CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Optimizing the generation of Java JMH Benchmarks

Master's thesis in Computer science and engineering

Anthony Path

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2022

Master's thesis 2022

Optimizing the generation of Java JMH Benchmarks

Anthony Path

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2022

Optimizing the generation of Java JMH Benchmarks
Anthony Path

Supervisor: Philipp Leitner, Department of Computer Science and Engineering
Examiner: Regina Hebig, Department of Computer Science and Engineering

Master's Thesis 2022
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Gothenburg, Sweden 2022

Optimizing the generation of Java JMH Benchmarks Anthony Path
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

The evolution of the software system during its development is a complex process, which is both very important and difficult to track. One of the methods that offers such functionality is microbenchmarking, which is a type of regression testing. Although its efficient at software project performance measurement and tracking, its also rather difficult to conduct and therefore rarely used in industry. In this study, several potential optimization approaches are applied to the benchmarking process in open-source projects in order to make it less complex and applicable to real-world scenarios, improving its accessibility for software engineering researchers and practitioners.

Keywords: Computer science, engineering, thesis, benchmarks, Java, JMH, optimization, regression testing.

Acknowledgements

The author would like to express their gratitude to the thesis supervisor Philipp Leitner for the support and feedback during the study as well as examiner Regina Hebig for feedback and advice on the general process.

Anthony Path, Gothenburg, June 2022

Contents

List of Figures

List of Tables

1
Introduction

The evolution of software system during its development, modification or maintenance can significantly impact its quality and performance. Although the changes are often made in order to bring new features and improvements to the system, they also have a potential to introduce unwanted defects, ranging from minor bugs to significant, fatal errors [1]. In order to avoid such undesirable consequences of software system changes, numerous approaches can be used.

Regression testing is one of the such approaches. It encompasses techniques that can be applied during the evolution of the software in order to detect potential defects in an efficient and effective way [2]. One of the ways to conduct the regression testing involves microbenchmarking, which can be used to implement a performance baseline for small fragments of code (thus it can be viewed as an alternative to load tests). Even though it is not a widely used testing method, it has noticeable advantages over traditional testing approaches (such as load testing) such as being faster to implement and execute than the latter and not needing a special execution environment [3].

One of the recently developed microbenchmarking methods is junit2jmh, which involves generation of microbenchmarks for JMH (Java Benchmark Harness) using existing unit tests produced by JUnit [4]. Although this method was proven to be reasonably efficient and promising, it's still at an early development stage, being unoptimized for different real-world scenarios such as large projects, limited run time etc. This may lead to very long execution times of generated benchmark suites (up to several days), severely limiting the tool's effectiveness[4]. Similar problem can be found in many Java projects that have their own large benchmark suites.

In this study, potential ways to optimize the generation of microbenchmark suites with regards to execution time, stability of resulting benchmarks, and code coverage were designed and tested in order to find the most efficient approach for the improvement of this regression testing technique. This involved both simple heuristics and more complex approaches. The suites generated by junit2jmh were used as an example, as they can be made from unit tests, greatly increasing the number of projects that can be assessed (test suites are much more common than benchmark suites). Improving the microbenchmarking process can make it much more practical, useful and efficient for application to the real-world Java projects, thus allowing great improvement of their quality.

1.1 Purpose of the study

The core purpose of this study is to provide an efficient approach for optimization of the microbenchmark suites. In the context of this study, optimization means selecting a subset of the existing suite that provides the best trade-off between execution time, stability and code coverage. In order to achieve this, potentially useful optimization approaches were applied to the suites generated by ju2jmh, which therefore served as a demonstration of new improvement methods' capabilities.

The wider purpose of the study is to contribute to and to improve regression testing field by providing better-performing and optimized microbenchmarking methods. The existing advantages of microbenchmarking over more widely used performance testing methods (e.g. load testing which can be slower to implement and run [3]) can potentially be amplified further via improvements outlined in this study. This can in turn lead to more efficient software testing options for both researchers and practitioners, and make benchmarking applicable to more projects, even the ones that don't have benchmark suites (only tests).

1.2 Research questions

The current method of the operation of ju2jmh involves using all the unit tests indiscriminately, which leads to long running times needed to execute the microbenchmark suite. Optimizing it via one or more approaches described below can potentially allow to not only greatly improve suite's efficiency, but also contribute to the regression testing research area by providing useful knowledge about benchmarking optimization approaches.

- **RQ 1** *How efficient is the usage of simple heuristics for the optimization of the suite?*

  Simple heuristics include potential improvement strategies that do not require complex algorithms or measurements. Examples of such approaches may include randomly selecting of a certain number of unit tests from several random packages, selecting only unit tests that have an execution time in certain interval (e.g. between 1ms and 1s) etc.

- **RQ 2** *How efficient is the optimization of the suite based on benchmark stability?*

  Stability of the benchmark is defined as a degree of variability of the results of its multiple runs [3]. It is a very important property of a benchmark, since it determines how large the performance change needs to be before it can be reliably detected. Consequently, prioritising microbenchmarks with a high result stability means that generated benchmarks are more likely to be useful for finding performance regressions.

- **RQ 3** *How efficient is optimisation of the suite based on static code coverage?*

  One of the conclusions that Laaber, Gall & Leitner [2] reached in their study was that static code coverage might be a viable alternative when the running time of microbenchmark suite is limited. Since optimization involves a reduction (and potentially, limiting) of suite running time, prioritizing static code coverage can improve the performance of the the suite. Additionally, static code coverage potentially allows to identify redundant benchmarks (that test the same code fragments), thus leading to further improvements of the generated benchmark suite.

2

Theory

In this chapter, some of the important academic works related to this study are presented. This includes both more widely known concepts such as regression testing and the less known ones such as benchmark testing and conversion of unit tests to benchmarks in Java.

2.1 Performance testing

Molyneaux [5] described importance of performance testing for the software development life-cycle and problems arising from its frequent neglecting by the business organizations (caused by lesser popularity of performance testing compared to operational acceptance testing and functional testing). To be precise, deployment of applications without sufficient information about their performance can potentially lead to serious problems with their scalability and operation soon after mentioned applications are released. This may in turn lead to a loss of time and money spent on performance improvements and also lead to negative impact on testers, coders, architects and other members of the development team. On the other hand, effective usage of performance testing during the development allows to identify possible operational bottlenecks early and quickly, at the stage when its still easy to mitigate them and deploy the resulting highly performing product with confidence. Lack of acceptance of performance testing by the IT practitioners outlined in Molyneaux's work signifies need for further research and work in this area, which this study is going to provide.

Stefan et al. [6] investigated the occurrences of the performance evaluation code in Java-based performance testing frameworks (found on GitHub), identifying their relevant approaches to performance testing as well as quantifying their use. The obtained data was then applied to adjust the SPL (Stochastic Performance Logic) performance testing framework, which was developed and modified by the study authors. The study has also concluded that only a small fraction of analyzed GitHub projects (less than 1%) used performance testing frameworks. The simpler Java function clock query method was used by only 3.4% of projects. Additionally, difficulties were discovered with the automation of the performance testing as well as test execution time. This study was interesting for the research, because it clearly shows lack of dedicated performance testing frameworks in most of the projects in the industry as well as signifying that test execution time is an issue in those frameworks, which can be solved with suite optimization.

Chittimalli [7] investigated application of regression testing during software development. Its expensive costs, which can even reach up to half of software maintenance costs were identified as a problem with this method. One of the possible solutions involves using the same test suite to test both current and future versions of the program. Since it may be too time consuming or expensive to re-run all the tests of such suite every time when regression testing is needed, several methods have been developed to optimize this process. Many such approaches use static code coverage obtained in previous testing stage to select the test cases for the next stage. This supports the notion behind the RQ 3 (optimization of benchmark suite basing on static code coverage), as this approach was proven to be efficient for regression testing before. However, it has not been applied to benchmarks yet. This definitely leads to the need for further research on its efficiency in such situation, which is one of aims of this study.

2.2 Benchmark testing

The Java Microbenchmark Harness (JMH) [8] software was used for running benchmarks during the experiment. Its a Java-based harness for build, run and analysis of differently-sized benchmarks implemented in Java as well as other programming languages that have the JVM as a target. In order to enable benchmarking in the project, the JMH dependency has to be added and the benchmarking classes with correct method annotations created. After that, the benchmarks have to be compiled and built into the .jar file, which can then be run with JVM in order to benchmark the analyzed code. The JMH has multiple possible parameters influencing its run of the benchmark suite, such as number of warmup and measurement iterations, forks, measurement iteration time etc. The JMH was instrumental for this study, as it was used to test and analyze benchmarking optimization approaches.

Laaber et al. [9] stated that execution of software benchmarks is a rather costly process in terms of time. This operation can potentially take many hours or days to execute, which makes frequent software assessment via this method (e.g. for continuous integration) impractical. Furthermore, attempting to change the configuration of a project's benchmark suite without considering the possible influence of said changes on the quality of obtained results can compromise the latter, severely reducing efficiency of the benchmarking for project performance assessment. The two serious issues raised by Laaber et al. [9] could potentially be solved via a novel benchmark optimization approach used in this study, severely improving the regression testing and making it more accessible for wider applications.

Cordeiro et al. [10] stated that complexity of software in enterprise applications and smartphones has greatly increased in recent times. For example, the popular Android OS (whose mobile applications reached almost 87% market share) is composed of a large library set (around 13 million lines of code). Those libraries contain both native code and Java code, which requires tools to verify its security attributes. Additionally, the Java alone is popular in business applications (mainly server-side programming), thanks to the existence of several high-performance frameworks for it (such as Spring). As such, verification methods for Java enterprise applications are also in demand. For example, large technology companies such as Amazon and Facebook invested substantial amounts of time and effort into development of effective and efficient verification methods for testing in order to assess reliability of some of their systems' aspects to improve their security and robustness. Cordeiro et al. [10] also noted that while there are many Java software verification tools (e.g. SPF, Bandera, JayHorn, JPF etc.), they may be rather difficult to compare in a practical setting because of a lack of the methods to reproduce and standardize the empirical evaluations as well as lack of common benchmark sets for them. The latter fact means that developing an easily usable and robust method of benchmark suite optimization (the aim of this study) can potentially help with assessment and comparison of Java-based software verification tools, allowing both practitioners and researchers of Java to greatly improve the process of Java code assessment. This may in turn lead to significant improvements of Java software in terms of quality and reliability, which will be very beneficial for IT industry where Java has historically been one of the most popular programming languages (Bissyandé et al. [11]).

Laaber et al. [2] investigated the application of coverage-based TCP (test case prioritization) techniques to software microbenchmarks, with 54 unique parameter instantiations. The study has concluded that TCP effectiveness had great variation depending on parameterization. However, it could still provide a noticeable overhead of 11% of the total execution time of microbenchmark suite. Additionally, the study results have demonstrated that static-coverage techniques were usually more efficient than dynamic-coverage ones because of their unacceptable analysis overhead. Thus, in cases with limited prioritization time, they could offer a viable alternative. Finally, it was concluded that the total strategy (ranks benchmarks by their total coverage) displayed better performance than the additional strategy (ranks benchmarks by coverage which was not covered by other, already ranked benchmarks). The results of this study imply that static coverage technique can be useful for optimization (which may include limited time to run and thus prioritize microbenchmarks) and also provide other potential ways in which the benchmarking suites can be improved.

Khatchadourian et al.[12] used JMH in their study for assessment of the performance impact of the novel automated refactoring method for the optimization of Java 8 stream code. They stated that using a test harness such as JMH is important for isolation and assessment of performance change caused by the changes of the code. Therefore, benchmark suites provide Java projects with very efficient performance improvement indicators. Additionally, during the study, some of the assessed projects did not include JMH benchmarks but contained large JUnit test suites. Some of the unit tests from such projects were converted into JMH benchmarks via replacement of @Test annotation with @Benchmark annotation, effectively turning the methods into JMH performance tests, which were then successfully used to assess projects' performance changes. This study shows that conversion of unit tests to benchmarks is an efficient way to assess performance of the projects that don't have benchmark suites (or have very small number of benchmarks), which corroborates the usage of ju2jmh unit test-to-benchmark conversion tool in this study. Another point raised in research work by Khatchadourian et al. is the importance of JMH suites for isolation and assessment of software performance changes. This means that improvements in benchmarking operation made possible by this study can potentially greatly contribute to quality and reliability of Java-based software (by providing Java developers with efficient performance tracking methods).

Laaber and Leitner [3] studied the quality of software microbenchmark suites, focusing on their CI integration as well as suitability to provide rapid performance feedback. Ten open-source libraries implemented in Java and Go were investigated with variable size of benchmark suites and duration runtimes. During the study, it was discovered that sometimes benchmarks display result variability equal or higher than 50%, signifying that some of them may not be efficient for discovery of slowdowns. Additionally, artificial slowdowns were introduced in assessed open-source projects to determine if the benchmark suites will detect them. A new metric for performance-test quality was introduced, named ABS (API benchmarking score) for more efficient benchmark suite assessment. In summary, the methodology developed in this study can be used to generate or suggest benchmarks for untested API parts, assess quality of microbenchmark suites as well as select a test set for an efficient continuous integration procedure. The ABS was of particular interest for this research, since it can be used to assess the efficiency of the optimization methods applied to microbenchmarking more efficiently.

