Master Thesis

30 January, 2020

Software Microbenchmark Reconfiguration

Reducing Execution Time without Sacrificing Quality

Stefan Würsten

of Stäfa, Switzerland (14-725-931)

supervised by

Prof. Dr. Harald C. Gall

Christoph Laaber

University of Zurich[UZH]

s.e.a.l.

software evolution & architecture lab

Master Thesis

Software Microbenchmark Reconfiguration

Reducing Execution Time without Sacrificing Quality

Stefan Würsten

University of Zurich[UZH]

s.e.a.l.
software evolution & architecture lab

Master Thesis

Author: Stefan Würsten, stefan.wuersten@uzh.ch

Project period: 01.08.2019 - 30.01.2020

Software Evolution & Architecture Lab
Department of Informatics, University of Zurich

Acknowledgements

This master's thesis is the final milestone for my graduation. I would like to thank all those who supported me during my studies at the University of Zürich.

I would like to thank Prof. Dr. Harald Gall for giving me the opportunity to write this master's thesis in the software evolution and architecture lab at the University of Zürich. Special thanks go to my advisor, Christoph Laaber, for his amazing support and guidance in our meetings throughout the entire six-month period. I already possessed some technological and theoretical experience in the area of performance engineering due to my master project. However, the chance to receive weekly feedback on current problems, design decisions and the general plan for how to answer the research questions was amazing. Without such great support, it would have been impossible to create such a polished result within six months. The time spent writing this master's thesis will remain in my memory as a period of great challenge and profound education.

Abstract

In recent years, performance testing has gained increasing popularity. Such tests measure a software component's execution time. Compared to traditional functional tests, it is not sufficient to execute a test once. Instead, performance should be measured multiple times to obtain a representative distribution. However, this results in a far more time-intensive execution.

First, we investigate how developers currently configure software microbenchmarks, including how the proposed default values are modified and how this affects the execution time. Our analysis reveals that software microbenchmarks are often never modified after being written. Many projects reuse the default values for certain parameters. However, if user-defined values are set, this often results in a shorter execution time.

Second, we investigate the consequences of dynamically determined execution configurations. In regular intervals, we check the characteristics of the performance distribution and decide whether more data points are required. We compare our novel approach with the standard execution. For a preponderant majority of software microbenchmarks, the novel approach produces a similar performance distribution for which an A/A test cannot detect significant differences. However, depending on the stoppage criteria, up to 82% of the execution time can be saved. Our novel approach should help developers to shorten the time-consuming execution while still producing a sound result.

Zusammenfassung

Die Popularität von Performance Tests ist über die letzten Jahre stetig gestiegen. Solche Tests messen die Ausführungszeit einer Softwarekomponente. Im Vergleich zu herkömmlichen funktionalen Tests genügt es aber nicht diesen einmal auszuführen. Die Performance sollte mehrfach gemessen werden, um eine representative Verteilung zu generieren. Diese zahlreichen Ausführungen benötigen aber eine wesentlich höhere Ausführungszeit.

Zuerst beleuchten wir wie Entwickler aktuell solche Software Mikrobenchmarks konfigurieren. Geprüft wird, inwiefern die vorgeschlagenen Standardwerte modifiziert werden und welche Auswirkungen dies auf die Ausführungszeit hat. Unsere Untersuchung hat gezeigt, dass Software Mikrobenchmarks häufig nur einmal geschrieben und danach nie mehr angepasst werden. Viele Projekte greifen bei einigen Parametern auf Standardwerte zurück. Werden eigene Werte verwendet, resultiert daraus häufig eine kürzere Ausführungszeit.

Anschliessend untersuchen wir die Auswirkungen, wenn statische Konfigurationen dynamisch bestimmt werden. In regulären Intervallen prüfen wir die Eigenschaften der Performanceverteilung und entscheiden, ob noch mehr Datenpunkte benötigt werden. Wir vergleichen dabei unseren neuartigen Ansatz mit dem traditionellen. Für eine überwiegende Mehrheit der Software Mikrobenchmarks führt unsere neuartige Vorgehensweise zu ähnlichen Performanceverteilungen, für welche statistische A/A Tests keine signifikanten Unterschiede feststellen können. Jedoch kann die Ausführungszeit je nach gewähltem Abbruchkriterium um bis zu 82% reduziert werden. Unser neuartiger Ansatz soll Entwicklern helfen, die zeitintensive Ausführung zu verkürzen, ohne Abstriche bei der Resultatqualität zu machen.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Performance testing has been gaining increasing popularity in recent years. Software microbenchmarks comprise one type of performance tests, measuring the performance of small, isolated components [34]. They are similar to unit tests, which focus on functional requirements. However, performance tests should be executed multiple times to obtain a representative performance distribution. This results in a more time-intensive execution compared to unit tests. The question thus becomes how benchmark execution should be configured so that a sound result is measured, but no time is wasted. Previous work has revealed that numerous developers struggle to find appropriate execution configurations [44]. Additionally, several bad practices can negatively affect the benchmark result [13]. Writing and evaluating performance tests is not a trivial task. A prototype exists that attempts to automate the generation of software microbenchmarks [44]. However, the time-consuming execution is still not eliminated with well-written benchmarks.

Furthermore, research has suggested that non-functional tests, such as performance tests, should be automated and made part of the continuous integration pipeline [17]. However, this poses a challenge, as the execution is costly and slows down the continuous integration feedback cycle. Many practitioners have circumvented this problem by not executing the performance testings on every build [7]. Unfortunately, a disadvantage of not executing tests on each commit is that identifying the root cause for a problem becomes more difficult [15]. Several solutions have already been proposed to reduce the execution time. One option involves only executing a subset of all benchmarks on every commit [39]. With static code analysis and the historical information of previous executions, benchmarks are selected that offer the greatest chance of introducing performance regressions. Another option consists of dynamically deciding how long to execute performance tests. Such statistics-based performance testing has already been implemented for load tests [23]. Compared to the static execution, sequential testing is performed during the performance measurement, with a stoppage rule deciding when to stop the measurement [55]. Metrics such as the memory usage or throughput are continuously monitored [3]. In regular intervals, an algorithm decides whether a point is reached where it is highly probable that the metric characteristic does not change anymore.

