UNIVERSITÀ DEGLI STUDI DI SALERNO

Dipartimento di Informatica

Corso di Laurea Magistrale in Informatica

TESI DI LAUREA

Ju2jmh: a case study on steady state reaching

RELATORE
Prof. Dario Di Nucci
Dott. Antonio Trovato
Università degli Studi di Salerno

CANDIDATO
Ludovico Lerose
Matricola: 0522501124

Anno Accademico 2023-2024

This song is new to me, but I am honored to be a part of it.

Abstract

When developing a software system, it is essential to ensure its proper operation, which depends not only on its compliance with functional requirements but also on its nonfunctional attributes, such as execution speed and throughput. Complex software systems often have constraints on such attributes to ensure an optimal user experience and, in the most critical cases, even users' safety. The compliance of the systems with these constraints can be assessed through performance testing, particularly microbenchmarking.

Despite the existence of the JMH framework in the Java landscape to help developers deal with microbenchmarks, writing them correctly and adequately requires significant effort. Therefore, research has proposed a few tools to help developers by automatically generating JMH microbenchmarks (e.g., *ju2jmh*). However, it is uncertain whether such microbenchmarks reach a steady state, i.e., a phase where performance measurements are quite stable because the Java Virtual Machine has concluded its optimizations. Hence, developers could underestimate the warm-up phase duration, leading to inaccurate results.

This thesis aims to evaluate to what extent and when the microbenchmarks automatically generated by *ju2jmh* reach a steady state. The results are compared to those obtained by analyzing handwritten JMH microbenchmarks to verify whether *ju2jmh* microbenchmarks are equally effective and efficient. The results clearly show that the microbenchmarks generated by *ju2jmh* are significantly better, in terms of stability and time taken, than those manually written by developers.

Indice

Elenco delle figure

Elenco delle tabelle

CAPITOLO 1

Introduction

This chapter briefly introduces the thesis work, its context and the motivations behind it. It also shows the structure of the thesis.

1.1 Application Context, Motivations And Goals

This thesis work is based on the application context of performance testing and consists, more precisely, of a case study where the academic tool *ju2jmh*, which automatically generates performance microbenchmarks in Java from JUnit test suites, was tested on the open source *RxJava* project to assess its potential. As will be detailed in Chapter 2, developers in fact encounter a number of difficulties when designing and implementing microbenchmarks to measure the execution times of Java methods, including the uncertainty of reaching the steady state, i.e., a phase in which measurements can be considered stable and reliable because the Java Virtual Machine has finished its optimisations. The main objective of the thesis work was therefore to evaluate the microbenchmarks automatically generated by the *ju2jmh* tool in terms of steady-state achievement, through a state-of-the-art steady-state detection technique, and to see whether this tool can be a useful resource for developers in the microbenchmarking field.

1.2 Thesis Structure

The thesis is composed by the following chapters:

- **Chapter 2** introduces and describes all the theoretical concepts needed to fully understand the thesis work; several research works related to this context are also mentioned and described to deepen the discussion.
- **Chapter 3** briefly describes the ju2jmh tool and the modifications made to conduct the case study.
- **Chapter 4** shows all the information related to how the case study was designed and conducted, including the results and threats to validity.
- **Chapter 5** summarises the conclusions and potential future developments

CAPITOLO 2

Background And Related Work

This chapter introduces the concepts of performance testing and microbenchmarking, shows the main tools used for microbenchmarking in Java, gives an overview of some difficulties that could arise and finally shows the steady state detection technique adopted.

2.1 Performance Testing And Microbenchmarking

When developing a software system, it is important to ensure its proper operation. This depends not only on its adherence to functional requirements but also on its non-functional attributes, which play a key role in determining the quality and usability of the system. Examples of these attributes are execution speed, throughput, response time, scalability, stability, and resource usage. These aspects sometimes do not receive much consideration [1, 2, 3], unlike functional behavior , thus leading to the possible occurrence of performance issues [1]. Appropriate complex software systems have precise performance constraints to ensure optimal user satisfaction and, in the most critical cases, even users' own safety. For example, a Google study [4] shows that increasing Web search latency from 100 ms to 400 ms reduces the daily number of searches by 0.2 to 0.6 percent; for longer delays, the loss of searches persists for a period even after the latency returns to previous levels. The growing importance of developing systems capable of meeting high performance standards has led to the maturation of an important branch of software engineering: **performance engineering**, whose purpose is precisely to produce software that meets non-functional requirements. More precisely, the compliance of the system to these constraints can be assessed through **performance testing**, which aims to systematically evaluate performance attributes of software systems and/or hardware resources and detect changes in performance observable by users.

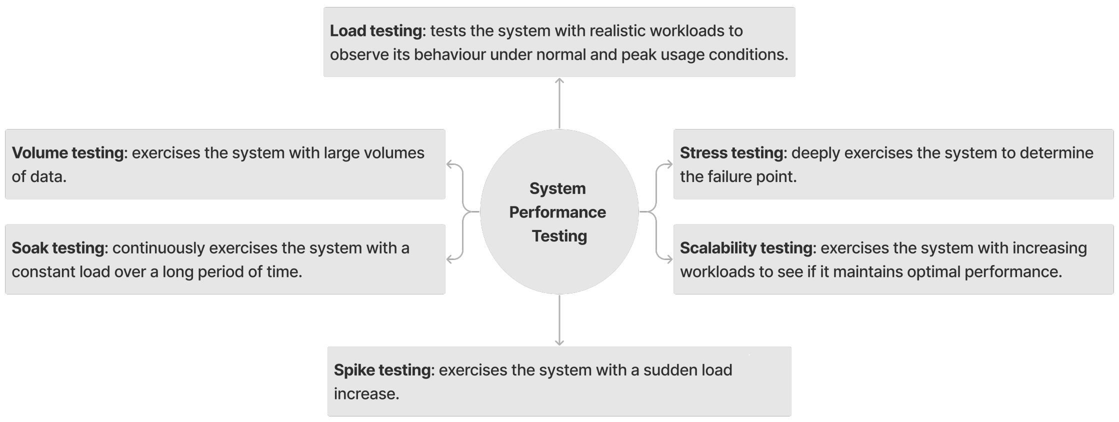
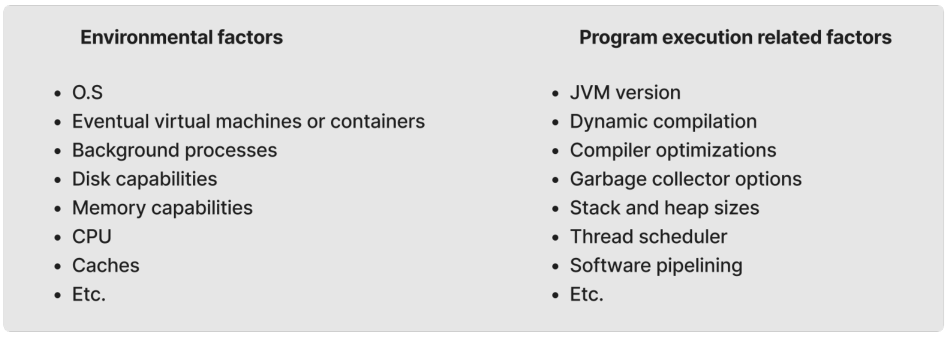

Load testing: tests the system with realistic workloads to observe its behaviour under normal and peak usage conditions.
Volume testing: exercises the system with large volumes of data.
System Performance Testing
Stress testing: deeply exercises the system to determine the failure point.
Soak testing: continuously exercises the system with a constant load over a long period of time.
Scalability testing: exercises the system with increasing workloads to see if it maintains optimal performance.
Spike testing: exercises the system with a sudden load increase.