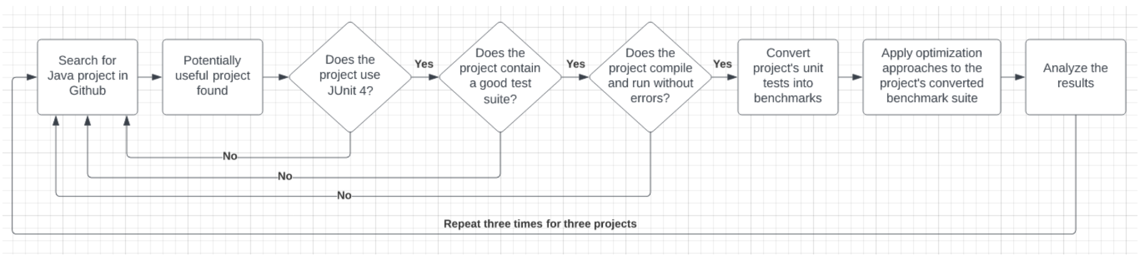
2.3 Benchmark testing with junit2jmh

The junit2jmh is a relatively new benchmarking approach, which uses existing JUnit 4 unit tests of the project in order to generate benchmark suites exercising the same functionality as the tests and enable continuous performance assessment [4]. It was developed to address the rarity of benchmark suites in Java projects and enable developers to conduct easier benchmarking without spending time on writing the benchmarks manually. The main idea behind usage of existing tests for benchmark generation was that primary objects of regression testing (continuous integration projects) are very likely to have a test suite already. This makes usage of benchmarking in such projects much more practical and less time-consuming.

3 Methods

The main scientific method used in this project was an experiment [13]. It was based on open-source projects and had the following variables. Dependent variables were the quality of the generated benchmark suite (in terms of code coverage and stability of the resulting benchmarks) and its execution time. Controlled variable was represented by the execution conditions - all tests were performed in a public/private cloud environment in order to ensure stable and reliable performance that would not impact study results in a negative way. Finally, the independent variable was the type of the optimization approach - the strategy/method used to improve performance of the analyzed microbenchmarking method.

3.1 Experiment setup



Search for Java project in Github
Potentially useful project found
Does the project use JUnit 4?
Yes
Does the project contain a good test suite?
Yes
Does the project compile and run without errors?
Yes
Convert project's unit tests into benchmarks
Apply optimization approaches to the project's converted benchmark suite
Analyze the results
No
No
No
Repeat three times for three projects

Figure 3.1: Experiment setup diagram

The experiment was conducted as follows (Fig. 3.1). Firstly, the source code of several different open-source projects was obtained to be used in the study. The main criteria for the project selection was presence of good test suites (that contain several hundreds to thousands test cases and have as few failures as possible) in the project that can be converted into the benchmarks and used for the analysis. Also, the project had to compile and run without errors and also use JUnit 4 (so that junit2jmh will be able to convert its test cases into benchmarks). Additionally, the virtual testing environment was set up and tested. After that, the optimization testing tool was implemented and tested. The optimization approaches were then applied to the benchmark suites generated by junit2jmh (sixty runs in total for each approach, explained in detail in section 3.1.4.1) and results of this application were analyzed. The process was repeated three times with three different Java projects.

Name	Description	Version	Test suite size
RxJava	Java VM-based Reactive Extensions implementation	3.0.0	12285
Mockito	Mocking framework for unit tests implemented in Java	4.5.1	2067
Stubby4j	HTTP/1.1, HTTP/2 & WebSockets stub server	7.5.2	446

Table 3.1: Study objects

3.1.1 Study objects

Study objects included several open-source Java projects with test suites that could be converted into benchmarks with junit2jmh (Table 3.1).

3.1.1.1 RxJava

The first project chosen for the assessment was RxJava [14], version 3.0.0. It's a Java VM-based Reactive Extensions implementation, which can be used for composing event-based and asynchronous programs, using observable sequences. The main reason why this project was chosen is its extensive test suite of over 12000 cases, which provides lots of material for testing of optimization approaches. Additionally, the project contained its own benchmark suite and had necessary configurations and settings for running both tests and benchmarks, making conduction of the experiment easier.

3.1.1.2 Mockito

The second experimental objects was Mockito [15], version 4.5.1. It's a popular mocking framework for unit tests implemented in Java. The Mockito framework was chosen for the experiment because of its rather large test suite of over 2000 cases which could be converted into a correspondingly large benchmark suite.

3.1.1.3 Stubby4j

The last project chosen for the optimization testing was Stubby4j [16], version 7.5.2. It's a HTTP/1.1, HTTP/2 and WebSockets stub server that can be used for stubbing (introducing components that simply return certain result for testing purposes) of distributed web services in non-containerized environments and docker for contract and integration testing. It had smaller test suite compared to the other study objects (448 cases), and was chosen in order to study if patterns in optimization approach efficiency remain similar for both larger and smaller project sizes.

3.1.2 Evaluated scenarios

In order to ensure high accuracy of the research and maintain consistency, each optimization method would be run under the same conditions (sixty benchmarks per suite, twenty measurement iterations, ten warmup iterations, two forks, one second warmup and iteration measurement time). The parameters were chosen as they both enable reasonably fast benchmark running time (around forty seconds) and allow to collect enough data from the benchmark run for the analysis. In addition, randomly selected sixty benchmarks were run as a baseline for further comparison of the optimization approaches. The optimization scenarios themselves were created according to the research questions. There were four basic scenarios: prioritization of most stable benchmarks, prioritization of benchmarks whose "parent" tests running times were in a certain interval (included three sub-scenarios with intervals which contained larger number of tests compared to others), selection of number of random tests from a number of random packages and prioritization of benchmarks with the highest static code coverage (included two sub-scenarios with instruction and branch coverage).

There were also other potential optimization approaches (such as prioritization of benchmarks by dynamic code coverage, selection of benchmarks with certain score confidence interval etc.), but they were not used in the experiment for a variety of reasons. For example, measurement of dynamic code coverage is rather complex in terms of implementation, as it's needed to both run the program and then analyze its behavior using collected runtime information such as file accesses, network activities, system call traces, and system logs, which requires substantially more resources than static code coverage [17]. As another example, selection of benchmarks basing on the run results of the entire unoptimized suite (e.g. via score confidence interval) was impractical because of very long execution time of said suite (up to nineteen days with chosen experimental JMH settings, depending on project).

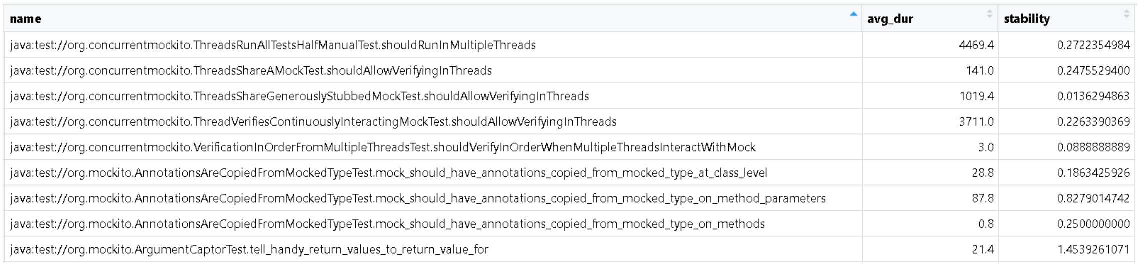
3.1.2.1 Baseline

The baseline to be used in optimization method comparison was the least complex approach - a completely random selection of sixty benchmarks from the entire suite. This was accomplished via the Optimizer software (described below), in two stages. The first stage involved search for benchmark methods in all classes, assignment of an index to each of them and determination of their number in the project. The second stage involved generation of sixty random integers between zero and a number of benchmarks in the suite and selection of benchmarks with indexes corresponding to those integers. As a result, sixty completely random benchmarks were selected and others were ignored during the JMH run. The performance of the suites generated in this way was used as a baseline threshold for the other approaches (performing better than baseline shows efficiency of the approach when it comes to a particular metric).

3.1.2.2 Stability prioritization

For prioritization by stability, the test suite of the analyzed project was run five times with IntelliJ [18], producing .xml test results with each test case's run times. As according to Alexandersson [4], stability of benchmark made with junit2jmh corresponds to the stability of the unit test it was converted from, the data from .xml files was used to calculate standard deviation of time of five test runs (Fig.

3.2). It was in turn used for the benchmark stability prioritization approach (RQ 2), as the lower the standard deviation is, the more stable and constant the test's running time is over repeated runs. Stability prioritization may potentially lead to successful optimization in terms of the benchmark stability.



name	avg_dur	stability
java:test://org.concurrentmockito.ThreadsRunAllTestsHalfManualTest.shouldRunInMultipleThreads	4469.4	0.2722354984
java:test://org.concurrentmockito.ThreadsShareAMockTest.shouldAllowVerifyingInThreads	141.0	0.2475529400
java:test://org.concurrentmockito.ThreadsShareGenerouslyStubbedMockTest.shouldAllowVerifyingInThreads	1019.4	0.0136294863
java:test://org.concurrentmockito.ThreadVerifiesContinuouslyInteractingMockTest.shouldAllowVerifyingInThreads	3711.0	0.2263390369
java:test://org.concurrentmockito.VerificationInOrderFromMultipleThreadsTest.shouldVerifyInOrderWhenMultipleThreadsInteractWithMock	3.0	0.0888888889
java:test://org.mockito.AnnotationsAreCopiedFromMockedTypeTest.mock_should_have_annotations_copied_from_mocked_type_at_class_level	28.8	0.1863425926
java:test://org.mockito.AnnotationsAreCopiedFromMockedTypeTest.mock_should_have_annotations_copied_from_mocked_type_on_method_parameters	87.8	0.8279014742
java:test://org.mockito.AnnotationsAreCopiedFromMockedTypeTest.mock_should_have_annotations_copied_from_mocked_type_on_methods	0.8	0.2500000000
java:test://org.mockito.ArgumentCaptorTest.tell_handy_return_values_to_return_value_for	21.4	1.4539261071

Figure 3.2: Example of test case stability and mean run time data

3.1.2.3 Running time interval heuristic

For the running time interval prioritization heuristic, a similar approach was used as during Stability prioritization. Namely, the results of five runs of project's test suite in IntelliJ (.xml files) were analyzed (Fig. 3.2). The mean running time of each test case's five runs' results was used for time interval heuristic (RQ 1), with random sixty benchmarks being selected from a set with given mean running time interval. Three intervals were selected (0 to 5ms, 5 to 10ms and 10 to 15ms) for RxJava and Mockito, as most of those projects' test cases had run time in this interval (Fig. 3.3), meaning larger sample and more validity for such interval pick. The situation was different for Stubby4j, which had a comparably small test suite and thus possessed less than sixty benchmarks for all time intervals above 10ms. Two intervals were selected for it (0-5ms and 5-10ms). Prioritization of benchmarks whose "parent" unit tests had such short running times may potentially lead to shorter running time for the entire suite.

3.1.2.4 Random package heuristic

The random package selection heuristic was implemented in a similar way to the baseline, but with three stages instead of two. In first stage, the optimizer software would search for unique packages in the analyzed project and assign the index to each one. In the second stage, a number (determined by the user) of random integers between zero and unique package count was generated and packages with those indexes were selected. In the third stage, a number (again determined by the user) of benchmarks was randomly selected from each package in a way described in "Baseline" subsection. This led to a subset of the main benchmark suite with two randomization stages. Such randomization is more precise and more evenly distributed than simple selection of the benchmarks from the entire suite and thus may offer improved results over the baseline (completely random selection).

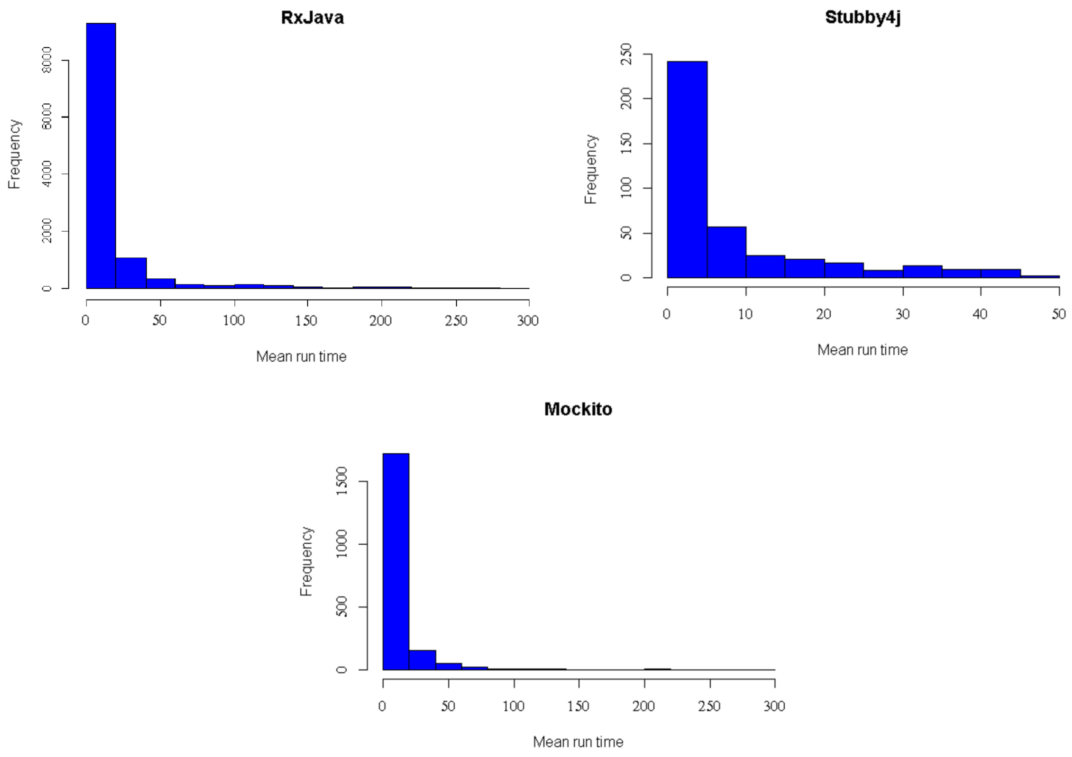


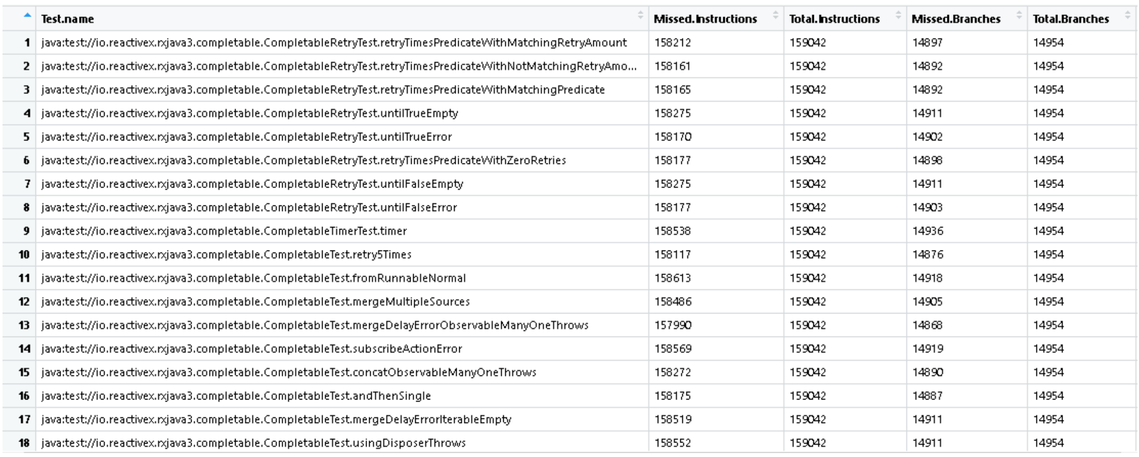
Figure 3.3: Frequency of test cases in different time intervals in test subjects