Existing statistic-based performance testing approaches do not focus on software microbenchmarks. However, microbenchmarks are short-running compared to load tests, where the execution takes days. A research gap exists concerning how software microbenchmarks are configured. The configuration defines how often a benchmark is executed and influences the execution time. Additionally, it has not yet been examined whether and to what degree statistics-based performance testing is applicable for software microbenchmarks. Additionally, while software microbenchmarks are short-running compared to load tests, it remains unclear how much execution time can be saved, and how the execution result is affected. We assume that no universal execution configuration exists that always offers an optimal balance between no time wasted but a sound result still being produced. Depending on the task performed and the required resources,

the optimal configuration varies. Manually identifying the optimal balance is time-consuming. A more viable solution is to dynamically decide when the result is sound enough to stop the execution.

This thesis focuses on two aspects: First, we mined 753 Java projects to obtain insights into how developers currently deal with execution configurations. Compared to previous work [32, 52], we perform a fine-tuned investigation of which execution parameters are chosen, as well as how they affect a benchmark's execution time. Based on the quantitative data, we suspect that identifying good execution configurations comprises a non-trivial task. As such, we presented a novel approach in the second part of this work.

The effective benchmark configuration is not a priori defined. A developer simply defines an upper-bound execution configuration and stoppage criteria. The performance-testing framework itself measures whether the benchmark produces a sound result and then stops the execution. Such a statistics-based approach offers the advantage of no time being wasted. We compare three different stoppage criteria — the CoV, CI width and divergence — against the standard execution approach in terms of result quality and execution time. The reconfigured execution produces a high result quality if the characteristics of the resulting distribution is not significantly different from the standard execution result.

The results indicate that software microbenchmarks are usually written once, and then never updated. Depending on the configuration parameter, the default value is used between 55.2% and 98.7% of the time. For 24.5% of the benchmarks, the execution configuration is not modified. Changes in the configuration often results in a shorter execution time, and potentially less sound results. Projects with more stars and contributors do not configure benchmarks differently. The analysis of the statistics-based execution approach demonstrates that the divergence stoppage criteria save 79.5% of the time. The divergence stoppage criteria possesses an average mean change rate over all benchmarks of 2.4%. The overhead for the statistical testing during the execution equals 4.32% for the divergence stoppage criteria. For most benchmarks ($\sim$80%), the execution result is not insignificantly changed compared to the standard execution.

Following these findings, we conclude that benchmarks are not configured as proposed by the default values; in the majority of these cases, the execution time is reduced. A common practice for reducing the execution time is to set the number of forks to one. However, multiple forks are important to obtain a sound result. The empirical evaluation of the dynamic approach indicates the execution time can be reduced and the result characteristic remains similar. The CI width stoppage criteria do not save as much as the others, but the best results are produced. The divergence stoppage criteria save more time, but the execution results differ more from the standard execution.

1.1 Motivating Example

A project possesses 50 software microbenchmarks written in JMH — the de facto standard for microbenchmarking in Java. An average developer is not an expert in performance engineering, so the proposed default values of the framework are employed. This means that the execution time of each benchmark takes 500 seconds. This results in a project test suite execution time of 6.94 hours. If five commits are performed per day, and benchmarks should be executed as part of the continuous integration pipeline, we run into the issue that executing on a single machine takes too long, and the queue of pending executions grows. The benchmark execution shall not be parallelized on the same machine, as otherwise, the results are falsified, because resources are shared. Several bare-metal machines would be required to divide the workload of executing all benchmarks on each commit. However, physical computing resources are costly. It is unrealistic for numerous projects to execute all benchmarks on each commit if several bare-metal machines are necessary for benchmarking.

Nevertheless, if all benchmarks should be executed on a single machine on each commit, the execution time must be drastically reduced. An alternative is to execute the software microbenchmarks with the reconfiguration approach to optimize the result quality execution time trade-off. The benchmark mode is set to reconfigure, and a stoppage criteria is chosen. With the default stoppage criteria divergence, 79.5% of the time is saved. The result distribution is similar to the standard execution. For over 75% of the benchmarks, the mean change rate is, at most, 2% changed. The number of forks is reduced on average from 5 to 4.1, while the warmup time is reduced from 50 seconds per fork to 14.1. As sound results are produced after the warmup phase, the measurement phase is shortened from 50 to 10 seconds.

1.2 Contribution

This thesis makes the following contributions:

- We create the GitHub JMH dataset of 13'387 JMH benchmarks from 753 GitHub projects. The dataset contains the mined execution configurations and features usage metrics.
- We extend the Bencher tool with several new features such as source code parsing or the extraction of JMH parameters in state objects.
- We propose a statistics-based execution approach for software microbenchmarks and implement a prototype in JMH.
- We empirically evaluate three different stoppage criteria in terms of result quality and execution time.

1.3 Thesis Outline

The remainder of this thesis is structured as follows: Chapter 2 provides necessary background information, including basic terms of software microbenchmarking, the JMH framework and other works in this area. Chapter 3 presents the research questions we seek to answer. Furthermore, the overall approach is summarized concerning how the data collection and the different evaluations are connected. Next, we analyze the state of practice regarding how developers employ JMH in Chapter 4. First, the data extraction is described. Afterwards, the different aspects of the execution configuration are examined. Our novel reconfiguration approach is presented afterwards in Chapter 5. The high-level concept of how the dynamic stoppage criteria works is illustrated. Additionally, the modified JMH version is presented. Next, an evaluation is performed to analyze result quality and execution time. In Chapter 6, our findings are discussed and improvements provided concerning both how developers should configure benchmarks as well as the JMH implementation. Additionally, the limitation and threats of validity are elaborated. Lastly, this work is concluded in Chapter 7, where the main findings are summarized and possible future work presented.