Figura 2.1: The different types of system performance testing

Performance testing can be applied at the system level, through various techniques that evaluate end-to-end performance in different situations (shown in Figure 2.1). **Load testing** uses external loads that simulate realistic usage of the software system by multiple users accessing it concurrently, in order to collect data on metrics such as response time and throughput. Loads can simulate both normal and peak usage, allowing situations that may affect the quality of service provided by the software system to be identified in advance [5]. **Volume testing** consists in submitting huge amounts of data to the software system in order to identify the quantity that lead to system's unacceptable performance degradation. **Soak testing** (also known as *endurance testing*) consists in submitting a constant amount of data over time to the software system; the load used is usually less than the expected peak load and the aim is to identify resources, such as disk capacity or memory, that may run out over time and thus become problematic [5]. **Spike testing** verifies if the software system is able to tolerate sudden and huge amounts of data; it is useful to simulate events which suddenly attract many users such as ticket sales for a popular artist's live or Black Friday [6]. **Stress testing** involves testing the software system under extreme conditions to check its degree of robustness and performance when pushed beyond its normal operating limits. This could involve testing the software with a very high number of concurrent users or excessive data input to determine its breaking point and how it recovers from failure [5]. Finally **Scalability testing** is a form of load testing where load is incrementally scaled up to see how system's performance change with it.

Performance testing can be performed at the unit level, in which case it is called **microbenchmarking**. A microbenchmark measures in isolation the performance of small portions of code, such as methods, resembling so functional unit tests. System performance testing practices usually involve large scale, time consuming and not completely automatable tests, so they are not suitable for integration with DevOps approaches [7], where the continuous release of high quality code is essential; however, microbenchmarks provide fast feedbacks and can be easily integrated with Continous Integration pipelines, forming a performance regression suite to continuously assess whether source code changes have introduced performance degradations.

It is often the case that developers do not specifically design and write microbenchmarks in software projects to measure performance, but rather they perform implicit performance tests, i.e., performance data that are retrieved from functional tests' executions [3]. In any case, this practice is not optimal and may give inaccurate and inconclusive results. ~~In fact~~ numerous factors can influence the execution time of a software unit at a certain moment, causing the measurements about it to be nondeterministic. These factors may be related to environmental aspects or to the program execution and some of these are shown in Figure 2.2. It is therefore necessary to run the microbenchmarks several times to collect multiple observations and perform statistical analysis on these to derive meaningful information.



Environmental factors

- O.S
- Eventual virtual machines or containers
- Background processes
- Disk capabilities
- Memory capabilities
- CPU
- Caches
- Etc.

Program execution related factors

- JVM version
- Dynamic compilation
- Compiler optimizations
- Garbage collector options
- Stack and heap sizes
- Thread scheduler
- Software pipelining
- Etc.

Figura 2.2: Factors that can influence microbenchmarks' execution time

To appropriately create a microbenchmark, a performance critical portion of code must be identified, enclosed in an independent program, i.e., the payload, and be executed many times by the scaffold to evaluate the execution time [8]. The development of microbenchmarks is not a trivial activity and it requires a deep knowledge because some problems could arise otherwise, such as invalid results. The technical difficulties involved in designing and writing microbenchmarks led to the emergence of **microbenchmark frameworks** that could help developers by generating the scaffolds [8]. Currently there are many microbenchmarking frameworks for different programming languages like Java, Go, Kotlin and C++.

2.2 Java Microbenchmarking Tools

The following sections describe the main Java framework to write microbenchmarks and an existing tool in literature to generate them automatically.

2.2.1 Java Microbenchmark Harness

Java Microbenchmark Harness (**JMH**) is ~~the facto~~ standard microbenchmarking framework in Java ecosystem and it offers various Java annotations to configure

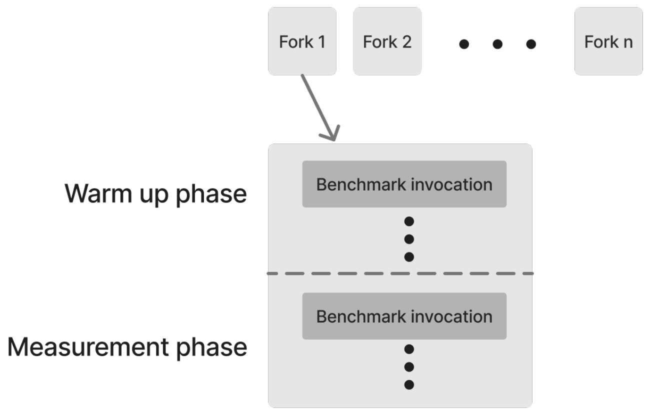


Figura 2.3: The JMH microbenchmark life cycle

microbenchmarks' behaviour. Microbenchmarks are continuously executed within a predefined amount of time, i.e., an **iteration**, and sequences of consecutive iterations on a separate, new instance of JVM constitute a **fork** [9]. Two kinds of iteration are used: **warmup** and **measurement**; data obtained by the former iteration type are discarded since they could be inaccurate, while the latter supplies the actual results which are collected. Typically microbenchmarks go through firstly a warmup phase, where warm up iterations occur, and then the measurement iterations follow. The reason behind this practice is that initially the benchmarks' execution times are high and unstable because of Java Virtual Machine (**JVM**); then the JVM detects frequently executed segments of code and dynamically compiles them into optimized machine code, starting the phase of microbenchmark's **steady state of performance**, where execution times are lower and more stable than the previous ones [9]. Figure 2.3 gives an overview of the whole process, while Figure 2.4 and Figure 2.5[1] show respectively a microbenchmark which reach a steady state of performance and a microbenchmark which does not.

[1]both derived from the following paper: `https://doi.org/10.1007/s10664-022-10247-x`

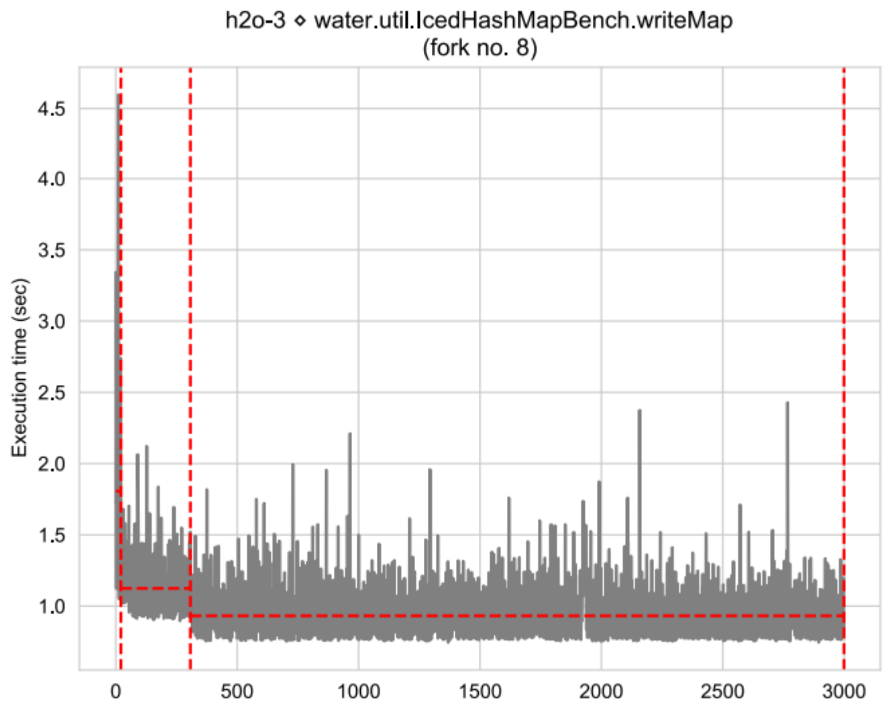


Figura 2.4: Example of stable microbenchmark execution

The fundamental annotation supplied by JMH is **@Benchmark**, which is used to denote the payloads that will be repeatedly executed and whose measurements will be collected. Then important benchmark related annotations are **@Fork**, whose parameter determines the number of forks, **@Warmup**, which indicates the number of warmup iterations and their duration, and **@Measurement**, which is the same but for measurement iterations. By default JMH uses 5 forks, 5 warm up iterations that last 10 seconds each and 5 measurement iterations with the same duration. Furthermore, JMH supports different measurement modes, that can be specified through the **@Mode** annotation: Throughput, which is the default one, AverageTime, SampleTime, which allows to infer informations like distribution and percentiles, SingleShotTime, which measures a single benchmark method invocation per iteration, and All, which comprises all the mentioned modes. There are some other annotations, such as @OutputTimeUnit or @Threads for example, that can be used depending on the needs. All these configuration parameters, expressed through Java annotations, can also be directly overridden by CLI arguments when benchmarks are executed. Figure 2.6 shows an example of microbenchmark.