3.1.2.5 Coverage prioritization

The prioritization by static code coverage was the most complex, as no working standalone tools were found that could calculate each individual test case's code coverage and provide this data in an analyzable report. Instead, the new functionality was implemented for Benchmark Optimizer software, which ran each test case individually with Jacoco plugin and parsed the static code coverage data from the run result into the .csv file (fig 3.4). This resulted in an easily analyzable table with branch and instruction coverage values for each test. The test cases were then sorted by the coverage (the less missed instructions or branches, the better) and 60 with the highest coverage were selected for the assessment of RQ 3 with two tested coverage types - branch and instruction. Such prioritization may lead to improved static code coverage of the optimized suite compared to the baseline (with instruction and branch coverage approaches potentially performing better in terms of their namesake metrics).

3.1.2.6 Approach comparison

The results of the optimization using those approaches (in form of .csv files) were analyzed and compared in order to determine which one is the most efficient at this task. Optimization methods were compared both to baseline (randomly selected benchmarks) and to each other in terms of performance. The three metrics used for comparison were static code coverage (in terms of missed instructions and branches),



	Test.name	Missed.Instructions	Total.Instructions	Missed.Branches	Total.Branches
1	java:test://io.reactivex.rxjava3.completable.CompletableRetryTest.retryTimesPredicateWithMatchingRetryAmount	158212	159042	14897	14954
2	java:test://io.reactivex.rxjava3.completable.CompletableRetryTest.retryTimesPredicateWithNotMatchingRetryAmo...	158161	159042	14892	14954
3	java:test://io.reactivex.rxjava3.completable.CompletableRetryTest.retryTimesPredicateWithMatchingPredicate	158165	159042	14892	14954
4	java:test://io.reactivex.rxjava3.completable.CompletableRetryTest.untilTrueEmpty	158275	159042	14911	14954
5	java:test://io.reactivex.rxjava3.completable.CompletableRetryTest.untilTrueError	158170	159042	14902	14954
6	java:test://io.reactivex.rxjava3.completable.CompletableRetryTest.retryTimesPredicateWithZeroRetries	158177	159042	14898	14954
7	java:test://io.reactivex.rxjava3.completable.CompletableRetryTest.untilFalseEmpty	158275	159042	14911	14954
8	java:test://io.reactivex.rxjava3.completable.CompletableRetryTest.untilFalseError	158177	159042	14903	14954
9	java:test://io.reactivex.rxjava3.completable.CompletableTimerTest.timer	158538	159042	14936	14954
10	java:test://io.reactivex.rxjava3.completable.CompletableTest.retry5Times	158117	159042	14876	14954
11	java:test://io.reactivex.rxjava3.completable.CompletableTest.fromRunnableNormal	158613	159042	14918	14954
12	java:test://io.reactivex.rxjava3.completable.CompletableTest.mergeMultipleSources	158486	159042	14905	14954
13	java:test://io.reactivex.rxjava3.completable.CompletableTest.mergeDelayErrorObservableManyOneThrows	157990	159042	14868	14954
14	java:test://io.reactivex.rxjava3.completable.CompletableTest.subscribeActionError	158569	159042	14919	14954
15	java:test://io.reactivex.rxjava3.completable.CompletableTest.concatObservableManyOneThrows	158272	159042	14890	14954
16	java:test://io.reactivex.rxjava3.completable.CompletableTest.andThenSingle	158175	159042	14887	14954
17	java:test://io.reactivex.rxjava3.completable.CompletableTest.mergeDelayErrorIterableEmpty	158519	159042	14911	14954
18	java:test://io.reactivex.rxjava3.completable.CompletableTest.usingDisposerThrows	158552	159042	14911	14954

Figure 3.4: Fragment of an individual test case coverage file

benchmark stability and running time of the benchmark suite. The RStudio software was used for this task, because of its efficiency in analyzing vast amounts of experimental data.

3.1.3 Execution environment

The execution environment was composed of two identical cloud-based virtual machines, each running an Ubuntu 20.04 OS and having 16 GB of RAM and 40 GB of storage. The machines with those parameters were chosen because earlier test attempts of experimental procedure on VMs with less RAM failed as a result of insufficient memory. Each machine had copies of cloned GitHub repositories of analyzed open-source projects, a .jar file of compiled optimization tool and all necessary software installed for experimental operation (e.g. gradle, nohup).

3.1.4 Used tools

Several tools were used during the study, both for direct application of the optimization approaches to study objects and for other activities necessary for the experiment, such as test running time measurement and compilation of study subjects' source code. These tools are described below.

3.1.4.1 Optimization tool

In order to test the optimization methods, a BenchmarkOptimizer software was developed in Java. It consists of five classes that provide different functions for automation of the testing process. (3.5).

The Main class contained the main method used to run the program as well as methods for recursive search of the analyzed project's directory for files, preparation of log files and debug diagnostics. It also uses three other classes to apply different optimization strategies (chosen by the user).

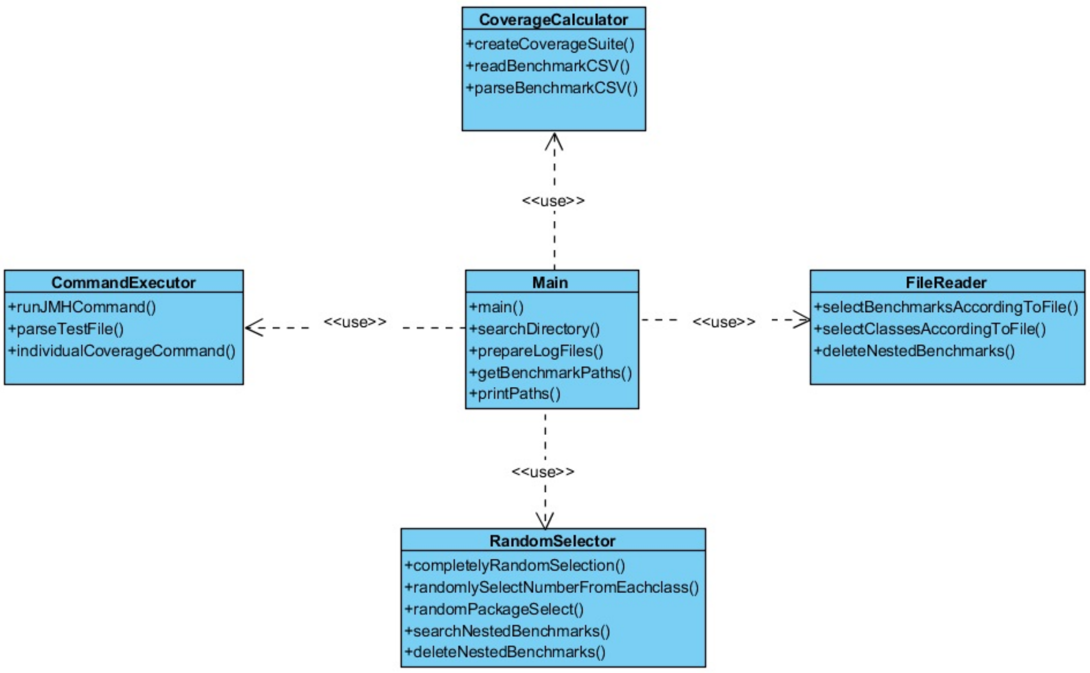


Figure 3.5: Benchmark optimizer class diagram

The FileReader class allowed to select benchmarks and classes containing them according to the text file provided by the user. Since benchmarks' parameters such as coverage and stability were calculated outside of the program, this class allowed to test suites created by such methods. It was used to investigate RQ 2, 3 as well as an prioritization of benchmarks according to their execution time (part of RQ 1).

The RandomSelector class was used to test the random package selection heuristic and to produce a baseline (randomly selected benchmark suite) for comparison to other approaches.

The CommandExecutor class provided functionalities for building the optimized JMH benchmark suite into the JAR file, its subsequent execution according to the parameters defined by the user and parsing of the JSON files created by JMH runs into CSV format, which can then be used for data analysis. It also contained method for analysis of individual test cases' static code coverage (for RQ 3).

The CoverageCalculator class enabled assessment of the optimized benchmark suites' static code coverage (which is one of the metrics for optimization approaches' efficiency comparison).

The entire tool works as a fully-automatic pipeline (example shown in fig 3.6). Firstly, the user has to start it with necessary arguments (e.g. benchmark directory, test type, number of iterations etc.). Secondly, the tool cleans the old log files (to avoid process slowdown caused by lack of memory) and conducts optimization approach testing loops/cycles. There are two types of loops: outer (which involve

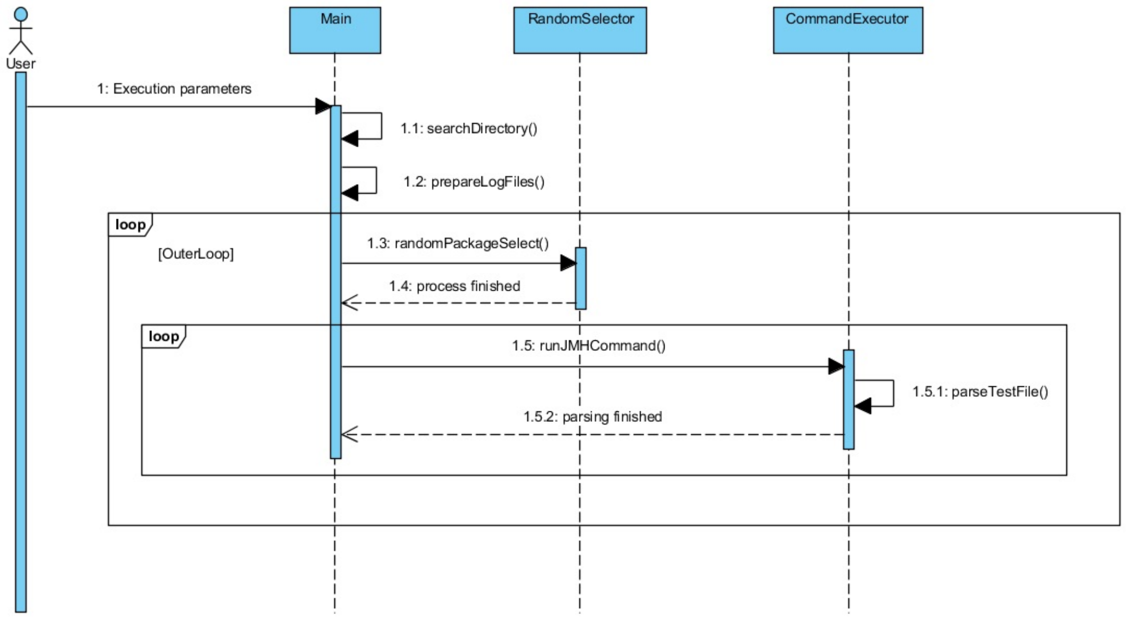
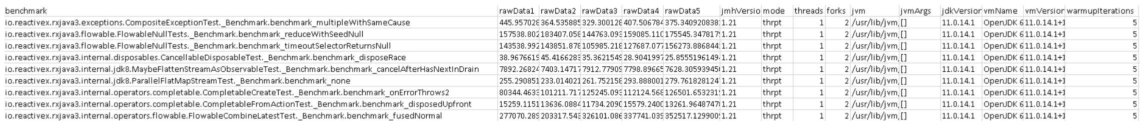