Chapter 2

Background and Related Work

The following chapter introduces basic terms and discusses the central topics upon which this thesis is based. First, performance testing and existing related works are explained. Following this, the utilized statistic methods are presented.

2.1 Performance Testing

Performance testing describes an area in software engineering where non-functional metrics, such as the throughput of a software component, are evaluated [56]. Compared to functional testing, such as unit tests, performance test do not simply pass or fail; rather, the return value is a distribution of measurement values. Performance regression testing is required to verify whether performance degradation occurred compared to the previous version. Such an execution of performance regression tests often takes several hours or days [18].

Several types of performance tests exist [34]. The first group consists of performance smoke tests. These high-abstraction tests measure end-to-end performance. A project typically possesses one or two of such tests so as to quickly identify large performance regressions. The second group comprises microbenchmarks. Like unit tests, they are only written for a small piece of code. An extensive test suite is required to test each detail of the production code. One-shot-performance tests form the next type. They are often written during development to support the decision process, such as by comparing the same functionality of different frameworks. Often, such tests are source code artifacts and no longer deliver an additional benefit to the project. Lastly, regular functional tests verify the performance behavior with assertions.

In this work, we focus solely on software microbenchmarks, sometimes referred to as performance unit tests [25]. Such software microbenchmarks repetitively measure the performance of a low-level code component. Compared to load or stress tests, only short-running scenarios are tested. A typical software microbenchmark tests the performance of a single method, an algorithm implementation or a specific data structure [13]. Most frameworks execute a single benchmark as many times as possible in a certain time period. During the execution performance, metrics such as execution time or heap utilization are recorded. Afterward, the metrics are aggregated and reported.

The following presents the JMH framework and studies on performance testing frameworks. Afterwards, a microbenchmark generation approach is illustrated. Following this, alternative execution methods are elaborated and performance testing as part of the continuous integration pipeline discussed. Then, existing literature on misleading results and performance degradation is detailed. Lastly, a statistics-based approach for how to dynamically stop load tests is introduced.

2.1.1 Java Microbenchmark Harness (JMH)

The Java de facto standard for microbenchmarking is JMH[1], developed by the OpenJDK team. A developer can define benchmarks through annotations. Listing 2.1 presents a modified example benchmark from *ReactiveX/RxJava*. A public method is annotated with `@Benchmark` and can be run via the Java API or a generated jar file, which is executed over the command line.

```
1 @Fork(1)
2 @Warmup(iterations = 10, time = 1, timeUnit = TimeUnit.SECONDS)
3 @Measurement(iterations = 20, time = 1, timeUnit = TimeUnit.SECONDS)
4 public class OperatorFlatMapPerf {
5   @State(Scope.Thread)
6   public static class Input extends InputWithIncrementingInteger {
7     @Param({"1", "1000", "1000000"})
8     public int size;
9   }
10
11  @Fork(2)
12  @Benchmark
13  public void flatMapIntPassthruSync(Input input) {
14    input.flowable.flatMap(new Function<Integer, Publisher<Integer>>() {
15      @Override
16      public Publisher<Integer> apply(Integer v) {
17        return Flowable.just(v);
18      }
19    }).subscribe(input.newSubscriber());
20  }
21 }
```

Listing 2.1: Modified JMH example benchmark from *ReactiveX/RxJava*

A benchmark normally possesses a certain state while measuring the performance. Such a state can be outsourced into classes, annotated with `@State` as in the listing the `Input` class. A benchmark uses the class where it is defined and the method arguments as state objects if they feature the corresponding `@State` annotation. JMH supports parameterization of benchmarks. Parameterized tests are a technique to execute a single test method with different parameter combinations multiple times [53]. Parameters are state object instance variables, which are annotated with `@Param`. Each parameter accepts a set of values. In this example, three parameterization combinations are defined. If multiple JMH parameters are present, the Cartesian product is built. As an example, if a benchmark features the parameters $a = 10$, $b \in \{0, 1, 2\}$ and $c \in \{"a", "b"\}$, this results in six parameterization combinations.

Optionally, state objects can feature methods annotated with `@Setup` and `@Teardown` to initialize and reset the environment, respectively. As an annotation property, a developer can specify how the state object is shared (between benchmark, threads or groups) and when the fixture methods are called (before/after each trial, iteration or invocation). In the example, the state object is shared within the thread, and no setup and tear down is defined.

In JMH, a class where benchmarks are defined, or the method itself can have optional annotations to specify how to execute the benchmarks. In the example, the number of measurement iterations is set with `@Measurement(iterations = 20)`. If no execution configuration is present, the default values are used. If a class and method annotation are present, the method annotation

[1] `https://openjdk.java.net/projects/code-tools/jmh`

Configuration parameter	Abbreviation	JMH version < 1.21	JMH version ≥ 1.21
Warmup forks	wf	0	0
Warmup iterations	wi	20	5
Warmup time	wt	1 second	10 seconds
Forks	f	10	5
Measurement iterations	mi	20	5
Measurement time	mt	1 second	10 seconds
Mode	-	throughput	throughput

Table 2.1: Default configuration of JMH

possesses a higher priority than class annotation. The example benchmark is executed with two forks as the value set on method level features a higher priority. A developer must not override all configuration options. For example, no `@BenchmarkMode` annotation is present, so the execution is performed with the default mode. Lastly, if a benchmark is executed, optional Command Line Interface (CLI) arguments can be passed to override the annotations. The default values of JMH are listed in Table 2.1. In version 1.21, the default values are updated because the OpenJDK development team found some environments and configurations that possessed a poor time-to-performance [51]. The number of iterations is reduced, but each iteration now takes longer. The number of forks is also reduced, because five forks are sufficient to capture the variance between trials [50].