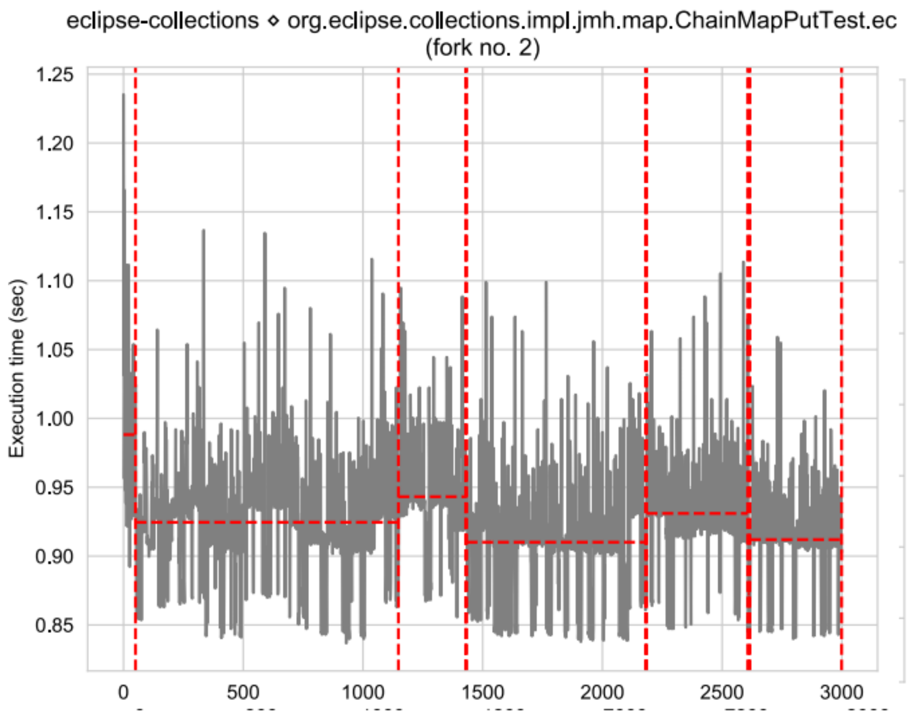
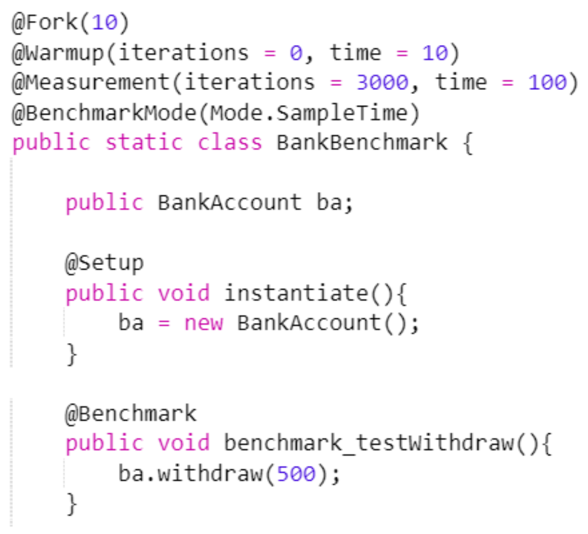


Figura 2.5: Example of unstable microbenchmark execution.

2.2.2 Automatic Microbenchmark Generation And AUTOJMH

Writing microbenchmarks is not a popular activity among developers. Leitner and Bezemer [3] have considered 111 Java based open source projects, discovering that only 36% of these use microbenchmarks and 16% use dedicated frameworks like JMH; furthermore, it was found that in 48% of the projects, only one developer was responsible for the microbenchmarks and these were usually written once and rarely updated. The authors of the study suggest that microbenchmarking is overlooked because developers may think it is challenging to appropriately write microbenchmarks because of various problems. In fact, although frameworks such as JMH generate the scaffold, the appropriate creation of the payload remains difficult [8]. According to Rodriguez-Cancio et al. [8], developers tend to repeat two mistakes very frequently, i.e., they do not design the payload in such a way that the Just In Time (**JIT**) compiler does not perform dead code elimination and constant folding: these actions cause the payload execution to be over-optimised and thus the measured times do not reflect the actual execution time of the code segment. Other problems mentioned in the study are the choice of irrelevant initialisation values and unrealistic steady states.



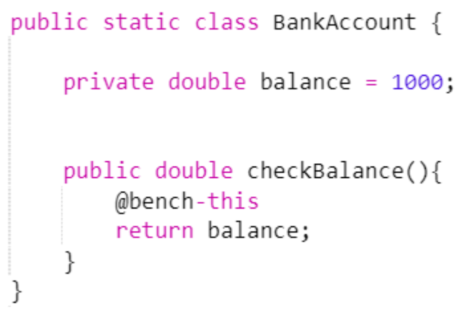
```
@Fork(10)
@Warmup(iterations = 0, time = 10)
@Measurement(iterations = 3000, time = 100)
@BenchmarkMode(Mode.SampleTime)
public static class BankBenchmark {

    public BankAccount ba;

    @Setup
    public void instantiate(){
        ba = new BankAccount();
    }

    @Benchmark
    public void benchmark_testWithdraw(){
        ba.withdraw(500);
    }
```

Figura 2.6: Example of JMH microbenchmark



```
public static class BankAccount {

    private double balance = 1000;


    public double checkBalance(){
        @bench-this
        return balance;
    }
}
```

Figura 2.7: Example of statement marked by AUTOJMH annotation

Leitner and Bezemer [3] suggest that the use of tools that automatically generate microbenchmarks, speeding up the activity and avoiding human errors in design and configuration, could bring developers closer to this practice. The automatic microbenchmarks generation is still a very overlooked topic by research, in fact only a couple of tools has been presented in literature so far. Rodriguez-Cancio et al. [8] proposed **AUTOJMH**, which automatically generates compilable payloads for JMH from certain annotated code segments[2], like the one shown in Figure 2.7; the generated payloads also avoid or mitigates the previously mentioned problems. The experiments, performed on five large Java projects, showed that the microbenchmarks automatically generated by AUTOJMH systematically perform as well as the benchmarks constructed by JMH experts with a confidence level of 0.05; furthermore, the microbenchmarks generated by AUTOJMH avoid errors commonly made by

[2]The annotation is @bench-this

Java developers without prior microbenchmarking experience. However, the major limitation of this tool is that it is not fully automated, since code segments of interest have to be manually annotated; furthermore, it is now obsolete.

2.3 Problems Related To Java Microbenchmarking

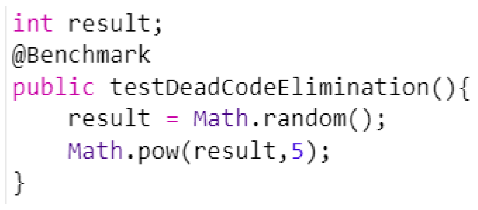
Although JMH helps developers to create microbenchmarks, there are some difficulties that must be addressed by developers. The following sections describe the difficulty of setting the right duration for the warm-up phase and the most common microbenchmark design pitfalls.