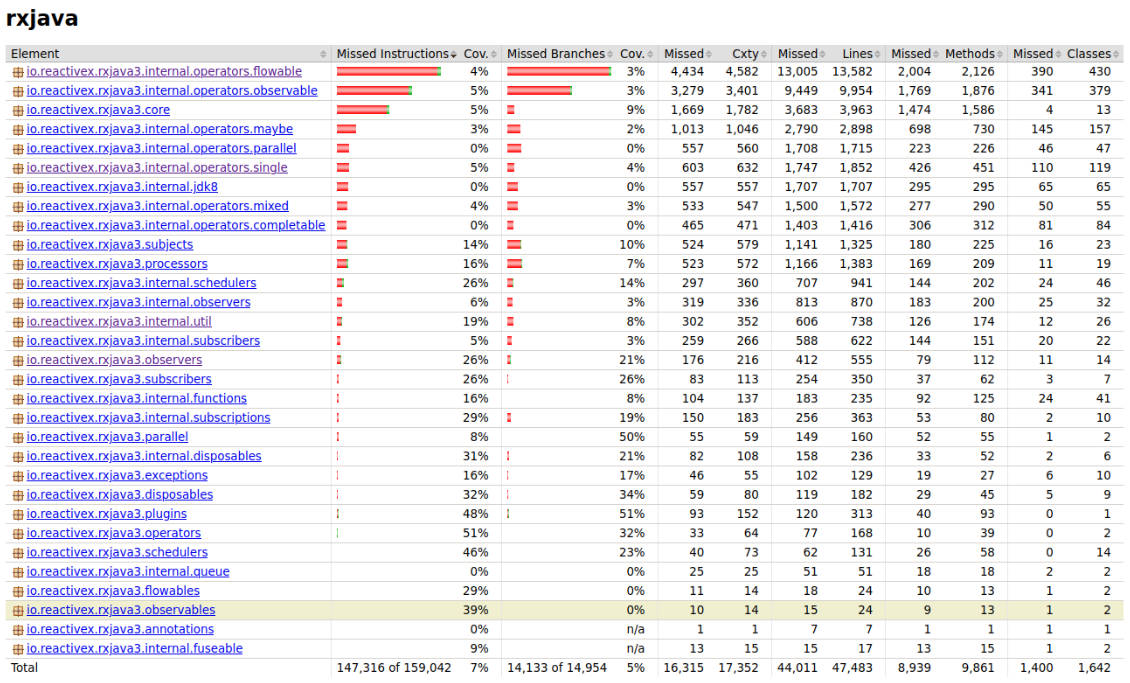
Figure 3.6: Benchmark optimizer system sequence diagram



benchmark	rawData1	rawData2	rawData3	rawData4	rawData5	jmhVersio	mode	threads	forks	jvm	jvmArgs	jdkVersior	vmName	vmVersior	warmupIterations
io.reactivex.rxjava3.exceptions.CompositeExceptionTest._Benchmark.benchmark_multipleWithSameCause	445.957028	364.535885	329.300128	407.506784	375.340920838	1.21	thrpt	1	2	/usr/lib/jvm,	[]	11.0.14.1	OpenJDK 6	11.0.14.1+1	5
io.reactivex.rxjava3.flowable.FlowableNullTests._Benchmark.benchmark_reduceWithSeedNull	157538.802	183407.058	144763.093	159085.110	175545.347817	1.21	thrpt	1	2	/usr/lib/jvm,	[]	11.0.14.1	OpenJDK 6	11.0.14.1+1	5
io.reactivex.rxjava3.flowable.FlowableNullTests._Benchmark.benchmark_timeoutSelectorReturnsNull	143538.992	143851.878	105985.218	127687.077	156273.886844	1.21	thrpt	1	2	/usr/lib/jvm,	[]	11.0.14.1	OpenJDK 6	11.0.14.1+1	5
io.reactivex.rxjava3.internal.disposables.CancellableDisposableTest._Benchmark.benchmark_disposeRace	38.9676619	45.4166283	35.3621549	28.9041997	25.8555196149	1.21	thrpt	1	2	/usr/lib/jvm,	[]	11.0.14.1	OpenJDK 6	11.0.14.1+1	5
io.reactivex.rxjava3.internal.jdk8.MaybeFlattenStreamAsObservableTest._Benchmark.benchmark_cancelAfterHasNextInDrain	7892.26824	7403.14717	7912.77909	7798.89665	7628.30593945	1.21	thrpt	1	2	/usr/lib/jvm,	[]	11.0.14.1	OpenJDK 6	11.0.14.1+1	5
io.reactivex.rxjava3.internal.jdk8.ParallelFlatMapStreamTest._Benchmark.benchmark_none	255.290851	233.014021	261.752158	293.888001	279.761828124	1.21	thrpt	1	2	/usr/lib/jvm,	[]	11.0.14.1	OpenJDK 6	11.0.14.1+1	5
io.reactivex.rxjava3.internal.operators.completable.CompletableCreateTest._Benchmark.benchmark_onErrorThrows2	80344.4633	101211.717	125245.093	112124.568	126501.653231	1.21	thrpt	1	2	/usr/lib/jvm,	[]	11.0.14.1	OpenJDK 6	11.0.14.1+1	5
io.reactivex.rxjava3.internal.operators.completable.CompletableFromActionTest._Benchmark.benchmark_disposedUpfront	15259.1151	13636.0884	11734.2090	15579.2400	13261.9648747	1.21	thrpt	1	2	/usr/lib/jvm,	[]	11.0.14.1	OpenJDK 6	11.0.14.1+1	5
io.reactivex.rxjava3.internal.operators.flowable.FlowableCombineLatestTest._Benchmark.benchmark_fusedNormal	277070.289	203317.543	326101.086	337741.039	352517.129900	1.21	thrpt	1	2	/usr/lib/jvm,	[]	11.0.14.1	OpenJDK 6	11.0.14.1+1	5

Figure 3.7: Fragment of an optimizer's benchmark run output

generation and building of new set of benchmarks via chosen approach) and inner (which involve same set of benchmarks being run several times and take place inside outer loop). Each test run in the study contained thirty outer and two inner loops, leading to sixty iterations for each approach in total. After the end of every inner loop, the data from JMH run's JSON file is parsed (benchmark name, run parameters, score etc, see fig 3.7) into CSV format and added to the output file. This "constant file writing" aspect greatly increases tool's fault tolerance, as even if the program fails at some point, the data it recorded before stopping will still be available. After the last loop was finished, the tool stops automatically. The data from the run could then used to assess the efficiency of the tested approach. As the tool's run may take a long time (around forty two hours) in the experimental setup, a "nohup" tool from Ubuntu was used to make the optimizer run in the background without need of user intervention.
Additionally, the BenchmarkOptimizer was used to calculate code coverage of benchmark suites generated by junit2jmh. The principle behind this operation is similar to the optimization functionality - the coverage algorithm selects a subset of unit tests corresponding to the analyzed benchmark suite (since they cover the same parts of the program) and runs it with the Jacoco plugin [19], producing the coverage report (HTML example shown in fig 3.8). After the run is complete, coverage figures (missed instructions and branches) are extracted from the test report, recorded to .csv file and stored for further analysis.



rxjava

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
io.reactivex.rxjava3.internal.operators.flowable		4%		3%	4,434	4,582	13,005	13,582	2,004	2,126	390	430
io.reactivex.rxjava3.internal.operators.observable		5%		3%	3,279	3,401	9,449	9,954	1,769	1,876	341	379
io.reactivex.rxjava3.core		5%		9%	1,669	1,782	3,683	3,963	1,474	1,586	4	13
io.reactivex.rxjava3.internal.operators.maybe		3%		2%	1,013	1,046	2,790	2,898	698	730	145	157
io.reactivex.rxjava3.internal.operators.parallel		0%		0%	557	560	1,708	1,715	223	226	46	47
io.reactivex.rxjava3.internal.operators.single		5%		4%	603	632	1,747	1,852	426	451	110	119
io.reactivex.rxjava3.internal.jdk8		0%		0%	557	557	1,707	1,707	295	295	65	65
io.reactivex.rxjava3.internal.operators.mixed		4%		3%	533	547	1,500	1,572	277	290	50	55
io.reactivex.rxjava3.internal.operators.completable		0%		0%	465	471	1,403	1,416	306	312	81	84
io.reactivex.rxjava3.subjects		14%		10%	524	579	1,141	1,325	180	225	16	23
io.reactivex.rxjava3.processors		16%		7%	523	572	1,166	1,383	169	209	11	19
io.reactivex.rxjava3.internal.schedulers		26%		14%	297	360	707	941	144	202	24	46
io.reactivex.rxjava3.internal.observers		6%		3%	319	336	813	870	183	200	25	32
io.reactivex.rxjava3.internal.util		19%		8%	302	352	606	738	126	174	12	26
io.reactivex.rxjava3.internal.subscribers		5%		3%	259	266	588	622	144	151	20	22
io.reactivex.rxjava3.observers		26%		21%	176	216	412	555	79	112	11	14
io.reactivex.rxjava3.subscribers		26%		26%	83	113	254	350	37	62	3	7
io.reactivex.rxjava3.internal.functions		16%		8%	104	137	183	235	92	125	24	41
io.reactivex.rxjava3.internal.subscriptions		29%		19%	150	183	256	363	53	80	2	10
io.reactivex.rxjava3.parallel		8%		50%	55	59	149	160	52	55	1	2
io.reactivex.rxjava3.internal.disposables		31%		21%	82	108	158	236	33	52	2	6
io.reactivex.rxjava3.exceptions		16%		17%	46	55	102	129	19	27	6	10
io.reactivex.rxjava3.disposables		32%		34%	59	80	119	182	29	45	5	9
io.reactivex.rxjava3.plugins		48%		51%	93	152	120	313	40	93	0	1
io.reactivex.rxjava3.operators		51%		32%	33	64	77	168	10	39	0	2
io.reactivex.rxjava3.schedulers		46%		23%	40	73	62	131	26	58	0	14
io.reactivex.rxjava3.internal.queue		0%		0%	25	25	51	51	18	18	2	2
io.reactivex.rxjava3.flowables		29%		0%	11	14	18	24	10	13	1	2
io.reactivex.rxjava3.observables		39%		0%	10	14	15	24	9	13	1	2
io.reactivex.rxjava3.annotations		0%		n/a	1	1	7	7	1	1	1	1
io.reactivex.rxjava3.internal.fuseable		9%		n/a	13	15	15	17	13	15	1	2
Total	147,316 of 159,042	7%	14,133 of 14,954	5%	16,315	17,352	44,011	47,483	8,939	9,861	1,400	1,642

Figure 3.8: Example of Jacoco HTML coverage report

The optimizer software was developed in several iterations, with more functionality added on each stage (which was also tested every time in order to ensure high reliability). After the tool was developed, its optimization methods were applied to the benchmark suites generated by junit2jmh. While the random package heuristic (RQ 1) and baseline (random selection) were straightforward, other optimization approaches required more work (described in "Evaluated scenarios" section).

3.1.4.2 IntelliJ

IntelliJ [18] is an integrated development environment (IDE) for Java, developed by JetBrains. It was mainly used in the study for running unit tests in analyzed projects and producing the .xml data of the run results that could be used for prioritization of benchmarks by stability and run time interval.

3.1.4.3 Gradle

Gradle [20] is a tool for build automation that can be used for software development in multiple programming languages. Apart from building the project, it also provides functionalities for testing and publishing of the software. In this study, Gradle was used to build and run the optimized JMH suites during the experiments.

3.1.4.4 RStudio

RStudio [21] is an IDE for R, which can be used for statistical computing, data analysis and creation of analytical graphics (histograms, scatter plots etc.). In the study it was mainly used to analyze and process experimental data in order to visualize it and compare efficiency of different optimization approaches accurately.

3.1.4.5 MS Excel

Microsoft Excel [22] is a spreadsheet software developed by Microsoft that can be used for processing (e.g. calculation, computation) as well as visualization of data. In this study, the Excel was used for processing and subsequent visualization of the experimental run data.

3.1.5 Analysis

After the experiment was conducted, the optimizer tool's .csv file outputs for both benchmark tests and Jacoco coverage reports were used to assess the quality of benchmark suites produced via different optimization approaches. Average running time was calculated for each sixty test runs of every optimization method as well as average static code coverage (in terms of missed branches and instructions) and average stability of each benchmark (standard deviation of its run time). This data was then used to create diagrams (via MS Excel and RStudio) and subsequently to analyze and compare different optimization approaches in terms of efficiency. After the most efficient ones were determined (the ones which produced benchmark suites with the best performance in terms of one or more experimental metrics), the recommendations were also produced on which approaches are to be used in which optimization-requiring situation.

3.1.6 Validity threats

Several factors could potentially compromise the validity of the results of this study, which are described in this section.

3.1.6.1 Internal validity threats

During the experiment, only a limited number of repetitions could be run for each approach because of time constraints. This factor could potentially negatively impact study results (less data, less knowledge about factors that could experimental runs etc.), which could in turn reduce validity of the study results.

Additionally, during the study, the free space of virtual machine used as execution environment was gradually reduced as more test subjects (open-source Java projects) and experimental data accumulated over time. Although no visible issues were detected during the study, such reduction of free storage space could potentially negatively impact the performance of the virtual machine, compromising validity of the study results.

3.1.6.2 External validity threats

Since the experiment was conducted with open-source Java projects only, the generalizability of obtained results may potentially be limited. For example, it's not known if the optimization methods that worked for experimental subjects in this study will still work for industrial, closed source Java projects or if they will be applicable to the software written in another programming language.

As the experiment was conducted in an Ubuntu-based virtual machine with specific properties (16GB RAM 40GB storage), the results may potentially have limited generalizability to other types of execution environments.

3.1.6.3 Construct validity threats

The virtual machine that served as an execution environment was periodically taken offline by the provider for maintenance. While the optimizer tool recorded test results iteratively and the results remained even if shutdown occurred, such events could potentially impact the work of the experimental tools and influence the result validity negatively. This threat was mitigated by running the tool outside of maintenance days.