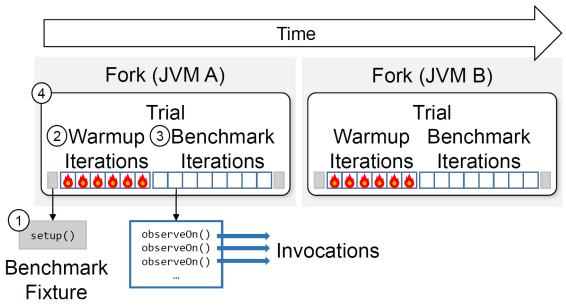


Figure 2.1: Execution flow of JMH for a benchmark with 2 forks, 6 warmup iterations and 8 measurement iterations [13]

Figure 2.1 summarizes the steps performed by JMH to measure the benchmark performance. (1) First, all required state objects are initialized. Here, they are initialized before each trial. (2) Next, the warmup iterations are executed. A developer can specify the time per iteration `@Warmup(time = 1, timeUnit = TimeUnit.SECONDS)` and the number of warmup iterations `@Warmup(iterations = 10)`. Warmup iterations are identical to measurement iterations, except their results are ignored for the evaluation. If a new Java Virtual Machine (JVM) is started first, a considerable amount of just-in-time compilations is performed. Such behavior influences the performance, which is why the data collected during the warmup phase is discarded [20]. The JMH default warmup configuration may not be sufficient to reach such a steady state. JMH does not verify and warn if no steady state is reached. (3) Next, the measurement iterations are executed. During each iteration, the benchmark is called as often as possible until the time per iteration is reached. (4) A run of steps (1) – (3) is called trial. An execution contains multiple sequentially executed trials. Each trial is performed in a separate JVM to reduce the influence

of these JVM optimizations. The number of runs is specified via the annotation `@Fork(1)`. If results are collected for the entire project test suite, the output is written to the console and/or JavaScript Object Notation (JSON) file.

The warmup time is composed of two parts: First, warmup forks can be defined, where the whole trial is not considered for the final result. The second part involves each trial's warmup phase, as explained previously. This provides Formula 2.1:

$$warmupTime = wf * (wi * wt + mi * mt) + f * wi * wt \tag{2.1}$$

The measurement time is simply the time where data is collected and persisted. This provides the following:

$$measurementTime = f * mi * mt \tag{2.2}$$

The total execution time is the sum of warmup and measurement time. This ends in the following:

$$executionTime = warmupTime + measurementTime \tag{2.3}$$

JMH offers two ways to avoid dead code elimination, loop optimization and constant folding [14]. First, the computed value can be used as a method return value. However, in more complex cases, multiple intermediate results are computed, and combining these into a return value is not trivial [27]. For such cases, the JMH framework provides the `org.openjdk.jmh.infra.Blackhole` class, which features a `consume` method. The blackhole can be defined as a method argument of the benchmark and can be called multiple times inside the method body to consume these unused intermediate results.

2.1.2 Performance Testing Frameworks in Practice

In 2017, Stefan et al. performed a quantitative study on the usage of performance testing frameworks, providing various statistics for Java projects on GitHub [52]. First, they mined over 99'000 projects to determine which projects employed a testing framework. JMH was by far the most common performance testing framework. However, only 0.28% of all projects implemented performance tests. In a developer survey, the developers argued that the trust in results, active maintenance and good documentation represent the top reasons for implementing performance tests with JMH. Developers often further claimed that they only updated benchmarks if performance issues occurred. Only one fourth of the developers reported maintaining benchmarks as often as other code in the project.

Stefan et al. also classified the projects into categories to determine which type of projects often write performance tests. The most common categories were database systems, distributed systems, algorithms and data structures. Some projects measured their own performance, while other developers sought to compare the performance of external libraries. Comparing different libraries or algorithms can help to make design decision more easily. As for how often developers wanted to run benchmarks, 47% of the developers answered on each commit, and 37% only on each release. Many developers further stated that running performance tests on each commit is an open issue. 77% of the developers used a manual process to conduct the performance tests, 13% automated the plotting of diagrams, and only 6% employed an automated evaluation of the performance results.

Costa et al. empirically analyzed bad practices of JMH benchmarks [13], focusing specifically on five bad practices. The first was that, if not all intermediate results of a benchmark are returned or consumed by a blackhole, the JVM then eliminates the dead code. The second bad practice involved performing unsafe loops. Traversing an array of elements is fine, but the JVM is able

to unroll loops, which is an extensive performance optimization. The third bad practice consists of using the final keyword for the benchmark input, as constant folding can be optimized by the compiler. The fourth bad practice concerned running fixture methods before and/or after each invocation, as executing the setup and teardown method will highly influence the performance result. The final practice involved setting the fork parameter of a benchmark to zero. In such a scenario, the trials are not executed in separate JVMs, and incorrect results are thus produced. Their analysis of 123 projects revealed that bad practices occur frequently.

In a second part of their work, they fixed some of the benchmarks to illustrate the performance impact. As an example, for 35% of the benchmarks that do not properly use blackhole objects or return, the intermediate results are significantly influenced and had a large effect size. They suspected that the developer guides were not sufficient for rigorous performance testing. An average developer lacks sufficient knowledge regarding how the JVM internally works and does not understand the importance of avoiding these bad practices. A possible solution is to improve the developer tooling. Either the Integrated Development Environment (IDE) should warn against bad practices during the development process, or JMH should check the benchmark configuration and implementation before executing them. However, it remains unclear whether a developer can itself fix the benchmark. In the best case, JMH does automatically perform some simple code transformation and fix common bad practices.