2.3.1 Steady State Performance Analysis

Currently JMH developers statically set the duration of the warm-up phase according to their experience-driven predictions. However, making accurate predictions is not easy; an overestimation can lead to a waste of computation time while, on the other hand, an underestimation leads to measurements that do not represent the steady state and are therefore unstable and inaccurate [9]. The state-of-the-art technique for verifying whether and when a benchmark has reached the steady state phase was proposed by Barrell et al. [10], which is based on a standard change point detection algorithm called **PELT** [11]. It is because of the difficulties with static estimations by developers that Laaber et al. [12] have recently proposed an alternative approach, called dynamic reconfiguration, to automatically detect the end of the warm-up phase at run time.

The first comprehensive study on steady state performance assessment about Java microbenchmarks was conducted by Traini et al. [9]. The study examined 586 JMH benchmarks from 30 Java systems and focused on verifying when the benchmarks reach the steady state, evaluating the effectiveness of developers' static estimates and dynamic reconfiguration techniques, and finally quantifying the consequences of inaccurate warm-up duration estimates. Based on the obtained results, the authors of the study recommend that the warm-up phase should last at least 5 seconds, although the same study also revealed that actually the achievement of steady state is not guaranteed: approximately 11% of the involved microbenchmarks in fact never reaches it. Furthermore, the results show that the estimates provided by developers are very often inaccurate and tend to overestimate the time spent in the warm-up phase. Finally, it appears that dynamic techniques perform better than static configurations by developers, although the resulting estimates are still inaccurate.

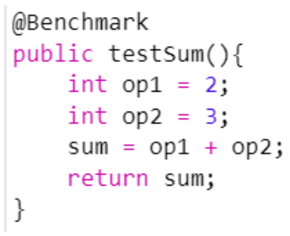
2.3.2 Over Optimisation Applied By The JIT Compiler



```
int result;
@Benchmark
public testDeadCodeElimination(){
    result = Math.random();
    Math.pow(result,5);
}
```

Figura 2.8: Example of microbenchmark with dead code

Despite JMH's support in writing Java microbenchmarks, it is still possible for developers to face pitfalls when designing payloads and these are often related to the over optimisation applied by the JVM's JIT compiler; when this happens, the microbenchmarks have lower execution times than the corresponding original code segments, on which the compiler does not apply these same optimisations, thus invalidating the measurements. **Dead code elimination** is one such unwanted optimisation and it occurs when designing a payload where an invoked method has no side effects and its returned value is not used, as is the case with the Math.pow invocation shown in the Figure 2.8. In such situations, the JIT compiler identifies the method invocation as dead code and will generate optimised machine code that will not include that instruction. There are essentially three ways to avoid this problem [8]: let the microbenchmark return the result value of the invoked method, save any results in public fields or use the black hole methods provided by JMH, which consume the specified parameters and so avoid dead code elimination. The first approach is safe and fast but can not be applied to cases where the method being tested is void, the second approach is still fast but less safe while the third is the safest by far as it does not alter the state of the microbenchmark; furthermore, it does not overly impact its performance.



```
@Benchmark
public testSum(){
    int op1 = 2;
    int op2 = 3;
    sum = op1 + op2;
    return sum;
}
```

Figura 2.9: Example of microbenchmark affected by constant folding

Another undesirable optimisation is **constant folding**. In microbenchmarks, variables are often initialised with constants and when the JIT compiler identifies a computation based on constant values, it may omit that computation and replace it directly with its result, as it would occur to the microbenchmark represented in Figure 2.9. To avoid this, constant values have not to be used as inputs in microbenchmarks but rather objects' states have to be referred. It is necessary to point out that constant folding is not something to be avoided at all costs; it is sufficient to make sure that the microbenchmark reflects the same execution conditions as the original code segment, so if this optimisation is applied in the original program then it should be applied also in the microbenchmark. It is important that developers are aware of these compiler optimizations to appropriately write microbenchmarks.

2.4 Steady State Detection Technique

The JMH fork-level steady state detection technique employed in this work is described by Traini et al. [9] and it consists in an adaptation to the current experimental context of a technique originally proposed by Bennett et al. [10] , which is based on **changepoint analysis**. Changepoint analysis is a statistical analysis conducted on timeseries data to derive different **segments**, that are groups of observations that we consider having the same behaviour, and data points (i.e., **changepoints**) where shifts between segments occur. In this case, a timeseries represents a fork and each timeseries observation represents the average execution time within a JMH iteration,

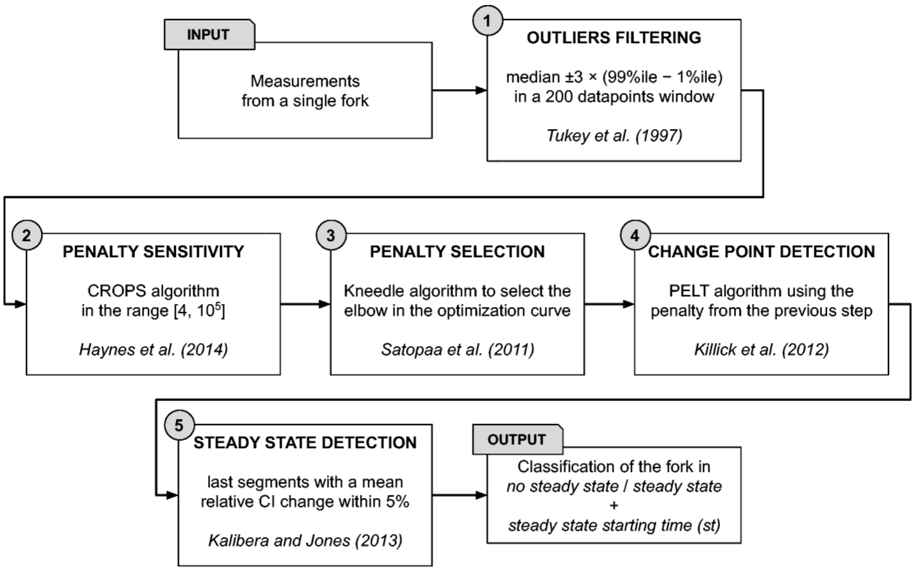

INPUT
Measurements from a single fork
1
OUTLIERS FILTERING
median ±3 × (99%ile − 1%ile) in a 200 datapoints window
Tukey et al. (1997)
2
PENALTY SENSITIVITY
CROPS algorithm in the range [4, 10^5]
Haynes et al. (2014)
3
PENALTY SELECTION
Kneedle algorithm to select the elbow in the optimization curve
Satopaa et al. (2011)
4
CHANGE POINT DETECTION
PELT algorithm using the penalty from the previous step
Killick et al. (2012)
5
STEADY STATE DETECTION
last segments with a mean relative CI change within 5%
Kalibera and Jones (2013)
OUTPUT
Classification of the fork in no steady state / steady state + steady state starting time (st)

Figura 2.10: Steps of the steady state detection technique

so each timeseries contains 3000 data points, since the forks in this experiment go through 3000 measurement iterations of 100 ms each, as explained in more detail in section 4.2.2.