4
Results

After the experimental runs were conducted, the quantitative data was obtained regarding the different optimization approaches' performance, which is presented in this section.

4.1 Coverage

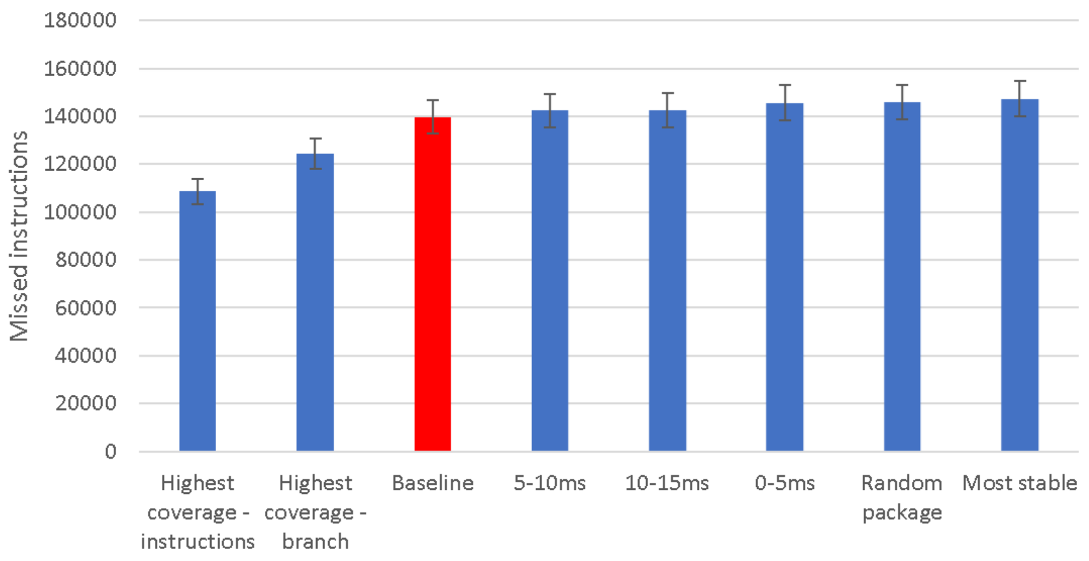


Figure 4.1: RxJava - missed instructions per different approaches

Static code coverage was assessed in terms of average number of missed instructions and branches for all run cycles of each method. Therefore, in the diagrams lower value shows higher coverage (less code was missed).

When it comes to static code coverage, the different optimization approaches demonstrated varying effectiveness when applied to RxJava benchmark suite (results of this assessment are shown in Fig 4.1 and 4.2). Prioritization of benchmarks by coverage (branch and instruction) produced the suites with the highest code coverage (least missed branches and instructions), which was expected, as it was one of the main potential advantages of this approach. All other approaches were less efficient in this regard, having slightly less static code coverage than even the baseline (completely random benchmark selection, highlighted in red on the diagram), however the performance difference between them was negligible. The prioritization of benchmarks

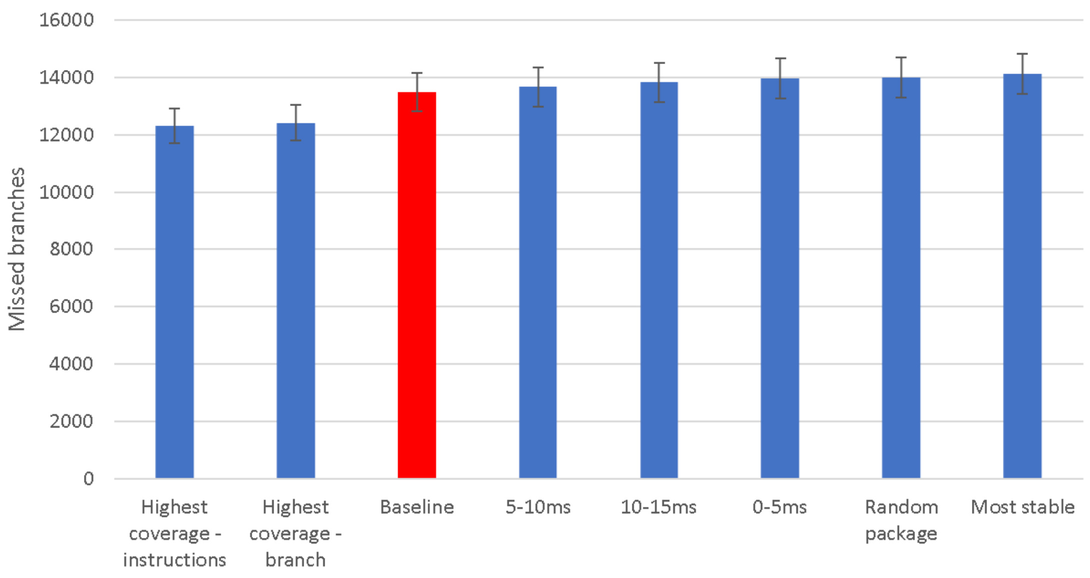


Figure 4.2: RxJava - missed branches per different approaches

whose tests ran in 5-10ms range produced second-best code coverage, while the 10-15ms and 0-5ms prioritization demonstrated slightly weaker results. The random package selection produced suite with second-least coverage, while prioritization of most stable benchmarks displayed the worst results in this regard. The minimal difference between worse-performing approaches could be caused by similar coverage in suites that were not selected basing on this metric. While the highest coverage prioritization approaches demonstrated noticeably higher coverage in terms of missed instructions than the baseline and other methods, the difference was smaller when it comes to branch coverage - around 10-15% between the baseline and the most efficient approaches. All other approaches had even less difference between their results, below 10%.

The optimization methods applied to the benchmark suite generated from Mockito unit tests demonstrated slightly varying coverage in terms of missed branches and instructions (Fig. 4.3 and 4.4). When it comes to instruction coverage, the 5-10ms prioritization produced the best results and was the only approach that performed better than the baseline. Random package selection heuristic, prioritization of most stable benchmarks as well as 10-15ms and 0-5ms interval prioritization produced underperforming results, while highest coverage benchmark prioritization approaches counter-intuitively produced suites with the lowest instruction coverage. When it comes to branch coverage, the results were slightly different. While 0-5ms prioritization, random package selection and highest instruction coverage prioritization approaches demonstrated similar performance as with instruction coverage, other methods produced different results. Most stable benchmark prioritization performed slightly better, while 10-15ms and 0-5ms prioritization were less efficient. Highest branch coverage prioritization also demonstrated slightly better results with this metric. In both branch and instruction coverage, the difference between all approaches was rather small, less than 10%.

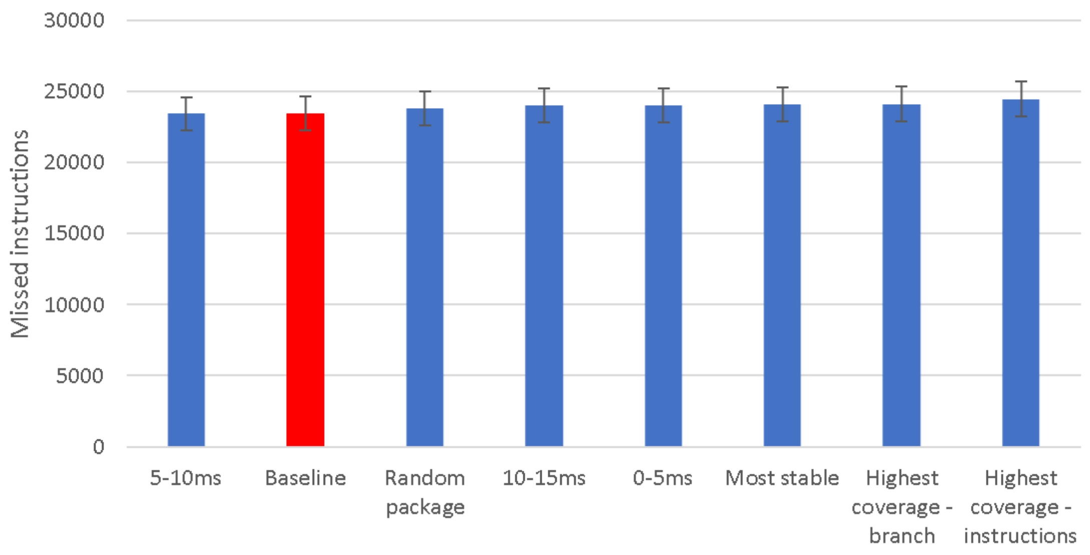


Figure 4.3: Mockito - missed instructions per different approaches

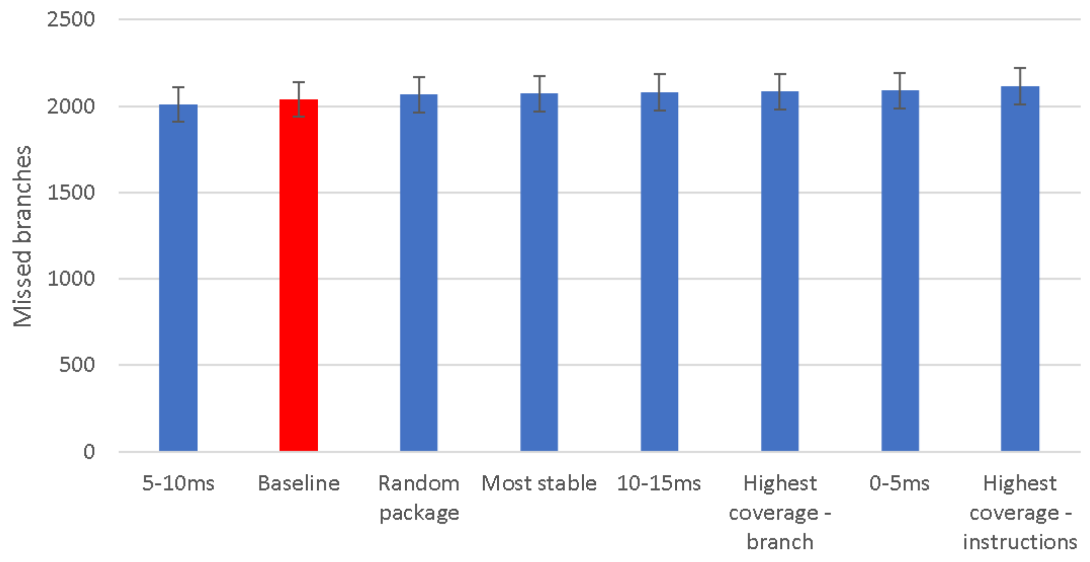


Figure 4.4: Mockito - missed branches per different approaches

When it comes to application of the optimization approaches to Stubby4j, they performed as follows in terms of static code coverage (Fig. 4.5 and 4.6). In a similar way to Mockito, highest coverage prioritization approaches actually produced suites with the lowest coverage, both in terms of branch and instruction coverage. Other optimization approaches were more efficient. In terms of missed instructions, 0-5ms and 5-10ms prioritization produced higher-quality results, but still fell short of most stable prioritization and random package selection (the latter being the only method that surpassed baseline in terms of instruction coverage). When it comes to branch coverage, the results were mostly similar to instruction coverage, but 5-10ms prioritization produced suites with the highest coverage, while random package selection and 0-5ms prioritization produced lower-quality results, with coverage between sta-