2.1.3 Automated Microbenchmark Generation

Rodriquez et al. suggested an approach where benchmarks are automatically generated to prevent the two most common mistakes: constant folding and dead code elimination [44]. Frameworks such as JMH simplify the measurement, but avoiding JVM optimizations remains challenging, as technical support is limited [42]. Their tool, called AutoJMH, requires an input of which code segment should be performance tested. Their process creates a set of the software microbenchmarks. They employed a technique called "sink maximization" to avoid dead code elimination and constant folding. In their evaluation, they first used real-world projects to analyze the limits of their automated benchmark generation approach. The main reason why the tool was unable to generate a benchmark was an unsupported variable. A reason why AutoJMH does not always correctly extract and wrap information would be loops. Next, they compared the performance of auto-generated benchmarks against handwritten benchmarks. The statistical evaluation revealed that their performance was similar. Lastly, they compared the tool-generated benchmarks against benchmarks handwritten by engineers lacking experience in performance engineering. AutoJMH avoided most of the mistakes made by the inexperienced developers.

2.1.4 Alternative Approaches for Executing Project Test Suites

A main issue of performance tests is that their execution is costly. However, running benchmarks on only a nightly or weekly build complicates the root case analysis of performance degradations. Oliveira et al. offered an approach where a subset of benchmarks is executed on each commit [39]. Their tool, Perphecy, collects static information from each commit without executing the benchmarks. Additionally, the tool stores dynamic information on benchmarks executed in the past. In the next step, indicator metrics are evaluated to characterize a change type. Lastly, Perphecy identifies a benchmark-specific threshold for each indicator, later used for predictions. In their experiment, they reduced the execution time of the test suite by 83% while continuing to detect 85% of the performance regressions.

Traditionally, performance tests are executed on bare-metal machines. Compared to virtual environments, such a physical server offers an advantage in that no noisy neighbor negatively influences the performance results [4]. However outsourcing computing resources to the cloud

is cost-effective [9]. As numerous companies and projects do not execute performance tests on dedicated hardware, executing the test on virtualized resources such as cloud infrastructure represents a realistic scenario.

Laaber et al. investigated whether it is always a bad idea to run performance tests in public clouds [33]. They first analyzed how variable different cloud providers and their cloud types are. Depending on the benchmark, the variability can vary widely. Some benchmarks are stable in all environment, while others are stable nowhere, and some benchmarks are only stable on some cloud instances. Second, they investigated whether it is still possible to detect performance degradations with a low false-positive rate in cloud environments. In approximately 80% of the benchmark-environment combinations, the slowdowns were detectable. However, a substantial number of executions on multiple cloud instances was necessary to yield a low false-positive rate.

Abedi and Brecht analyzed different execution strategies in the cloud [2]. They revealed that using a randomized, multiple interleaved trials strategy reduced the unpredictability of the results. As other jobs on the same physical machine often run in regular intervals, randomizing the benchmark order inside a trial distributes the negative influence over all benchmarks.

2.1.5 Performance Testing and Continuous Integration

Fagerstrom et al. conducted a case study analyzing challenges of non-functional regression test automation and offered improvements for how to master these challenges [17]. A technical challenge is that performance tests slow down the continuous integration feedback cycle. Duvall et al. suggested running tests on each commit [15], which is impossible with such time-consuming performance regression tests. Several practitioners mentioned in interviews that if a test fails, it takes a considerable amount of time and effort to identify the problem introducing the error. Another challenge is that requirements are often unclear when a non-functional test should pass or fail. A common suggestion from practitioners is that statistical intelligence should filter out noise, detect performance degradation based on the benchmark's history, and predict the future performance.

Bezemer et al. conducted a survey on how performance is addressed in DevOps [7]. Most companies analyze the performance, but only one third conducted performance analysis at least weekly. Performance engineering represents a complex task that requires understanding the software life-cycle. However, companies often feature a lack of knowledge on the underlying science, impeding their use. A less intensive performance engineering technique is desirable, requiring less in-depth background knowledge. Lastly, performance engineering tools are not nicely integrated into the DevOps pipeline. They recommended providing plugins for common continuous integration tools, such as Jenkins[2], where performance tests can be configured, automatically executed and evaluated.

The CSPA tool[3] is a continuous integration plugin that automates the execution of software microbenchmarks. Currently, only JMH benchmarks are supported. In the project settings, the performance tests are configured. A subset of tests can be selected for the execution. Tests are either executed on bare-metal machines or in the cloud. Different execution strategies are supported. Additionally, different A/A test are provided to automatically evaluate the performance and detect degradations. After executing the performance tests, several figures and tables summarize the execution and how the result differs from the previous execution.

[2] `https://jenkins.io`
[3] `https://github.com/sealuzh/cspa-core`

2.1.6 Misleading Results and Infrastructure caused Performance Degradation

Previous studies found common patterns that cause misleading performance results [25, 48]. A well-known issue is that a measured benchmark is not in a steady state. Warmup artifacts, such as class loading and just-in-time compilation negatively affect the performance. Another problem is that code is optimized by the compiler, producing non-realistic performance results. The garbage collector and its configuration (i.e., heap size) influence the performance, as demonstrated by Georges et al. [20].

Gil et al. investigated which factors influence the microbenchmark result [21]. One finding was that there is no single convergence point if a benchmark is in a steady state. If the JVM is restarted, the same benchmark produces another performance distribution, which is significantly different. There is a range of convergence centers, which in their experiments is approximately 2% wide. This explains why multiple forks are necessary to rigorously test microbenchmarks.

If a benchmark is executed, as many confounding factors as possible should be controlled by the developer. This produces more precise performance results. However, Harji et al. conducted experiments revealing that the Linux kernel evolution produces performance degradations [22]. In some cases, it is reasonable to not use the latest kernel version. Updating the kernel requires sanity checks on how the performance has changed.

Shipilev, a core developer of JMH, analyzed the granularity of the `System.nanoTime()` method in 2014 [49]. Granularity refers to the time difference between two method calls with different answers. In his experiment, the granularity equaled approximately 30 nanoseconds. The behavior is platform-dependent on how the system clock is implemented [25]. For example, Windows features a granularity of approximately 370 milliseconds. We can conclude that benchmarks with an execution time of nanoseconds cannot always be measured exactly.