More specifically, the algorithm used for changepoint analysis is called **PELT** [11] and it detects variations in timeseries by considering both the mean and the variance of the execution time. Figure 2.10[3] represents the various steps of the employed steady state detection technique. First of all, filtering of outliers is applied using the method defined by Tukey [13], i.e., by considering windows of 200 datapoints and excluding those that lie outside the range defined by $median \pm 3 \times (99\%ile - 1\%ile)$. Among the various parameters of the previously mentioned PELT algorithm there is the **penalty** that, as well as allowing us to avoid under/over fitting, allows us also to determine the sensitivity of the algorithm in detecting changepoints, i.e., the number of changepoints detected is inversely proportional to its value. A fixed penalty value is not used for all timeseries, due to the differences that can distinguish them, but rather the most suitable penalty value for each timeseries is adopted. To do this, it is first necessary to use the **CROPS** algorithm [14] on each timeseries to generate the optimal segmentations for each penalty value within the continuous range $[4, 10^5]$. Given a certain timeseries, the number of changepoints identified

[3]derived from the following paper: `https://doi.org/10.1007/s10664-022-10247-x`

and the corresponding penalty values form an optimisation curve and, following Lavielle's suggestion [15] , a penalty point at the elbow area should be chosen: this point can be automatically selected via **Kneedle's algorithm** [16], which selects the point of maximum curvature in an optimisation curve. Then the PELT algorithm is run with the penalty value just returned by the Kneedle algorithm and it produces a segmentation. The last step consists in the actual steady state detection on the fork. Replicating what Barrett et al. [10] did, the fork is labelled as stable if the last 500 data points are part of the same segment or at least segments that are fairly similar to each other, i.e., whose variance is not significant. The segments are compared using the technique of Kalibera and Jones [17], i.e., a tolerance of 5% is applied for the mean relative performance change respect to the average of the final segment; a percentage is prefered by Traini et al. [9] over a fixed threshold in units of time, contrary to what Bennett et al did with the chosen value of 0,001 s, due to potentially high variability in the scale of microbenchmarks' execution times. If the last 500 data points are not part of similar segments, then the fork is labelled as unstable; in addition, if the fork is stable, the time taken to reach the steady state, i.e., the time beginning of the first segment that can be considered stable, will also be reported.

In this experiment, as described in section 4.2.2, microbenchmarks are run with ten JMH forks: a benchmark is labelled as **stable** if all its forks reach the steady state, **unstable** if none of its forks reach the steady state, and **mixed** if only a few forks reach the steady state. The adopted classification of benchmarks as well is the same of that described by Traini et al. [9]

CAPITOLO 3

Ju2jmh Overview

This chapter briefly presents *ju2jmh* along with its features, its automatic micro-benchmark generation process and its limitations. Furthermore, it shows also the preliminary adjustments that had to be made before the start of the study.

3.1 Overview

Jangali et al. [7] proposed, in the context of automatic microbenchmark generation, **ju2jmh**, a tool integrated into the Gradle build system which automatically derives JMH microbenchmarks from a suite of JUnit4 functional unit tests. The tool goes through two steps: **Analysis** and **Benchmark generation**. Figure 3.1 shows an high level representation of this process. In the analysis step, *ju2jmh* analyzes the input test classes and detects the test methods (annotated with @Test), which will be converted into JMH benchmarks, along with other test features that are necessary for a correct execution, such as fixture methods, test rules, expected exceptions and ancestor classes. The analysis is practically executed through *Apache Commons BCEL* to inspect the bytecode of JUnit input test classes. Then in the benchmark generation phase, *ju2jmh* essentialy generates a copy of each JUnit input test class with a nested benchmark class, which contains a benchmark method for each test method in the

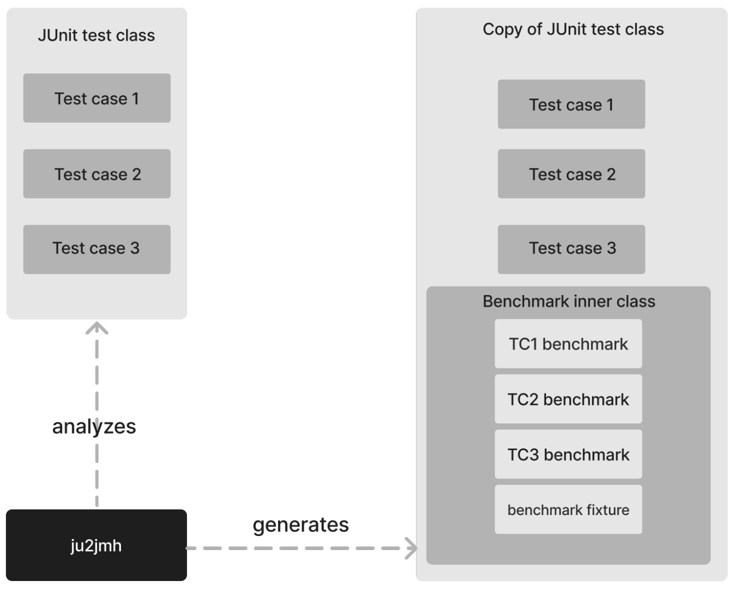


Figura 3.1: ju2jmh overview

outer class. More precisely the generation is performed through the *Java Parser* library; firstly the tool parses junit test classes into **Abstract Syntax Trees**[1] (AST), then it make copies of them, it generates the ASTs for corresponding nested benchmark classes using the information obtained in the analysis step, it inserts these new ASTs into the copied ones as static inner classes and finally it writes the results to Java source files. All the inner benchmark classes extend *JU2JmhBenchmark*, which defines methods that will be used to properly run JUnit tests within the benchmarks. The experiments, conducted on 3 large Java projects, showed that the microbenchmarks generated by *ju2jmh* are superior to those generated by AutoJMH and comparable with those handwritten by experts in terms of performance bugs' detection capability and stability; however, the microbenchmarks generated by *ju2jmh* cover more code than the handwritten ones [7]. Nevertheless the tool presents some limitations:

- It supports only the JMH default configuration values.

[1]Tree data structure where each node represents a construct used in a source code program

- a JUnit test suite must necessarily exist to generate microbenchmarks.
- It converts every single test method of the input classes into microbenchmarks, including the trivial ones.
- Only JUnit4 is supported, while JUnit5 is not.

The thesis work focuses on and resolves only a part of the limitations listed above.