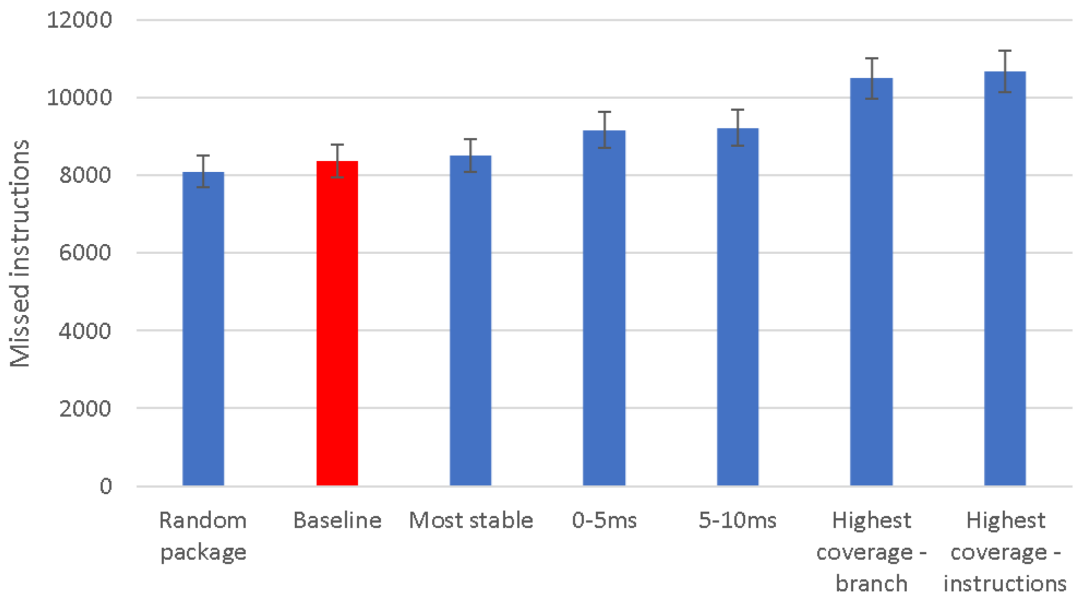


Figure 4.5: Stubby4j - missed instructions per different approaches

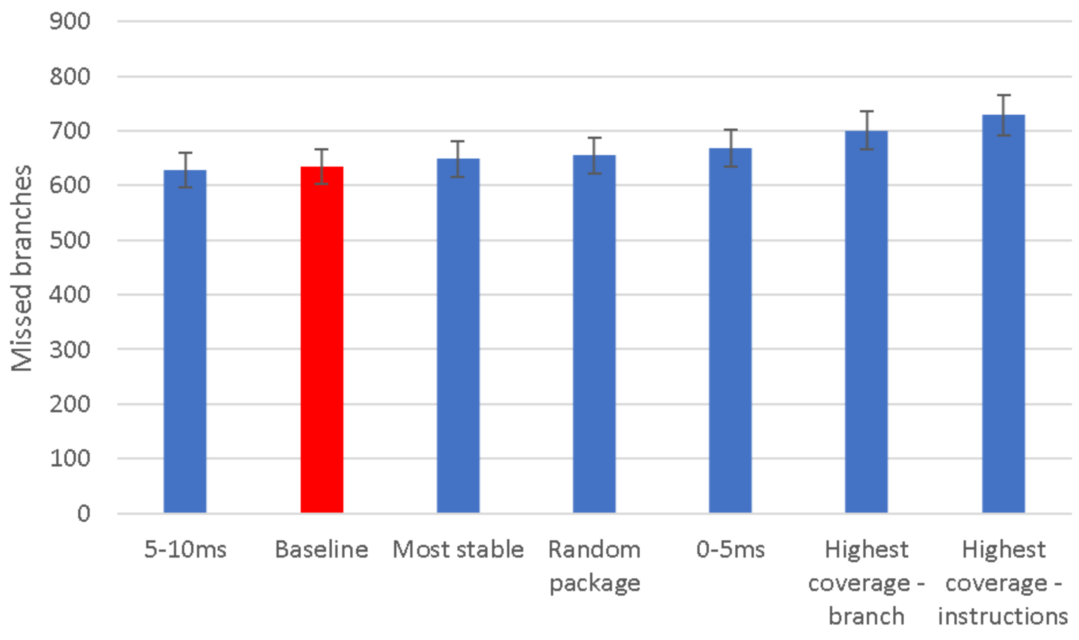


Figure 4.6: Stubby4j - missed branches per different approaches

bility prioritization and coverage prioritization approaches. The difference in coverage between different approaches was noticeable here, sometimes reaching over 20% (e.g. between the baseline and highest instruction coverage prioritization approach). The suboptimal performance of coverage prioritization approaches in terms of suite coverage metric was unexpected, as (as seen from the approaches' names) high performance in this area is supposed to be their main advantage. It could potentially be explained by overlapping of the code covered by multiple benchmarks in the suite (when several of them run and cover the same branches/instructions), thus resulting in less coverage overall.

4.2 Benchmark stability

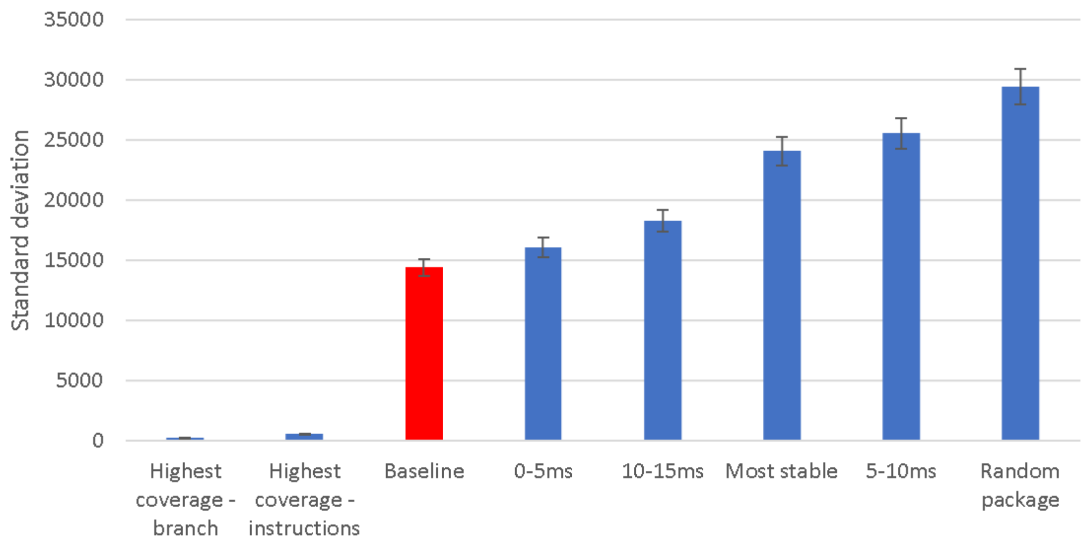


Figure 4.7: RxJava - benchmark stability of different approaches

Individual benchmark stability was calculated as a standard deviation of each benchmark's fifteen run iterations' duration in milliseconds (per experimental setup, Fig 4.7). After this, the average stability of the entire run of the optimization method was calculated and used for the analysis. Since the diagram represents average standard deviation, smaller value shows higher stability. In terms of benchmark stability for RxJava, the highest coverage benchmark prioritization approach showed the best performance by a very wide margin, which was surprising given that prioritization of most stable benchmarks showed noticeably inferior results. All other approaches produced much more unstable benchmark suites, but 10-15ms prioritization showed much more stability than 5-10ms prioritization, in turn being outperformed by 0-5ms prioritization. Finally, the random package heuristic produced the worst suite in terms of stability. All optimization approaches except highest coverage prioritization failed to surpass the baseline when it comes to the benchmark stability metric.

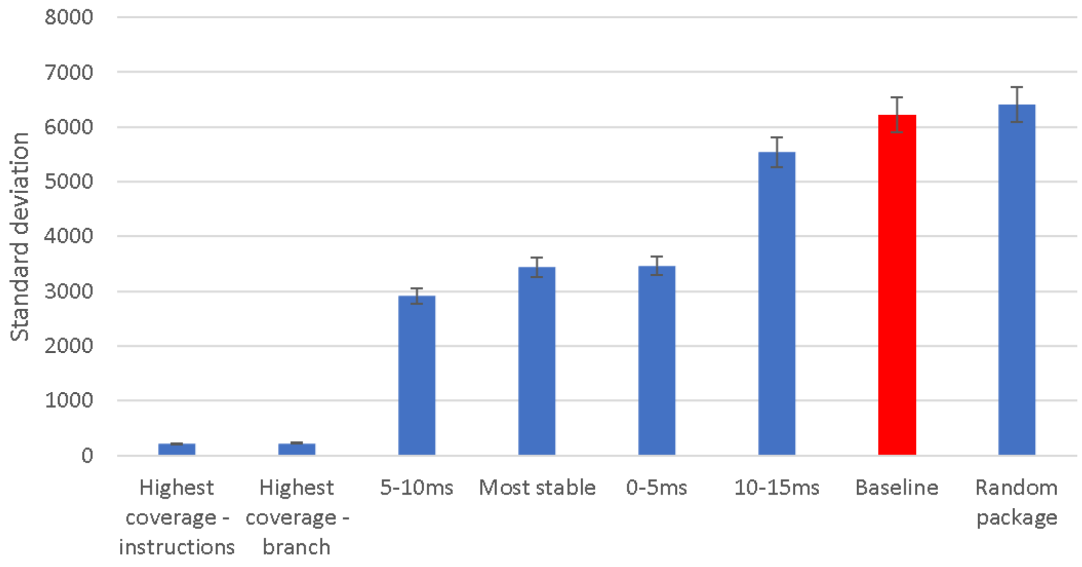


Figure 4.8: Mockito - benchmark stability of different approaches

When it comes to benchmark stability for Mockito suite, optimization approaches performed as follows (Fig. 4.8). Both highest coverage prioritization approaches demonstrated the best benchmark stability, while other methods were considerably less efficient in this regard. 5-10ms and 0-5ms prioritization as well as stability prioritization approaches created much less stable suites than coverage prioritization method. However, they still were more efficient than 10-15ms prioritization and random package selection (which was also the only approach that failed to surpass baseline when it comes to stability metric).

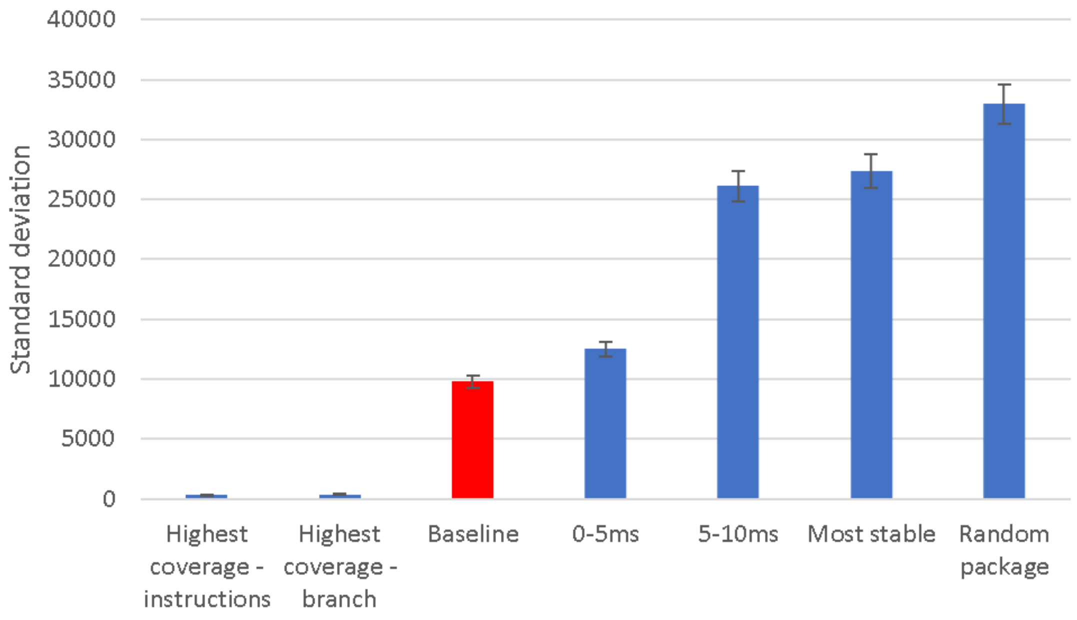


Figure 4.9: Stubby4j - benchmark stability of different approaches

In terms of stability, the optimization approaches applied to Stubby4j demonstrated the following results (Fig. 4.9). Similarly to the other study objects, highest coverage approaches again demonstrated exceptional benchmark stability, greatly outperforming all other optimization methods (which also did not overcome baseline) in terms of this metric. The 0-5ms prioritization had the second-best results in terms of stability (but still very subpar compared to coverage prioritization methods), while remaining approaches demonstrated even more substandard performance, with random package selection heuristic producing the least stable benchmark suites. In a similar way to RxJava, prioritization of the most stable benchmarks produced second least stable suite, which was unexpected and counter-intuitive, as potential advantage of this approach was increased stability of the resulting benchmark suite.

4.3 Running time

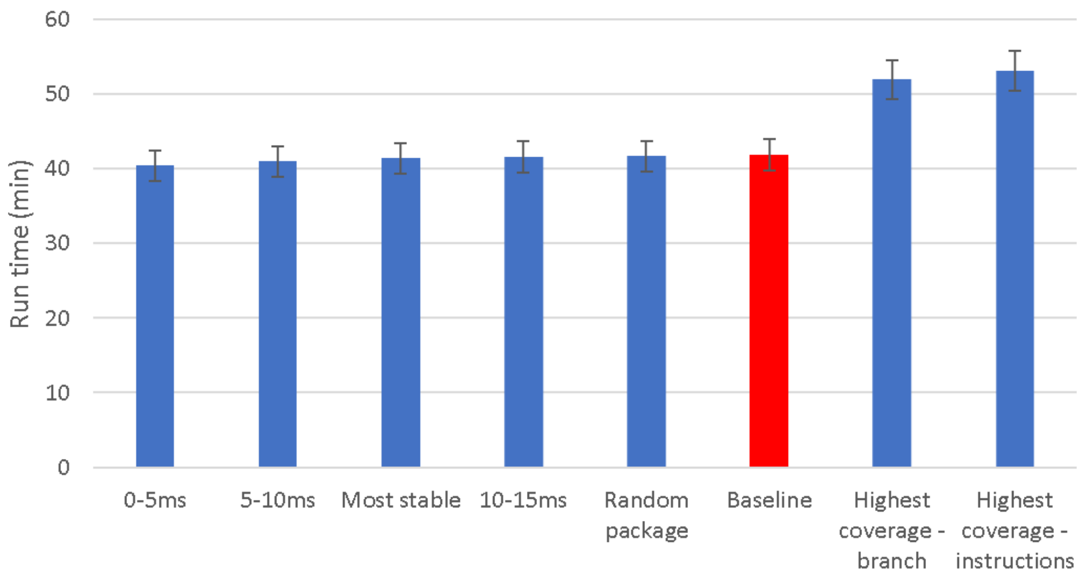


Figure 4.10: RxJava - suite running time of different approaches

The running time was calculated as an average of execution times of each cycle for each approach (Fig 4.10). When it comes to this metric in RxJava, almost all the optimization approaches performed similarly: around forty minutes on average for sixty benchmarks, with 0-5ms prioritization producing the fastest-executing suite. The only exceptions were highest coverage prioritization approaches, which were significantly slower: more than fifty minutes.

