor ability to reliably detect performance regressions when compared to the randomly sampled suites, even if the stability of the microbenchmarks before and after artificial slowdown injection in the detection calculation was included in the performance regression detection calculation. Additionally, there was no interesting difference between the metrics in terms of their ability to detect performance regressions, as all of them performed quite poorly. The stability of the loop-metric-derived suites in RXJava showed that the stability is likely to get worse if more loops are executed, as the results show that the loop-metric-suite yields microbenchmarks with worse stability when compared to the randomly sampled suites. However, this needs to be investigated further as the results were not the same in Mockito using the loop metric.

For Mockito, the stability of the results of the metric-based suites was quite good when compared to the randomly sampled suites. But, overall, the JUnit tests from Mockito were more unstable as microbenchmarks when compared to the JUnit tests of RXJava, which makes it unlikely that the results in terms of stability from Mockito are significant as the stability seems highly dependent on the project.

The main factor that made the metrics unreliable is likely that the parsing of the source code and estimation of the metrics were done in a purely static environment, which could lead to poor estimations. If a similar study is conducted in the future, a dynamic approach to parsing and estimating the metrics is to be preferred, even if the cost is longer execution times related to the parsing process.

8.1 Future Work

During this study regarding test case minimisation for JMH microbenchmark suites, several interesting areas of further research and development have been brought to light. This includes both improvements to the methods used in this study and potential new areas of research that could be interesting to investigate.

Since the study has been conducted within a limited time frame and due to the time needed to execute microbenchmark suites, it could prove insightful to repeat the same study on a larger time scope where the method does not have to take time into account for every aspect. This could lead to more extensive sensitivity analysis and the execution of microbenchmark suites with parameters that offer more data at the cost of increased execution time.

The implementation of the metric calculations used in this study does not account for overlapping coverage, which means that some of the microbenchmarks in the metric-derived suites cover some of the same code. This also makes it likely that the microbenchmarks encounter the same artificial slowdowns. Overlapping code coverage is inherently ineffective, as covering the same code several times is costly in terms of execution time and may not add additional interesting information. For future work, including a strategy that accounts for overlapping coverage while also calculating the metrics more correctly with a dynamic approach could help improve overall coverage and suite quality a lot.

For this study, the artificial slowdowns are injected randomly into source code methods. This is something that can be conducted differently to further the sensitivity analysis. One potential way this could be done is instead of only randomly injecting artificial slowdowns, one could, for example, introduce some form of bias during the selection process of which methods are to be injected with slowdowns. This could be done by favouring methods that have a high metric value, for instance, lines of code, while still keeping some form of random selection. A second potential way that is much more complex but could provide a way to use non-artificial slowdowns and instead use actual performance regressions is to use a version control system like Git to sample real performance regressions. This could prove to be valuable as it can evaluate the minimised suites on real performance regressions. However, the process of sampling enough regressions to do any form of comparison might prove to be very time-consuming, and these regressions will be project-specific and hence offer little to no value between different projects. Furthermore, using an alternative type of slowdown could also prove valuable. For instance, instead of forcing threads to sleep for a specific duration, one could introduce slowdowns by performing different types of calculations, which could allow for more control and even smaller performance regressions.

Based on the results, several of the metrics are worth investigating and refining further. This includes both testing and modifying individual metrics and trying different weights on individual metrics in the combined metric. It is very probable that this could increase the performance of the metrics, which might lead to metrics that consistently outperform the suites based on random sampling. Furthermore, since this study focused on metrics derived from static code analysis, future work could also try to use the same metrics but using a dynamic approach. For instance, instead of trying to estimate computational complexity by counting loops through static code analysis, one could try to dynamically see which execution paths a program runs to get a more precise evaluation. However, this will most likely be a much more complex and time-consuming process.

Overall, this paper has provided several potential ways to do and not to do future research targeting test case minimisation for JMH. Any future work that plan continuing this work would most probable be able to create metrics that consistently outperform randomly sampled suites if the lessons learned in this work is taken into account.

Bibliography

[1] E. Weyuker and F. Vokolos, "Experience with performance testing of software systems: Issues, an approach, and case study," *Software Engineering, IEEE Transactions on*, vol. 26, pp. 1147–1156, Jan. 2001. DOI: 10.1109/32.888628.

[2] OpenJDK. "Java microbenchmarking harness (jmh)." (2023), [Online]. Available: https://github.com/openjdk/jmh (visited on 11/22/2023).

[3] Oracle. "Java virtual machine technology overview." (2021), [Online]. Available: https://docs.oracle.com/en/java/javase/21/vm/java-virtual-machine-technology-overview.html#GUID-982B244A-9B01-479A-8651-CB6475019281 (visited on 11/23/2023).