3.2 Tool Adjustments

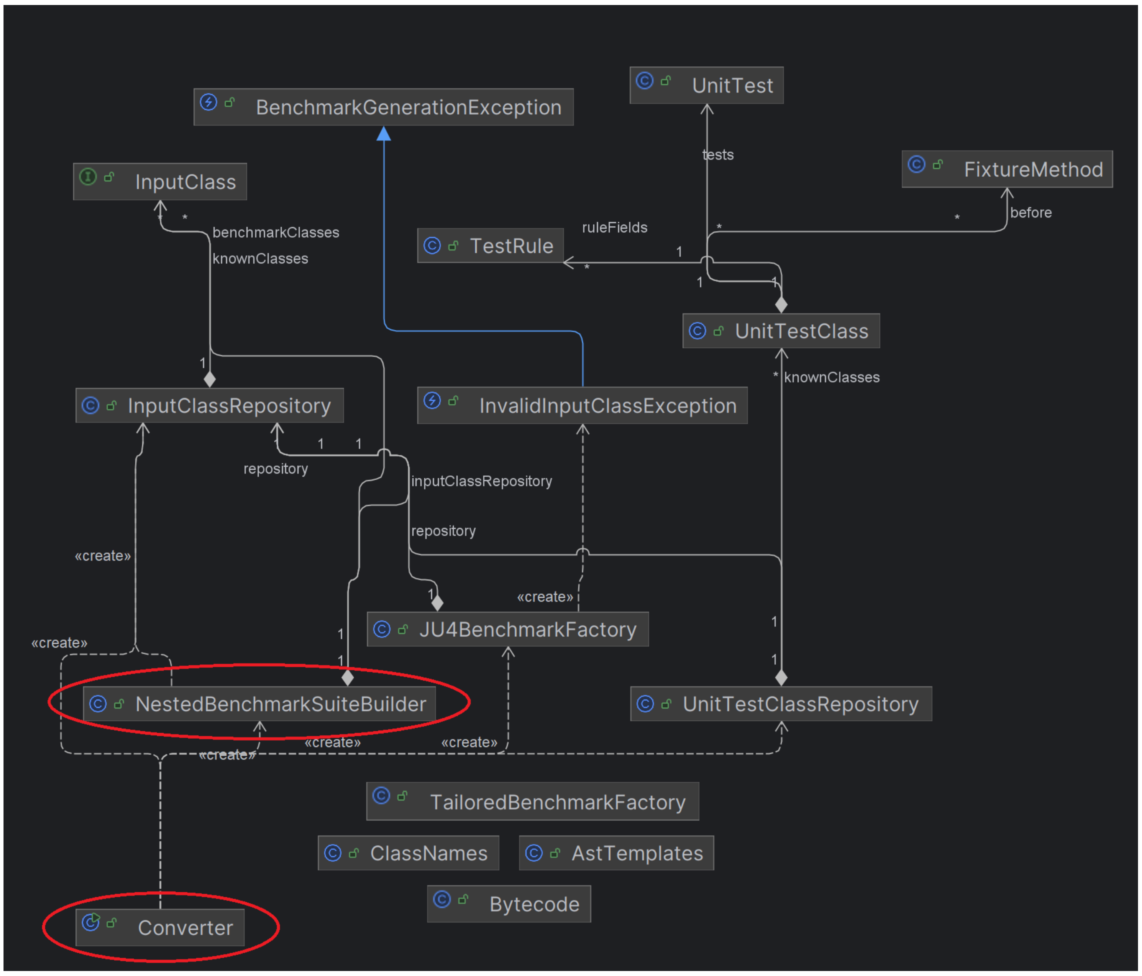


Figura 3.2: Class diagram of *ju2jmh*

The limitation regarding the lack of support for non-default configuration values was resolved. Resolving this limitation was a priority as otherwise it would not have been possible to conduct the experiment as designed. So, before the study began, the *ju2jmh* tool was modified to support the customisation of some JMH configuration options for the microbenchmarks that will be generated, such as number of forks, number of warm-up iterations, number of measurement iterations, duration of measurement iterations, measurement unit and benchmarking mode.

Figure 3.2 shows the class diagram of *ju2jmh*; the classes circled in red constitute the Starting Impact Set (SIS), because modifications essentially consist in two operations:

- Make the tool support new options from the command line
- Make the tool explicitly insert lines of code, more precisely JMH annotations with values related to the previosly cited options, into the benchmark inner classes.

and these two classes are in fact related to these operations. The Candidate Impact Set (CIS) coincides with the SIS because no class is dependent on these two, as shown in the diagram.

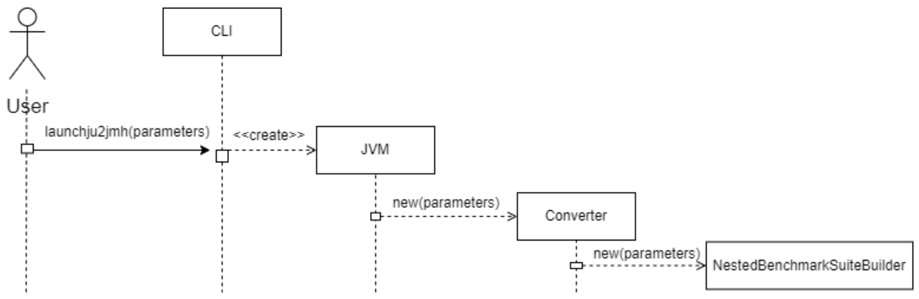


Figura 3.3: Sequence diagram that represents the execution flow between the Converter and NestedBenchmarkSuiteBuilder classes

The *Converter* class, which receives command line arguments, was modified. The arguments in the class are managed and controlled through *picocli*, a framework that makes it easy to build Java applications that make use of the command line. In addition, the *NestedBenchmarkSuiteBuilder* class, which gets instantiated by *Converter*, was modified to receive the new parameters from *Converter* and, using the *Java Parser* library, add the appropriate annotations to the internal classes (in which the benchmarks are defined) that will be generated by the tool. Figure 3.3 shows a sequence diagram that depicts the execution flow between the two classes.

After the change, a regression test was conducted with all test cases of the project to ensure that the new feature did not introduce any bugs. All tests passed and thus it was confirmed that the changes did not impact any other classes in the project.

CAPITOLO 4

Case Study

This chapter shows the research questions that guided the thesis work and how the experiment was designed and conducted, along with its results and threats to validity.

4.1 Research Questions

The research questions on which the thesis is based are listed below.

🔍 **RQ$_1$.** *Are the JMH microbenchmarks automatically generated by ju2jmh more effective and efficient than those handwritten by developers to achieve a steady state?*

🔍 **RQ$_{1.1}$.** *To what extent do JMH microbenchmarks hand-written by developers reach a steady state and when?*

🔍 **RQ$_{1.2}$.** *To what extent do JMH microbenchmarks automatically generated by ju2jmh reach a steady state and when?*

The technique described by Traini et al. [9] is utilized to find out if and when forks reach the steady state and answer all the mentioned research questions. This technique is a modified and adapted version of an approach originally proposed by

Bennett et al. [10]. The classification of benchmarks is described by Traini et al. [9] and benchmarks can be labelled, as previously said, as stable, unstable or mixed.

4.2 Experimental Design

This section describes various aspects of the experiment such as the chosen subject, the microbenchmarks' setup, the adopted steady state detection technique, the modifications applied to *ju2jmh* and the experiment's execution environment.

4.2.1 Subject

The open source project chosen as the subject of the experiment is *RxJava*, whose repository is available on GitHub[1]. This project presents an high number of stars (i.e., 47.759 at the time of writing), have an adequate number of JMH microbenchmarks for an experiment and is still quite supported and updated. The adopted version is the 3.1.8.

The project's Junit test suite contains too many test classes, i.e., 943, which would lead to excessive execution times of the related microbenchmarks generated by ju2jmh (a single microbenchmark method would take 50 minutes); therefore a statistically significant sample, i.e., a subset that is supposed to reflect the behaviour of the whole population not by chance, is randomly selected among both the already existent JMH microbenchmarks' suite and the JUnit test suite, with a confidence level of 95% and a margin of error of 5%. The formula used to compute the sample size is the following:

$$\frac{\frac{z^2 \times p(1-p)}{e^2}}{1+\frac{z^2 \times p(1-p)}{e^2 N}}$$

where N is the population size, e is the margin of error and z is the z-score[2].
Then, 44 JMH microbenchmarks and 274 JUnit test classes have been selected and used for the experiment. The handwritten and the automatically generated microbenchmarks needed about 25 days to complete their execution despite the use of

[1] `https://github.com/ReactiveX/RxJava`

[2] The number of standard deviations a given proportion is away from the mean. Its value depends on the confidence level. In this case it is equal to 1.96

six different servers, which is the reason why the study focused on a single subject project.

4.2.2 Microbenchmark Setup

Based on the information provided by Bennett et al. [10] and Traini et al. [9], each microbenchmark used in the experiment is run with ten JMH forks and each of these forks will have 3.000 measurement iterations of 100 ms each and no warm-up iterations to gather all the taken measurements; a single fork will therefore take 3.000 seconds to complete execution and a microbenchmark method, with all its forks, will take instead fifty minutes. Sample mode is used as the benchmarking mode. The JMH CLI arguments are used to apply this configuration to existing JMH microbenchmarks, whereas the microbenchmarks generated by *ju2jmh* will present these configuration values directly in the code through annotations thanks to a modified version of the tool that supports additional generation options unlike the base version.

4.2.3 Execution Environment

The workload was distributed on six bare metal servers in the cloud. All servers have the same specifications, namely an Intel Xeon E3-1231v3 processor with a clock frequency of 3.4 GHz, a 32 GB RAM based on DDR3 technology and two SSD storage units with 480 GB. Ubuntu Server 24.04 Noble Numbat LTS was installed as the operating system and Java 11 OpenJDK was used as the Java Development Kit (JDK) implementation.
As suggested by Traini et al. [9], a number of precautions were taken to eliminate possible interference in the execution environment so that the results of the microbenchmarks' execution would be as reliable as possible. First of all, the Turbo Boost mode, usually used by Intel processors to increase the clock frequency for particularly demanding tasks, was deactivated; hyperthreading, which allows multiple threads to run on a single physical core at the cost, however, of sharing execution resources and thus leading to contention that could affect execution times, was also disabled. Then, Address Space Layout Randomization (ASLR), which is a memory address

space management security technique, was disabled because it could as well affect measurements between forks. The maximum heap size of the JVM was set at 24 GB via the *-Xmx* flag to avoid memory fills and consequent garbage collector operations. The priority of the JVM process was set to the maximum value, i.e., -20, to reduce the overhead associated with context switching.
Finally, since the only way to interact with the servers is via an SSH connection, the *nohup* command was used to ensure that the JVM process was not killed on logout but rather continued to run in the background. The execution of all microbenchmarks, both handwritten and automatically generated, took about 25 days to complete.