The program performance depends on the experimental setup. Mytkowicz et al. revealed that the Unix environment size and the linking order create a measurement bias [37, 38]. As an example, each memory image of a process contains environment variables such as `HOME`. If the length of these (unused) environment variables is changed, this can produce performance slowdowns. They revealed for C programs that speedups or slowdowns of 10% are possible for the same workload. They further proposed an approach to minimize the measurement bias with randomized setups. Curtsinger and Berger implemented this approach [12]. Code changes such as adding a stack variable affected the order of the heap allocation, resulting in a different performance. Their tool, called "STABILIZER", randomizes the placement of stack frames, heap objects and program functions during the execution. With a performance overhead of less than 7%, they eliminated significant performance slowdowns.

2.1.7 Statistics-based Load Testing

Some previous work has investigated how load tests can be previously stopped and execution time saved. Load tests measure how a system's response time and throughput changes if it must handle simultaneous interaction [6]. However, it is difficult to construct test cases where performance degradation is quickly found. Often, numerous combinations of input parameters exist. Selecting an effective subset of input parameters represents a fundamental issue for reducing the execution time of load tests [36]. Luo et al. accordingly developed a tool called FOREPOST, which trains rules from the execution trace to later select effective test input data. Such an approach helps to identify performance issue in a shorter period of time.

Alghmadi et al. presented an approach that recommends when to stop a performance load test [3]. Such tests often run over a long period to uncover performance issues, such as memory leaks. Their tool tracks various metrics, such as the Central Processing Unit (CPU) usage, memory

usage and response time. The tool also periodically measures and stores these metrics. The tool further analyzes how the absolute and delta values change over time and computes the likelihood of repetitiveness for each metric. If it is likely that a stable point is reached, the test can be stopped. Such an approach often stops early because the tool focuses on application internal factors [23]. If an external factor, such as a noisy neighbor, influences the performance result, then the approach often features low accuracy.

Kullback-Leibler Divergence

He et al. proposed a performance testing methodology called PT4Cloud with a statistical stopping condition to dynamically determine whether the performance results seemed stable [23]. Running performance tests always presents a trade-off between execution costs and accuracy. However, PT4Cloud helps to manage this trade-off. The application under test is executed for a certain time period. Afterward, sequential testing helps to determine whether the new measured data points change the characteristic of the performance distribution compared to the previous runs. If the characteristic difference is classified as insignificant, the execution is stopped. Otherwise, the procedure is repeated until the difference is classified as insignificant.

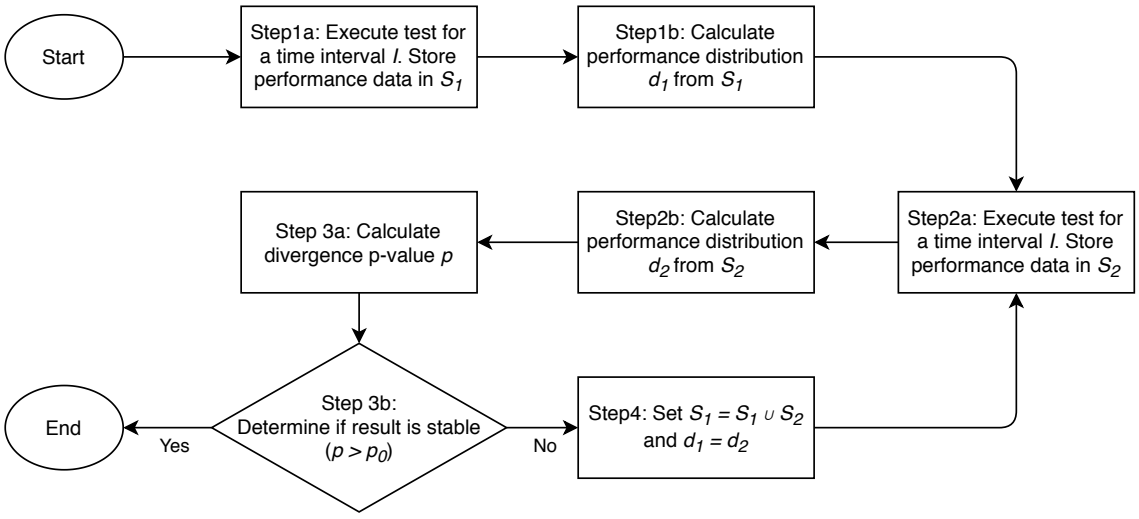


Figure 2.2: Workflow of the PT4Cloud methodology

Figure 2.2 illustrates the methodology workflow. First, the test is executed for a time interval I. The resulting data points are stored in S_1, and the performance distribution d_1 is calculated from S_1. The Gaussian kernel density estimation technique is applied to estimate the probability density function. Their empirical analysis revealed that partitioning the distribution into 1'000 strips is sufficient. More strips produced only slightly more accurate results, but the calculation time also increased. In step two, the test is executed once more, and S_2 and d_2 are computed.

In Step 3, d_1 and d_2 are compared. The Kullback-Leibler divergence between the two distributions is then calculated. The divergence compares two distributions P and Q over a random variable x. The equation for how P diverges from Q is as follows:

$$D_{KL}(P||Q) = \int P(x) \frac{P(x)}{Q(x)} dx \tag{2.4}$$

The divergence $D_{KL}(P||Q)$ produces a non-negative value between 0 and infinity. On one hand, a value of zero means that the distributions P and Q are identical. On the other hand, a value of infinity indicates no similarity between the two distributions. As the interpretation of such a value is non-intuitive for most end-users, the value is transformed into a likelihood with the following formula:

$$L(P||Q) = 2^{-D_{KL}(P||Q)} \tag{2.5}$$

The above likelihood is not symmetric, so we define the symmetric probability p as follows:

$$p = L(P||Q) * L(Q||P) \tag{2.6}$$

If the probability p is above a pre-defined threshold p_0, the performance distribution is deemed stable. They utilized p_0 0.9 as a threshold to decide whether their cloud performance measurements are stable. If the probability p is below the threshold, more test executions are performed until the performance is deemed stable. Before the next test execution is conducted, the operations $S_1 = S_1 \cup S_2$ and $d_1 = d_2$ are performed.