[4] D. Ståhl and J. Bosch, "Modeling continuous integration practice differences in industry software development," *Journal of Systems and Software*, vol. 87, pp. 48–59, 2014, ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2013.08.032. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121213002276.

[5] C. Laaber, S. Würsten, H. C. Gall, and P. Leitner, "Dynamically reconfiguring software microbenchmarks: Reducing execution time without sacrificing result quality," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020, Virtual Event, USA: Association for Computing Machinery, 2020, pp. 989–1001, ISBN: 9781450370431. DOI: 10.1145/3368089.3409683. [Online]. Available: https://doi.org/10.1145/3368089.3409683.

[6] C. Laaber and P. Leitner, "An evaluation of open-source software microbenchmark suites for continuous performance assessment," in *Proceedings of the 15th International Conference on Mining Software Repositories*, ser. MSR '18, Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 119–130, ISBN: 9781450357166. DOI: 10.1145/3196398.3196407. [Online]. Available: https://doi.org/10.1145/3196398.3196407.

[7] maven.org. "Maven jmh." (2024), [Online]. Available: https://mvnrepository.com/artifact/org.openjdk.jmh/jmh-core (visited on 02/07/2024).

[8] maven.org. "Maven junit." (2024), [Online]. Available: https://mvnrepository.com/artifact/junit (visited on 02/07/2024).

[9] D. Costa, C.-P. Bezemer, P. Leitner, and A. Andrzejak, "What's wrong with my benchmark results? studying bad practices in jmh benchmarks," *IEEE Transactions on Software Engineering*, vol. 47, no. 7, pp. 1452–1467, 2021. DOI: 10.1109/TSE.2019.2925345.

[10] M. Jangali, Y. Tang, N. Alexandersson, P. Leitner, J. Yang, and W. Shang, "Automated generation and evaluation of jmh microbenchmark suites from unit tests," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 1704–1725, 2023. DOI: 10.1109/TSE.2022.3188005.

[11] Y. Zhao, L. Xiao, A. B. Bondi, B. Chen, and Y. Liu, "A large-scale empirical study of real-life performance issues in open source projects," *IEEE Transactions on Software Engineering*, vol. 49, no. 2, pp. 924–946, Feb. 2023, ISSN: 1939-3520. DOI: 10.1109/TSE.2022.3167628.

[12] Oracle. "Java virtual machine technology overview." (2021), [Online]. Available: https://docs.oracle.com/en/java/javase/21/vm/java-virtual-machine-technology-overview.html#GUID-982B244A-9B01-479A-8651-CB6475019281 (visited on 02/01/2024).

[13] J. Community. "Java virtual machine optimizations." (2023), [Online]. Available: https://docs.oracle.com/en/database/oracle/oracle-database/21/jjdev/Oracle-JVM-JIT.html (visited on 11/23/2023).

[14] L. Traini, V. Cortellessa, D. Di Pompeo, and M. Tucci, "Towards effective assessment of steady state performance in java software: Are we there yet?" *Empirical Softw. Engg.*, vol. 28, no. 1, Jan. 2023, ISSN: 1382-3256. DOI: 10.1007/s10664-022-10247-x. [Online]. Available: https://doi.org/10.1007/s10664-022-10247-x.

[15] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, 2011. DOI: 10.1109/TSE.2010.62.

[16] J. Team. "Junit 5." (2024), [Online]. Available: https://junit.org/junit5/ (visited on 02/01/2024).

[17] H. Hemmati, "How effective are code coverage criteria?" In *2015 IEEE International Conference on Software Quality, Reliability and Security*, 2015, pp. 151–156. DOI: 10.1109/QRS.2015.30.

[18] J. contributors. "Jacoco java code coverage library." (2023), [Online]. Available: https://github.com/jacoco/jacoco (visited on 02/01/2024).

[19] S. Bhatia and J. Malhotra, "A survey on impact of lines of code on software complexity," in *2014 International Conference on Advances in Engineering & Technology Research (ICAETR - 2014)*, 2014, pp. 1–4. DOI: 10.1109/ICAETR.2014.7012875.

[20] M. Lipow, "Number of faults per line of code," *IEEE Transactions on Software Engineering*, vol. SE-8, no. 4, pp. 437–439, 1982. DOI: 10.1109/TSE.1982.235579.

[21] T. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, 1976. DOI: 10.1109/TSE.1976.233837.

[22] G. K. Gill and C. F. Kemerer, "Cyclomatic complexity density and software maintenance productivity," *IEEE transactions on software engineering*, vol. 17, no. 12, pp. 1284–1288, 1991.