4.3 Results

Benchmark type	**Stable forks**	**Unstable forks**	**Stable benchmarks**	**Mixed benchmarks**	**Mean time**
Handwritten	78,46%	21,54%	16,42%	83,58%	124,290 ms
Generated	**97,80%**	**2,20%**	**86,29%**	**13,71%**	**45,996 ms**

Tabella 4.1: Resulting statistics

To answer the research questions, it is first necessary to see what percentage of the forks and microbenchmarks, considering both those handwritten by the developers and those automatically generated by *ju2jmh*, have reached the steady state and then compare their results. It is recalled that forks have been labelled as stable or unstable through the adopted fork-level steady state detection technique, previously described in section 2.4, and microbenchmarks, which present ten forks, are labelled as **stable** if all its forks are stable, **unstable** if every fork is unstable and **mixed** in every other case. Table 4.1 show both the percentages of stable and unstable forks and the percentages of stable and mixed-behaving microbenchmarks, according to the labelling procedure previously described. It can first be noted that both groups do not present completely unstable microbenchmarks (i.e., with all ten forks labelled as *unstable*). Then, an important difference can be noted between the percentages of stable forks between the two groups (78.46% for the group of handwritten microbenchmarks compared to 97.80% for the group of ju2jmh-generated microbenchmarks): **almost all forks in the group of automatically generated microbenchmarks achieve stability.** Furthermore, although the group of hand-written microbenchmarks presents a rather high percentage of stable forks, there is a **large difference** between the two groups respect to the percentages of microbenchmarks labelled as stable (16.42% in the first group and 86.29% in the second) and the percentages of microbenchmarks labelled as mixed (83.58% versus 13.71%).

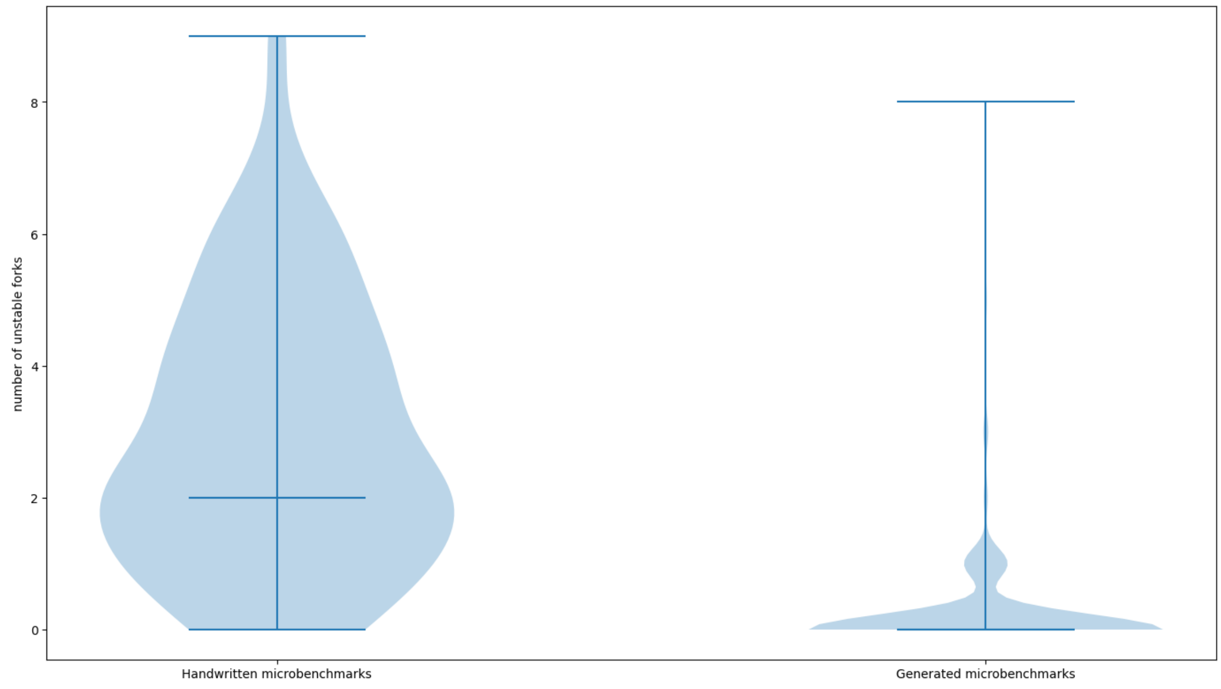


Figura 4.1: Distribution density respect to number of unstable forks in microbenchmarks

Figure 4.1 shows the distribution densities of microbenchmarks according to the number of unstable forks, and it can be seen that in the first group, the number of microbenchmarks with exactly one unstable fork is greater than the number of microbenchmarks with all stable forks; **the highest distribution density occurs at microbenchmarks with exactly two unstable forks**, and from three unstable forks onwards the distribution density gradually starts to decrease. Thus, many microbenchmarks present just one or two unstable forks, which is enough to label them as unstable in their entirety. It can then be seen in the distribution density of the second group that most of the microbenchmarks have no unstable forks, confirming the previously mentioned percentage, with only a few outliers.

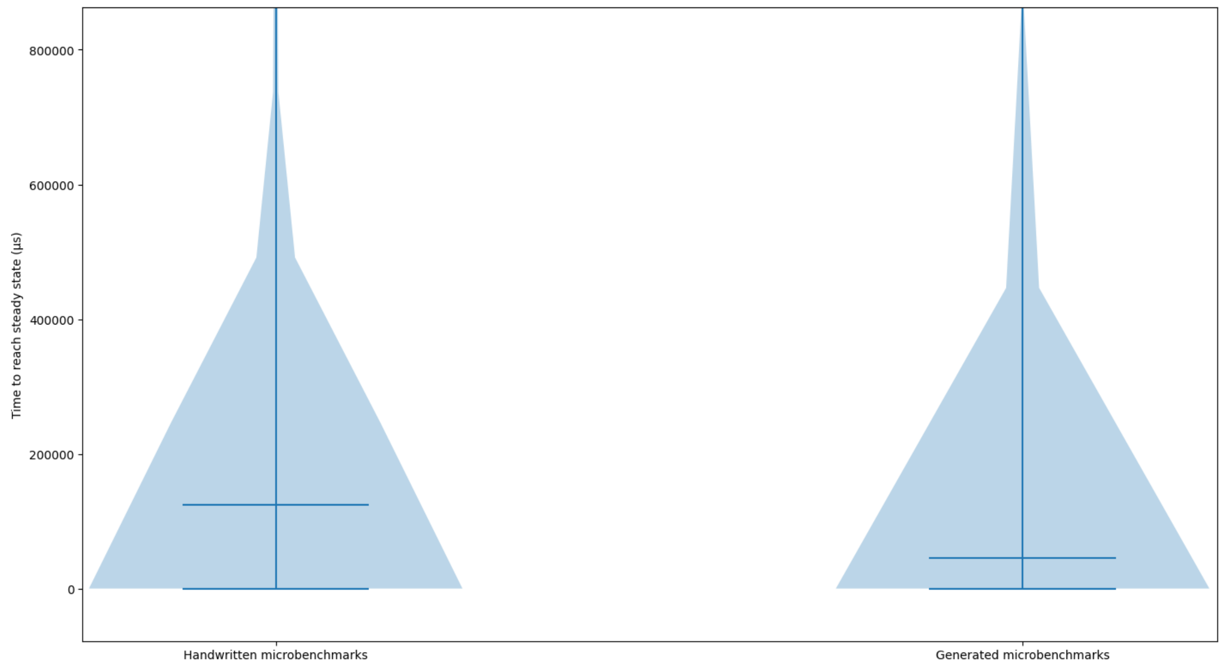


Figura 4.2: Distribution density respect to time required to reach the steady state

Finally, Figure 4.2 shows the distribution of the times taken by the microbenchmarks to reach the steady state (if they have reached it). it can be seen that the graphs for the two groups are very similar, but the mean of the second group is lower than the mean of the first: in fact, the average amount of time required to reach steady state in the first group is 124.290 ms while in the second group it is 45.996 ms.