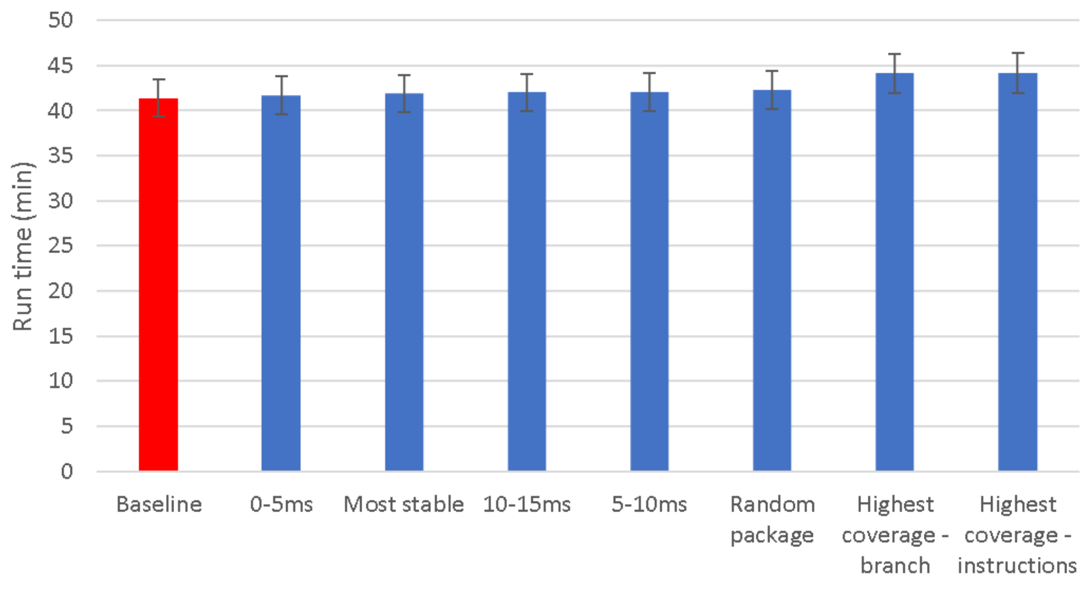


Figure 4.11: Mockito - suite running time of different approaches

Regarding the running time for the Mockito suites, all optimization approaches performed worse than the baseline (Fig. 4.11), but highest coverage methods produced the slowest-executing suites (about 10% more time to execute), in a similar fashion to the RxJava. All others methods produced slightly faster-executing suites, with 0-5ms prioritization being the fastest. However, in general there were no major differences between approaches when it comes to execution time in this project.

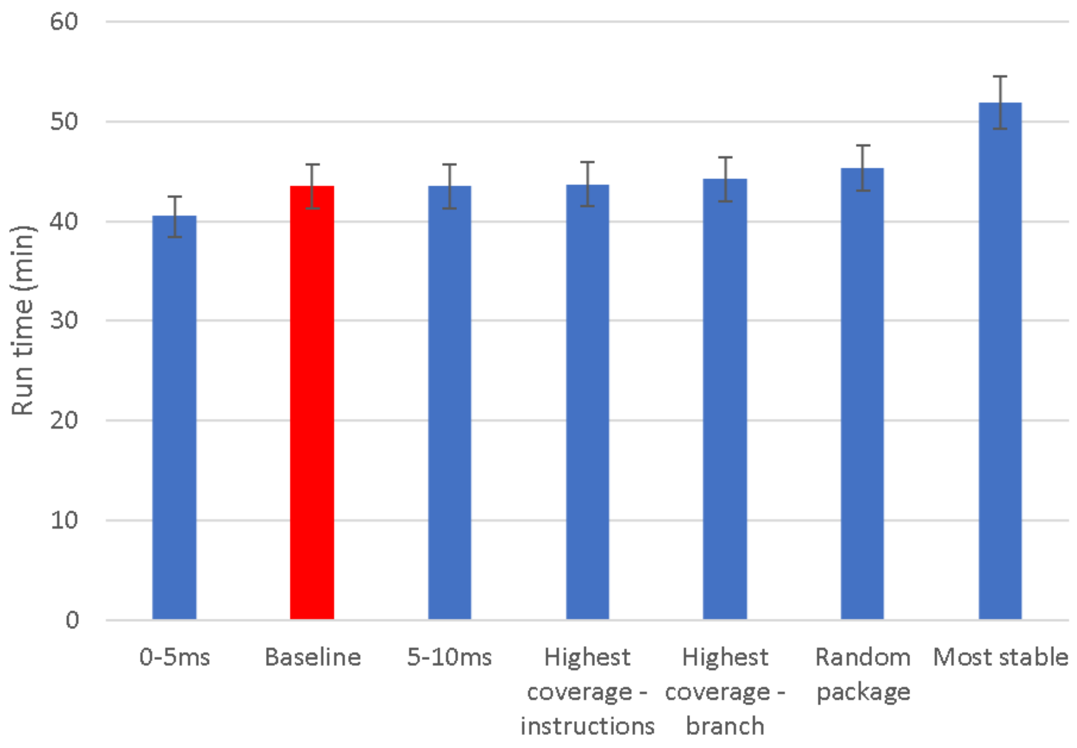


Figure 4.12: Stubby4j - suite running time of different approaches

In terms of running time, the optimization approaches applied to the Stubby4j performed as follows (Fig. 4.12). 0-5ms prioritization produced the fastest-executing suites, similarly to previous study objects, while other approaches produced comparable results, with only the most stable prioritization being an outlier, with the worst result for this project at over 50min average 60-benchmark suite running time.

5 Conclusion

In this study, various benchmark generation optimization approaches were designed and tested, producing different results.

5.1 Discussion

As seen from the results, the quality of the optimized benchmark suites varies greatly depending on the optimization method. The results of those approaches' application are presented in this section with corresponding research questions.

5.1.1 Research question 1

Different simple heuristics demonstrated varying effectiveness in terms of different metrics throughout the experiment. The in-depth description of their performance is presented in this section.

5.1.1.1 Random package selection

The random package selection was the least efficient approach among simple heuristics, for several reasons. Firstly, it consistently produced the least stable benchmark suites by a wide margin for all three assessed projects, regardless of their size. This means that the benchmark suite run results produced using this approach will vary considerably from one execution to another, severely decreasing the accuracy of the procedure and making it difficult to accurately measure the performance of the assessed software artifact. Secondly, its performance in terms of coverage was unsatisfactory as well, as it either produced generally poor coverage results (in RxJava) or failed to surpass baseline and produce improved results over other approaches (Mockito and branch coverage in Stubby4j). The one and only exception was instruction coverage in Stubby4j, where it produced the best results and was the only method to surpass the baseline. When it comes to running time, the results of random package selection heuristic were subpar as well, as it failed to surpass the baseline in two out of three projects and demonstrated similar results to other approaches in the third project (RxJava), without noticeable advantages.

Therefore, basing on very poor stability of suites generated by this approach as well as almost complete absence of advantages over other optimization approaches in terms of running time and coverage (with rare exceptions), it can be concluded that random package selection is not an efficient benchmarking optimization approach.

5.1.1.2 Running time interval prioritization

The running time interval prioritization heuristic produced variable results depending on the chosen interval. The in-depth discussion of those sub-approaches and their efficiency is presented below.

The 0-5ms prioritization approach produced suites with suboptimal coverage, never surpassing the baseline and being either inferior to most other approaches in terms of this metric (both coverage types in RxJava, and branch coverage in Mockito and Stubby4j) or having medium quality results (instruction coverage in Mockito and Stubby4j). This is not surprising, as faster-running test cases may potentially cover less code, resulting in less coverage for the converted benchmark suite. 0-5ms optimization results were noticeably better when it comes to stability. It was the third best-performing approach in terms of this metric in RxJava and Stubby4j (though still failing to surpass the baseline and performing significantly worse than highest-coverage approaches in both cases). It showed medium stability results in Mockito, surpassing the baseline and producing significantly more stable suites than 10-15ms prioritization and random package heuristic, although demonstrating less efficiency than other approaches. The running time has proven to be the biggest advantage of 0-5ms interval prioritization, as its suites consistently had the fastest execution time of all approaches and only failed to surpass the baseline in Mockito. This was expected, as fastest-executing test cases (compared to other optimization approaches) naturally produced the fastest-executing benchmark suite. It should be noted however that this advantage in execution time was quite small in all these cases, never surpassing 10%.
Basing on all these factors it can be concluded that prioritization of benchmarks whose "parent" test cases have running time in 0-5ms interval may potentially be an efficient optimization approach when benchmarking time has to be as short as possible, albeit the time savings may not reach more than 10%, as noted above.

The 5-10ms prioritization approach demonstrated somewhat inconsistent results when it comes to coverage. For example, in RxJava it produced medium coverage suites in terms of missed branches and instructions (noticeably less than the coverage prioritization approaches and less than the baseline, but slightly better than other approaches) and was one of the worst-performing approaches in terms of missed instructions in Stubby4j. On the other hand, it produced the highest-coverage suites (both in terms of branch and instruction coverage) in Mockito and demonstrated the highest branch coverage in Stubby4j, in both cases being the only approach to surpass the baseline. However, in both of these cases the advantage was quite small, less than 10% over the next-most efficient approach, in a similar manner to execution time advantage of 0-5ms prioritization approach mentioned above. When it comes to benchmark stability, the results were inconsistent yet again, as 5-10ms prioritization demonstrated one of the lowest suite stability figures in RxJava and Stubby4j, while being the third best method in Mockito in terms of this metric. Regarding the running time, this approach demonstrated very similar results to most other approaches (no noticeable advantages or disadvantages), with time difference between most of them being negligible (with the exception of outliers such as highest coverage approaches in RxJava and Mockito and stability prioritization in Stubby4j).

To sum up, the 5-10ms prioritization may not be an efficient optimization method because of its inconsistent performance in terms of static code coverage and benchmark stability and lack of noticeable improvements in terms of suite running time over other approaches.

The last time interval to be experimented with was 10-15ms. It was the only interval that was applied to two projects out of three, because of insufficient number of tests in this interval in the smaller Stubby4j project. In terms of static code coverage, it demonstrated medium-quality results in both Mockito and RxJava (in terms of branch and instruction coverage), being near the middle in every graph, failing to surpass the baseline and performing similarly to most other approaches with absence of significant advantages and disadvantages (with the exception of RxJava, where it performed noticeably worse than the highest coverage prioritization approaches). It was a bit unexpected since longer-running benchmarks can potentially cover more code (which is partially supported by the slow execution time of highest-coverage approaches' suites in the experiment) and result in higher-coverage suite. Some of the possible reasons for this are overlapping of code covered by different benchmarks as well as other reasons for longer test execution time unrelated to coverage such as time-inefficient or complex methods ran by some of the benchmarks in 10-15ms interval. In terms of benchmark stability, it produced mediocre results in both Mockito and RxJava (producing one of the least stable suites in the former and failing to surpass the baseline in the latter). In terms of running time, the 10-15ms prioritization did not differ noticeably from most other optimization methods, producing the suites with less than 10% difference in running time from them (with the exception of slowest approaches such as coverage prioritization in RxJava).

Considering both its mostly inefficient performance in terms of experimental metrics as well as its potential unsuitability for smaller projects (which may not have enough longer-running tests to create an optimized suite) as seen with Stubby4j, the 10-15ms time interval prioritization may not be considered an efficient optimization approach.

To sum up, most of the simple heuristics tested in the experiment may not be considered efficient benchmarking optimization approaches. The only exception to this is 0-5ms time interval prioritization, which could potentially be effective in situations with time constraints when its necessary to conduct the benchmarking as fast as possible. However, its rather small advantage in terms of time savings (less than 10%) as well as suboptimal performance in terms of static code coverage potentially make its usage for the benchmark suite optimization a questionable choice.

5.1.2 Research question 2

The prioritization of benchmarks by stability produced somewhat unexpected results. It demonstrated inconsistent results in terms of coverage, producing either suites with similar/slightly worse coverage than other approaches (in RxJava and Mockito) or demonstrating noticeably better results than most of them (in Stubby4j). However, the most surprising aspect of this method was the benchmark stability of the suites generated by it, which was either much worse than that demonstrated by most other approaches (in RxJava and Stubby4j, in both cases failing to surpass the baseline) or mediocre (in Mockito). Those results were rather unexpected, as prioritizing benchmarks by their "parent" test stability should have potentially led to creation of benchmark suites with the highest stability compared to other approaches. This is because stability of the converted benchmark should correspond to the stability of the unit test it was made from (according to Alexandersson [4]). A possible explanation for such stability paradox observed in the experiment is that stability of the unit test does not necessarily correspond to the one of its converted benchmark. Furthermore, the differences between each test-benchmark pair might be amplified with increasing size of the suite, leading to significantly lower stability results than expected. When it comes to running time, the results were inconsistent yet again, as stability prioritization approach demonstrated either similar results to other methods with negligible difference (in RxJava and Mockito) or performed the worst (it generated the slowest-running suite in Stubby4j by a wide margin).

Considering its unimpressive results in terms of coverage and running time as well as surprisingly bad stability of the benchmark suites generated with it, the benchmark prioritization by stability may not be considered an efficient optimization approach, at least when it comes to the benchmarks converted from unit tests as in this study.