An advantage of this methodology is that non-typical distributions, such as produced by performance tests, can be compared. They presented an experimental evaluation to verify the accuracy of their methodology. They utilized two different public cloud providers with different Virtual Machine (VM) configurations and a time interval length of one week. They also provided a technique to reduce the number of test runs. As a result, they achieved an average accuracy of 95.4%, reducing the execution time by 62%.

2.2 Statistics

The benchmark performance distribution is usually not a normal distribution. The distribution often resembles Figure 2.3, possessing a long tail distribution. Extraordinary events are more likely than in a normal distribution. In the language of performance tests, this means that some data points are extreme outliers (with a factor of 10 or more). So, the analysis must take such extreme events into account. Additionally, a minimum invocation time technically cannot be undercut.

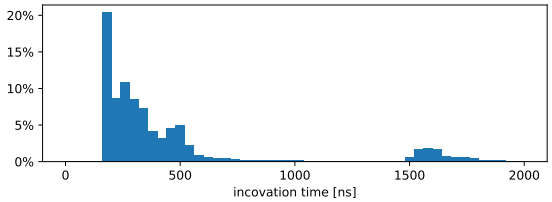


Figure 2.3: Performance distribution of the benchmark *org.apache.logging.log4j.perf.jmh.FileAppenderBenchmark.logbackAsyncFile* from *apache/logging-log4j2*

2.2.1 Coefficient of Variation (CoV)

The CoV, also called relative standard deviation, refers to a descriptive statistic measuring the distribution variability. For a distribution M, the CoV is defined as $CoV(M) = \frac{\mu(M)}{\sigma(M)}$, where $\mu(M)$ is the mean and $\sigma(M)$ the standard deviation. The CoV has already been employed in previous studies to compare performance variation in software engineering [33, 35]. The advantage of this

statistic metric is the unit-less comparison of two distributions [1]. For example, in this work, we utilize the CoV to compare the variability of benchmarks possessing various execution times. The variability of a benchmarks with an average invocation time of one millisecond can be compared to a benchmark of 100-nanoseconds invocation time. In this work, we do not interpret the value as a percentage, because the value can be larger than one, which is non-intuitive.

2.2.2 Confidence Interval (CI)

Another variability metric comprises the width of a confidence interval as a percentage of the mean. Georges et al. found that the width depends highly on the task performed by the benchmark [20]. In their experiments, the width converged between 1% and 3% using a 95% confidence level. As explained above, the performance data is not normally distributed. As such, we cannot simply calculate the confidence interval with the mean and standard deviation. We instead apply bootstrapping with hierarchical random resampling and replacement, as proposed by Kalibera and Jones for rigorous performance measurement of benchmarks [28, 29]. The hierarchical resampling is performed on three levels: trial-level, iteration-level and invocation-level. Previous literature suggests using 10'000 bootstrap simulations, as the Monte Carlo variation is reduced with more samples [24]. However, this presents a trade-off between computation time and the accuracy of the confidence interval.

2.2.3 Sequential Testing

Traditionally, an experiment is performed, after which the statistical test is computed. However, sequential testing offers an alternative approach [55]. During the sequential hypothesis testing, a stoppage rule is applied, enabling three decisions: accept the hypothesis, reject the null hypothesis, or continue the experiment and reapply the rule later. Such an approach can decrease the experiment's sample size, resulting in faster execution. Frick first produced a minimum number of measurements, and afterwards regularly checked whether the p-value was smaller than 0.01 to reject the null hypothesis or if the p-value was larger than 0.36 to accept the hypothesis [19]. In their experiments, such a stoppage rule was 30% more efficient than the traditional approach.

2.2.4 A/A Testing

The literature on performance evaluation suggests two A/A tests to evaluate whether a performance change between the test and the control group is significant. The first involves the overlapping confidence interval of the mean [20, 29]. The two groups are statistically different if no overlap exists for the confidence interval. This approach is similar to the computation of the mean confidence interval explained previously.

The second approach comprises a Wilcoxon rank-sum test (also called Mann-Whitney U test) combined with an effect-size analysis [8]. The null hypothesis is that the test and control groups feature the same performance. The tests produce a p-value that is compared with the confidence level to decide whether or not the null hypothesis is rejected. However, for large samples, a small change can already be statistically significant. Literature further suggests reporting the effect size [16]. We compute the Cliff's Delta effect size [10]. Only if both the hypothesis test and effect size test conclude that a significant change occurred do we argue that a significant performance change happened.

Chapter 3

Study Design

In this chapter, this study's research questions are presented first. Afterward, a study overview is provided and the project selection described.

3.1 Research Questions

Previous work has already investigated the popularity of different performance testing frameworks for Java [52] and bad practices of JMH [13]. The performance testing literature has also mentioned that executing microbenchmarks is time-consuming. A fine granular analysis is necessary regarding the developer preferences as to how they configure software microbenchmarks. A developer possesses different configuration parameters, which all influence the execution time. In the first part of this thesis, we seek to more deeply understand how developers configure benchmarks and the execution time this produces. To do so, the following research question is defined:

RQ1: How are software microbenchmarks configured and what are their resulting execution times?

As this first question contains various aspects, we further distinguished different sub-questions. First, we are interested in how often developers set user-defined values and not simply use the default configuration. This raises the question of whether some configuration options are modified more often than others. Second, we analyze, on a configuration parameter level, which user-defined values are commonly employed. On one hand, we investigate whether configuration parameters are modified in isolation. On the other hand, we explore whether projects with more stars or contributors configure software microbenchmarks differently.

RQ1.1: Which custom software microbenchmark configurations are defined by developers?