It is now possible to answer the research questions:

☞ **Answer to RQ$_{1.1}$.** Microbenchmarks written by developers are mostly inconsistent in terms of stability, which is due to the fact that many of them have just one or two forks labelled as unstable.

☞ **Answer to RQ$_{1.2}$.** The microbenchmarks automatically generated by *ju2jmh* are mostly stable, as most of them have all stable forks, and microbenchmarks with one or more unstable forks constitute a small exception.

☞ **Answer to RQ$_1$.** in the specific case of Rxjava, the microbenchmarks generated automatically by *ju2jmh* are significantly better than those manually written by developers, both in terms of the percentage of stable microbenchmarks and forks identified and the time taken.

Such a marked difference in results should be investigated by replicating the experiment on other projects to see if this phenomenon is repeated or is specifically related to *RxJava*. Therefore, replication of the experiment on other projects to verify this fact would be the priority objective for any future developments. At the moment, it can be assumed that this difference is probably due to the nature and design of the JUnit test cases, which tend to exercise more trivial methods (e.g. the correct insertion of an element in a queue) than those exercised by the JMH microbenchmarks manually written by developers.

4.4 Threats To Validity

Some potential threats to validity are listed below, grouped by category.

Construct Not enough time or measurement iterations may have been given for all potentially stable forks to reach the steady state. This risk is certainly mitigated by the large number of measurement iterations a fork goes through, which causes each microbenchmark to be invoked at least 3000 times per fork, a far higher number than the JMH defaults or the values commonly used by developers.

Internal The experiment was run through six different servers, which, although shared the same hardware, operational and configuration features, they were not under the direct physical control of the experimenter, rather they were only remotely accessible via SSH. The SSH logins themselves could have affected the experiment in some way, which is why the experimenter has tried to limit them as much as possible until the end of the microbenchmarks' execution and only log in if there was an important need.

External In order to generalise, this work should obviously be repeated on further projects, possibly of a different size and nature respect to *RxJava*. Furthermore, the number of classes involved, especially that of the JMH microbenchmarks, may not be high enough to generalise the results, despite the fact that a project with an average number of JMH microbenchmarks was chosen. Another important factor that could influence the generalisation of the experiment's results is the extremely vary and heterogeneous nature of tests, which can generally exercise profoundly different methods, taking from a few nanoseconds to several seconds in the most demanding cases; it must also be considered that the microbenchmarks generated by *ju2jmh* invoke essentially functional JUnit test cases, so the difference in how ad hoc JMH microbenchmarks and JUnit test cases are designed is another factor that should not be overlooked.

CAPITOLO 5

Conclusions

So, in conclusion, the thesis work revealed that, in the specific case of *RxJava*, the microbenchmarks automatically generated by *ju2jmh* from the JUnit functional test case suite are much more stable, and thus reliable, than those already present, hand-written by the developers. As already mentioned, the study should be repeated on other projects to validate these results, in which case the tool would prove to be an important resource for developers, helping them to quickly create a large suite of quality microbenchmarks.

The tool can also be further improved, and some potential future developments include support for JUnit5 or the addition of a selection mechanism to filter out trivial test methods (e.g., test methods that exercise getters and setters) and prevent them from having dedicated microbenchmarks. Another interesting opportunity for the evolution of the tool could be to support additional ways of generating microbenchmarks, e.g., by relying on meta-heuristic search criteria in such a way that a starting functional test case suite would no longer even be necessary.

Bibliografia

[1] E. J. Weiuker and F. I. Vokolos, "Experience with performance testing of software systems: Issues, an approach, and case study," *IEEE Transactions on Software Engineering*, vol. 26, no. 12, pp. 1147–1156, December 2000. (Citato a pagina 3)

[2] C.-P. Bezemer, S. Eismann, V. Ferme, J. Grohmann, R. Heinrich, P. Jamshidi, W. Shang, A. van Hoorn, M. Villavicencio, J. Walter, and F. Willnecker, "How is performance addressed in devops?" in *ICPE '19: Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, 2019, pp. 45–50. (Citato a pagina 3)

[3] P. Leitner and C.-P. Bezemer, "An exploratory study of the state of practice of performance testing in java-based open source projects," in *ICPE '17: Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, 2017, pp. 373–384. (Citato alle pagine 3, 5, 9 e 10)

[4] J. Brutlag, "Speed matters for google web search," June 22 2009. [Online]. Available: https://blog.research.google/2009/06/speed-matters.html (Citato a pagina 3)

[5] B. Wescott, *The every computer performace book*. Fraser Publishing Company, 2013. (Citato alle pagine 4 e 5)

[6] "Spike testing: A beginner's guide," January 30 2024. [Online]. Available: https://grafana.com/blog/2024/01/30/spike-testing (Citato a pagina 5)

[7] 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, April 2023. (Citato alle pagine 5, 16 e 17)

[8] M. Rodriguez-Cancio, B. Combemale, and B. Baudry, "Automatic microbenchmark generation to prevent dead code elimination and constant folding," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2016, pp. 132–143. (Citato alle pagine 6, 9, 10 e 12)

[9] L. Traini, V. Cortellessa, D. D. Pompeo, and M. Tucci, "Towards effective assessment of steady state performance in java software: are we there yet?" *Empirical Software Engineering*, vol. 28, no. 1, pp. 1–57, January 2023. (Citato alle pagine 7, 11, 13, 15, 21, 22 e 23)

[10] E. Barrett, C. F. Bolz-Tereick, R. Killick, S. Mount, and L. Tratt, "Virtual machine warmup blows hot and cold," *Object-Oriented Programming, Systems, Languages Applications*, no. 52, pp. 1–27, October 2017. (Citato alle pagine 11, 13, 15, 22 e 23)

[11] R. Killick, P. Fearnhead, and I. Eckley, "Optimal detection of changepoints with a linear computational cost," *Journal of the American Statistical Association*, vol. 107, no. 500, pp. 1590–1598, October 2012. (Citato alle pagine 11 e 14)

[12] C. Laaber, S. Würsten, H. C. Gall, and P. Leitner, "Dynamically reconfiguring software microbenchmarks: reducing execution time without sacrificing result quality," in *ESEC/FSE 2020: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 989–1001. (Citato a pagina 11)

[13] J. W. Tukey, *Exploratory data analysis Vol.2*. Addison-Wesley Publishing Company, 1977. (Citato a pagina 14)

[14] K. Haynes, I. A. Eckley, and P. Fearnhead, "Efficient penalty search for multiple changepoint problems," *arXiv:1412.3617*, December 2014. (Citato a pagina 14)

[15] M. Lavielle, “Using penalized contrasts for the change-point problem,” *Signal Processing*, vol. 85, no. 8, pp. 1501–1510, August 2005. (Citato a pagina 15)

[16] V. Satopaa, J. Albrecht, D. Irwin, and B. Raghavan, “Finding a "kneedle" in a haystack: Detecting knee points in system behavior,” in *2011 31st International Conference on Distributed Computing Systems Workshops*, 2011. (Citato a pagina 15)

[17] T. Kalibera and R. Jones, “Rigorous benchmarking in reasonable time,” *ACM SIGPLAN Notices*, vol. 48, no. 11, pp. 63–74, June 2013. (Citato a pagina 15)

Ringraziamenti

Ringrazio chiunque abbia fatto parte del mio percorso negli ultimi due anni.