5.1.3 Research question 3

As seen from the results, the static code coverage prioritization approach was unique among others in terms of performance. The main reason for this was extraordinary stability of the benchmark suites generated with it (both with instruction and branch coverage prioritization), which bested all other optimization approaches by several orders of magnitude in all tested projects. This was a really surprising and unexpected result, as such performance was expected from the stability prioritization approach mentioned above and not from coverage prioritization. One potential explanation for this is that long running time of the benchmarks contained in the suite (noticeably longer than other optimization methods in RxJava and somewhat longer in Mockito) results in their better stability, as performance fluctuations have time to subside during the benchmark run, resulting in more consistent results. When it comes to code coverage, another paradox emerged. While this approach performed the best in the RxJava (much better branch and instruction coverage, only method that surpassed the baseline, instruction coverage prioritization expectedly had much higher instruction coverage), the results in other projects were opposite. In Mockito, both instruction and branch coverage prioritization approaches actually produced suites with the least coverage (albeit not very different from the ones made with other optimization methods). In Stubby4j, both approaches performed noticeably worse than others, with differences of up to 20% (between random package and highest coverage approaches in missed instructions metric in Stubby4j). The last case was especially unexpected, as main potential advantage of this approach (outlined in the experimental setup) was improved coverage of its suites. Such inconsistent results could be explained bu overlapping of the code covered by multiple benchmarks, in different projects. This means that static code coverage of this approach may potentially be heavily dependent on the quality of the benchmark/test suite of the assessed project. If those were made by skilled developers/researchers and there is minimal overlap, this approach may produce good results in terms of coverage (as seen in RxJava). Otherwise, the results may be inconsistent, as seen in Mockito and Stubby4j. When it comes to running time, the coverage prioritization produced the slowest-executing suites in RxJava (with slowdown of up to 20% compared to other methods) and Mockito (less pronounced than in RxJava, slowdown lower than 10%, but still noticeable), while demonstrating medium-speed results in Stubby4j (similar execution time to most other approaches). The slower execution time may be explained by higher complexity of the benchmarks that cover more code (and thus have more instructions to execute).

To sum up, while the coverage prioritization approach (both instruction and branch subtypes) may be slower than other assessed methods and may produce benchmark suites with inconsistent coverage depending on the project, it consistently demonstrates the highest stability among the tested methods by a very wide margin and thus can be considered a very efficient optimization approach. Its usage can be advised for most optimization-requiring situations, with possible exception, when very strict time constrains are in place (then 0-5ms prioritization approach may be used instead, greatly sacrificing stability for minor gains in execution time).

5.1.4 Baseline as a potential optimization approach

Per experiment setup, the simple straightforward selection of random benchmarks was used as a baseline for comparison to other optimization approaches. However, it demonstrated quite decent results on its own. For example, it consistently produced suites with one of the best static code coverage values (both in terms of missed branches and instructions), only being surpassed significantly in RxJava by the highest coverage prioritization approach. However, its coverage advantage over most other approaches was not significant in most cases (less than 10%). It also demonstrated the best stability behind the highest coverage approaches in RxJava and Stubby4j (although still significantly worse than them), but was second-worst in Mockito by a wide margin in terms of this metric. In terms of running time, it performed similarly to most other approaches (except the slowest ones), and while it produced some of the fastest-executing suites in Mockito and Stubby4j, the advantage was still very small compared to other methods.

Considering these decent results it can be concluded that random benchmark selection can potentially be used as an optimization approach on its own, especially considering simplicity of its application when compared to most other methods (for example, it doesn't require running benchmark "parent" tests, measuring their coverage and stability). However, it still performed much worse than coverage prioritization in terms of benchmark stability and latter remains the most recommended approach overall.

To summarize the results, from all potential optimization approaches outlined in research questions and tested in this study, only one can be considered truly efficient and consistent when it comes to at least one metric. This is of course the highest coverage prioritization approach (both branch and instruction coverage) which produced the suites with significantly higher stability than other optimization methods. As noted above, it can be recommended for the majority of the optimization-requiring situations. The only other approach that consistently demonstrated some sort of advantage was 0-5ms time interval prioritization heuristic, which always produced the fastest-executing suites. However, unlike the coverage prioritization approach, the advantage of this one is much smaller, never exceeding 10% speed-up in all experimental subjects and its numbers in terms of other metrics (benchmark stability and instruction/branch coverage) were not very impressive as well. All other optimization approaches demonstrated mostly suboptimal and inconsistent results and can be considered inefficient.

5.2 Future work

Future work may be mostly focused on experimentation with other potential benchmarking optimization approaches as well as improvements and additions to the current method outlined in this study.

Firstly, usefulness of prioritization of benchmarks by the dynamic code coverage can be investigated. Since static code coverage prioritization proved to be extremely efficient in terms of benchmark stability in this study, the dynamic coverage method could also potentially display promising results. However, calculation of the dynamic code coverage is notoriously difficult, requiring run of the assessed program and subsequent analysis of its behavior via usage of collected runtime information such as file accesses, network activities, system call traces etc. [17]. Therefore, lots of time and computing resources may be needed for such experiment.

Secondly, the run results of the entire benchmark suite of the project can be used as a source for optimization approaches. For example, benchmarks may be prioritized by their score, confidence interval, stability (of benchmarks themselves and not their "parent" unit tests), some of the secondary metrics etc. This method may also potentially produce good results, since can be viewed as a direct improvement of the existing benchmark suite without its "parent" test suite involved. However, it can take a very long time to run such an experiment, as JMH suites can take multiple weeks to execute.

Thirdly, prioritization of benchmarks by other types of coverage can be tested. In this experiment only the branch and instruction coverage were assessed, thus it leaves at least class and method coverage as potential optimization approaches to be tested in the future. However, since there were almost no major differences in terms of performance between the tested instruction and branch coverage (except instruction coverage in RxJava, where instruction coverage prioritization performed much better), testing other coverage types prioritization may potentially be redundant and may even produce similar results to the already evaluated similar methods used in this study.

Fourthly, combination of the most efficient optimization approaches can be investigated. Theoretically, a well-executed combination of methods based on a rational selection can potentially be more efficient than trying to develop or find a single most efficient approach [23]. Initial scope of this study involved investigation of such combined methods, but since the experiments did not produce a selection of "most efficient" optimization methods, with only the static code coverage consistently producing far more stable suites than other approaches and 0-5ms prioritization always being slightly faster to execute than other methods. However, with addition of new optimization approaches such as, for example, dynamic code coverage prioritization and benchmark selection by score, a selection of the "best" approaches may potentially be obtained and their combination(s) investigated thoroughly.

Fifthly, the junit2jmh optimization tool can be modified and modernized. For example, support for JUnit 5 test cases can be added, as lots of the newer Java projects use it [24], making them incompatible with the current version of junit2jmh. The modified tool can then be tested with the new projects available for analysis, potentially allowing for a larger selection of study objects to experiment with.

Sixthly, the sample size can be greatly expanded. Three projects were assessed in this study because of time constraints, but with more time available (and with modifications to junit2jmh outlined above), the number of study subjects can be greatly expanded, potentially allowing to obtain more data on efficiency of different optimization approaches and study each one of them more deeply. Since this will require lots of time, more virtual machines may be needed for such setup, in order to run more experimental iterations concurrently.

5.3 Study limitations

Even though the study has been conducted effectively and produced conclusive results, it still had its limitations, which will be described in this section.

Firstly, only Java benchmarking process' optimization was assessed, meaning that results of similar procedure being conducted for the software implemented in other programming languages may be different. Additional experiments have to be conducted in order to assess this. Secondly, because of time constraints, only three projects were selected and analyzed. However, they all produced differently-sized benchmark suites and similar patterns were observed during the experimental runs (e.g. consistently stable suites produced by coverage prioritization approaches). Thirdly, only a part of possible optimization approaches were used (for reasons described above), thus limiting efficiency assessment's scope to them.

5.4 Conclusion

In this study, multiple potential optimization approaches for Java JMH benchmarking were designed, presented, tested in an experimental environment, and analyzed. The approaches included both - simple heuristics (such as random package selection as well as prioritization of benchmarks whose "parent" test cases ran in a certain time interval (0-5ms, 5-10ms and 10-15ms)), and more complex methods (such as prioritization of benchmarks by stability and static code coverage (in terms of missed instructions and branches)). The experiment included application of these approaches to the benchmarks generated from differently-sized unit test suites of three different open-source projects (large suite for RxJava, medium-sized for Mockito and smaller for Stubby4j). The results obtained from the experimental runs were then analyzed and visualized. The analysis revealed that only one optimization method consistently outperformed others by a great margin when it came to at least one metric (static code coverage prioritization approach and its outstanding benchmark stability). Another method was found to consistently produce the fastest-executing suites (0-5ms time interval heuristic), but its advantage over other methods wasn't nearly as extreme as the one of code coverage prioritization approach.

The results of this study can be used as a guidance for the optimization of Java JMH benchmark suites in projects (both pre-existing and newly converted from unit test suites), which can potentially improve the benchmarking process, making it easier to conduct and thus more accessible to both practitioners and researchers. This may in turn lead to improvements in the regression testing field, as benchmarking possesses unique advantages compared to other methods in this area (such as being faster to setup and execute than load testing for example). This can potentially result in substantial improvements of software quality (as defects will be found and mitigated earlier, when the cost of this procedure is much lower), achieving better performance of the IT sector.

However, the benchmarking area still remains understudied compared to other regression testing methods and substantial amounts of additional research may be needed in order to explore its true potential and put it to use in the software development industry.

Bibliography

[1] W. T. Tsai, X. Bai, R. Paul, and L. Yu, "Scenario-based functional regression testing," *In 25th Annual International Computer Software and Applications Conference, COMPSAC 2001*, pp. 496-501, Oct 2001.

[2] C. Laaber, H. C. Gall, and P. Leitner, "Applying test case prioritization to software microbenchmarks," *Empirical Software Engineering*, vol. 26, no. 6, pp. 1-48, 2021.

[3] 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:ACM*, pp. 119-130, 2018.

[4] N. Alexandersson, "JUnit-to-JMH: Automatic Generation of Performance Benchmarks from Existing Unit Tests in Java," 2021.

[5] I. Molyneaux, "The art of application performance testing: from strategy to tools," *O'Reilly Media, Inc*, 2014.

[6] P. Stefan, V. Horky, L. Bulej, and P. Tuma, "Unit testing performance in Java projects: Are we there yet?" *In Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, pp. 401-412, 2017.

[7] P. Chittimalli and M. Harrold, "Recomputing coverage information to assist regression testing." *IEEE Transactions on Software Engineering*, vol. 35, no. 4, pp. 496-501, 2009.

[8] "Java microbenchmark harness (jmh)," https://github.com/openjdk/jmh, 2022.

[9] H. G. C. Laaber, S.Würsten 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*, pp. 989-1001, Nov 2020.

[10] L. Cordeiro, D. Kroening, and P. Schrammel, " Benchmarking of Java verification tools at the software verification competition," *SV-COMP*, 2018.

[11] T. F. Bissyandé, F. Thung, D. Lo, L. Jiang, and L. Réveillere, "Popularity, interoperability, and impact of programming languages in 100,000 open source projects," *In 2013 IEEE 37th annual computer software and applications conference*, pp. 303-312, 2013.

[12] R. Khatchadourian, Y. Tang, and M. Bagherzadeh, "Safe automated refactoring for intelligent parallelization of java 8 streams," *Science of Computer Programming*, 2020.

[13] S. L. Pfleeger, "Experimental design and analysis in software engineering," *Annals of Software Engineering*, vol. 1, no. 1, pp. 219-253, 1995.

[14] "RxJava: Reactive Extensions for the JVM," https://github.com/ReactiveX/RxJava, 2022.

[15] "Mockito: Most popular Mocking framework for unit tests written in Java," https://github.com/mockito/mockito, 2022.

[16] "Stubby4j:HTTP/1.1, HTTP/2 and WebSockets stub server for stubbing distributed web services in Docker and non-containerized environments for integration and contract testing," https://github.com/azagniotov/stubby4j, 2022.

[17] C. Y. Huang, C. H. Chiu, C. H. Lin, and H. W. Tzeng, "Code coverage measurement for Android dynamic analysis tools." *In 2015 IEEE International Conference on Mobile Services*, pp. 209-216, 2015.

[18] "Intellij idea," https://www.jetbrains.com/idea/, 2022.

[19] "JaCoCo Java Code Coverage Library," https://github.com/jacoco/jacoco, 2022.

[20] "Gradle - a build automation tool for multi-language software development," https://gradle.org/, 2022.

[21] "RStudio - an integrated development environment (IDE) for R," https://www.rstudio.com/, 2022.

[22] "Microsoft Excel," https://www.microsoft.com/en-us/microsoft-365/excel.

[23] L. W. Schuwirth and C. P. Van Der Vleuten, "Different written assessment methods: what can be said about their strengths and weaknesses?" *Computer*, vol. 38, no. 9, pp. 974-979, 2004.

[24] "JUnit 5," https://mvnrepository.com/artifact/org.junit/junit5-engine/, 2022.

A
Appendix

A.1 Experimental results - raw data

A.1.1 RxJava

Optimization approach	**Missed instructions**	**Missed branches**	**Stability**	**Run time**
Highest coverage - instructions	108602	12311	515	53.2
Highest coverage - branch	124372	12415	221	52
Baseline	139798	13492	14392	41.9
5-10ms	142395	13669	25557	41
10-15ms	142449	13837	18303	41.6
0-5ms	145637	13965	16061	40.4
Random package	145879	13993	29448	41.7
Most stable	147277	14127	24061	41.4

A.1.2 Mockito

Optimization approach	**Missed instructions**	**Missed branches**	**Stability**	**Run time**
Highest coverage - instructions	24481	2117	206	44.2
Highest coverage - branch	24139	2083	221	44.1
Baseline	23476	2038	6224	41.4
5-10ms	23443	2011	2911	42.11
10-15ms	24047	2080	5540	42
0-5ms	24052	2089	3460	41.7
Random package	23804	2066	6416	42.3
Most stable	24135	2072	3435	41.9

A.1.3 Stubby4j

Optimization approach	**Missed instructions**	**Missed branches**	**Stability**	**Run time**
Highest coverage - instructions	10674	729	284	43.7
Highest coverage - branch	10507	701	368	44.2
Baseline	8372	634	9783	43.5
5-10ms	9235	629	26089	43.51
0-5ms	9176	668	12490	40.5
Random package	8104	655	32963	45.3
Most stable	8511	649	27357	51.9