[23] C. Indumathi and K. Selvamani, "Test cases prioritization using open dependency structure algorithm," *Procedia Computer Science*, vol. 48, pp. 250–255, 2015, International Conference on Computer, Communication and Convergence (ICCC 2015), ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.

procs.2015.04.178. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050915006870.

[24] R. Singh and M. Santosh, "Test case minimization techniques: A review," *International Journal of Engineering Research & Technology (IJERT)*, vol. 2, no. 12, 2013.

[25] R. Mukherjee and K. S. Patnaik, "A survey on different approaches for software test case prioritization," *Journal of King Saud University - Computer and Information Sciences*, vol. 33, no. 9, pp. 1041–1054, 2021, ISSN: 1319-1578. DOI: https://doi.org/10.1016/j.jksuci.2018.09.005. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1319157818303616.

[26] H. Samoaa and P. Leitner, "An exploratory study of the impact of parameterization on jmh measurement results in open-source projects," in *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '21, Virtual Event, France: Association for Computing Machinery, 2021, pp. 213–224, ISBN: 9781450381949. DOI: 10.1145/3427921.3450243. [Online]. Available: https://doi.org/10.1145/3427921.3450243.

[27] C. Laaber, M. Basmaci, and P. Salza, "Predicting unstable software benchmarks using static source code features," *Empirical Softw. Engg.*, vol. 26, no. 6, Nov. 2021, ISSN: 1382-3256. DOI: 10.1007/s10664-021-09996-y. [Online]. Available: https://doi.org/10.1007/s10664-021-09996-y.

[28] C. Laaber, H. C. Gall, and P. Leitner, "Applying test case prioritization to software microbenchmarks," *Empirical Software Engineering*, vol. 26, no. 6, Sep. 2021. DOI: 10.1007/s10664-021-10037-x.

[29] A. Path, "Optimizing the generation of java jmh benchmarks," Available at: https://gupea.ub.gu.se/handle/2077/72730, M.S. thesis, Department of Computer Science, Engineering, Chalmers University of Technology, and University of Gothenburg, Jul. 2022.

[30] K.-J. Stol and B. Fitzgerald, "The abc of software engineering research," *ACM Trans. Softw. Eng. Methodol.*, vol. 27, no. 3, Sep. 2018, ISSN: 1049-331X. DOI: 10.1145/3241743. [Online]. Available: https://doi.org/10.1145/3241743.

[31] D. Wallace, A. H. Watson, and T. J. McCabe, "Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric," National Institute of Standards and Technology, Gaithersburg, MD, Tech. Rep. 500-235, 1996. [Online]. Available: https://www.nist.gov/publications/structured-testing-testing-methodology-using-cyclomatic-complexity-metric.

[32] G. Cloud. "Google cloud." (2024), [Online]. Available: https://cloud.google.com/?hl=en (visited on 02/21/2024).

[33] Debian. "Debian – the universal operating system." (2024), [Online]. Available: https://www.debian.org/ (visited on 02/21/2024).

[34] S. Campbell, M. Greenwood, S. Prior, *et al.*, "Purposive sampling: Complex or simple? research case examples," *Journal of Research in Nursing*, vol. 25, no. 8, pp. 652–661, 2020. DOI: 10.1177/1744987120927206. eprint: https://doi.org/10.1177/1744987120927206. [Online]. Available: https://doi.org/10.1177/1744987120927206.

[35] ReactiveX. "Rxjava: Reactive extensions for the jvm." (2023), [Online]. Available: https://github.com/ReactiveX/RxJava (visited on 02/01/2024).
[36] Mockito. "Mockito - tasty mocking framework for unit tests in java." (2024), [Online]. Available: https://site.mockito.org/ (visited on 04/12/2024).
[37] Oracle. "Thread (java platform se 8) - oracle docs." (2023), [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html (visited on 03/12/2024).
[38] F. D. Nordqvist and R. Otterlind. "Fnro.masterthesis." (2024), [Online]. Available: https://github.com/FN890/fnro.masterthesis (visited on 05/24/2024).
[39] T. J. Team. "Javaparser." (2024), [Online]. Available: https://github.com/javaparser/javaparser (visited on 02/01/2024).
[40] D. v. B. Nicholas Smith and F. Tomassetti, *JavaParser: Visited*. Lean Publishing, 2023.
[41] F. D. Nordqvist and R. Otterlind. "Jmh data." (2024), [Online]. Available: https://figshare.com/articles/dataset/JMH_data/25895269 (visited on 05/01/2024).
[42] Microsoft. "High-resolution timers." (2023), [Online]. Available: https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/high-resolution-timers (visited on 04/12/2024).
[43] eLinux. "High resolution timers." (2023), [Online]. Available: https://elinux.org/High_Resolution_Timers (visited on 04/12/2024).