With the chosen configuration, we can compute the execution time of an individual benchmark. We seek to analyze how the chosen user-defined configuration influences an individual benchmark's execution time. This raises a question of how often a modified configuration ends in a shorter or longer execution time compared to the default execution time. Furthermore, we compare if and how the warmup measurement proportion is changed, including whether developers consider one of the two phases to be more important.

RQ1.2: How do custom configurations affect benchmark execution time?

The number of benchmarks and parameterization combinations, plus the execution time of an individual benchmark, influences the project test suite execution time. In turn, this produces the following question:

RQ1.3: How long does it take to run the full benchmark suites of open-source projects?

Lastly, we are interested in how often a developer modifies the execution configuration of a benchmark after its creation. Additionally, we focus on the default value update and analyze how developers deal with the new standard configuration.

RQ1.4: How often are execution configurations modified?

In the second part of this work, we seek to reduce the execution time of the software microbenchmarks without sacrificing result quality. A dynamic reconfigured execution produces a high result quality if the characteristics of the resulting distribution are not significantly different from the standard execution result. We wish to apply sequential testing on microbenchmarks to dynamically decide whether to stop the execution. Several studies have already applied stoppage rules on load tests to reduce execution time [3, 23, 36]. However, none of them focused on short-running performance tests. Currently, in JMH, the warmup phase possesses a fixed size. We wish to reduce the execution time by stopping the warmup phase if a steady state is achieved. Additionally, we seek to investigate whether the measurement phase in JMH is too long and can be reduced. To this end, we compare three different stoppage criteria — CoV, CI width and divergence — against the standard execution. If such a dynamic reconfiguration of software microbenchmarks is faster, but still produces sound results, there are fewer reasons to set user-defined execution configurations. This results in the second research question:

RQ2: Can dynamic reconfiguration of software microbenchmarks reduce the execution time without sacrificing result quality?

The execution of a software microbenchmark is divided into two phases: warmup and measurement. During the warmup phase, a benchmark should reach a steady state that later a high-quality result is produced. The performance during the measurement phase is reported and later evaluated. With the default JMH configuration, the warmup and measurement phase requires the same amount of time. The question now becomes how long should the benchmark performance be captured during the measurement phase under the assumption that, during the warmup phase, a steady state is reached.

RQ2.1: How much does the length of the measurement phase matter after a steady state is reached?

Next, we analyze if and how the performance result is affected using reconfiguration compared to the standard execution. Another result characteristic is produced if, for example, the warmup phase is stopped too early.

RQ2.2: How does reconfiguration of software microbenchmarks affect the execution result?

Lastly, we seek to compare the execution time of a benchmark with reconfigured execution configuration against the standard approach, including how much time is saved and how large is the performance overhead with sequential testing. Additionally, we compare the results between the different project test suites.

RQ2.3: How much time can be saved with dynamic reconfiguration of software microbenchmarks?

3.2 Study Subjects

We created a new dataset to analyze the state of practice execution configurations. Figure 3.1 summarizes how the dataset was generated. As baseline, we used two preexisting datasets. The first dataset was previously used by Costa et al. to analyze bad practices in JMH [13]. The last Google BigQuery[1] snapshot from 2017 queried what GitHub projects with a JMH benchmark are extracted. A project possesses a benchmark if `org.openjdk.jmh.annotations.Benchmark` is imported and at least one method is annotated with `@Benchmark`. From this data source, 839 repositories were extracted.

For the second existing dataset, Java projects on GitHub were extracted if they employed a performance testing framework, such as JMH or Caliper[2]. We reused only the 1'204 projects where JMH benchmarks were found. Third, we mined Maven Central, which featured a list of 617 artifacts using JMH as dependency[3]. First, we crawled the artifact name from the website. Second, we checked whether the pom file of the current version possessed a `scm` element. This source code management element contained the git clone Uniform Resource Locator (URL). If the URL pointed to GitHub, we used the repository in our third dataset. For 128 artifacts, we found a valid source code repository on GitHub.

In the next step, we combined the three datasets. The GitHub repository name (e.g., *apache/logging-log4j2*) is a unique name. As such, we could remove duplicates if a project was present in multiple datasets. As an intermediate result, we obtained a list of 1'477 projects. Some repositories in our dataset are forks. If the parent repository name was present in the GitHub Application Programming Interface (API), we checked whether the parent repository was already part of our dataset. In 69 cases, we added the parent repository to our combined dataset. As the first dataset was from 2017, not all repositories were still available. Such projects were either removed or were no longer public available. If we called the GitHub Representational State Transfer (REST) API, and the response of the repository endpoint was a 404 error, we removed the project from the dataset. Accordingly, 140 projects were removed from the dataset. Some projects were moved to another user or organization or were renamed. However, if multiple repositories possessed the same git clone URL in the GitHub API, we could remove all repositories that simply pointed to another repository. From this, 32 repositories of the dataset just pointed to other repositories on GitHub. Lastly, we removed 13 projects lacking at least one benchmark in the current commit. Often, they still had JMH as dependency.

Forked projects bias the analysis. For example, our dataset features 40 forks of the project *netty/netty*. We grouped projects by the root repository name. For each group, we must select only one repository. However, simply choosing the root project itself is not the best solution because, for example, the *jgrapht* project is no longer developed in the root repository *lingeringsocket/jgrapht*; rather, the active development is performed in the fork *jgrapht/jgrapht*. We decided to choose per group the project with the highest star count. Finally, we obtained a cleaned dataset of 753 projects.

For each repository, we saved the following repository metrics: number of forks, stars, watchers, contributors and commits. On this bases, we can investigate differences between projects using correlation analyses. Additionally, the git clone URL, the last commit hash plus the default branch name were saved, as they are required for the repository mining.

[1] `https://cloud.google.com/bigquery`
[2] `https://github.com/google/caliper`
[3] `https://mvnrepository.com/artifact/org.openjdk.jmh/jmh-core/usages`

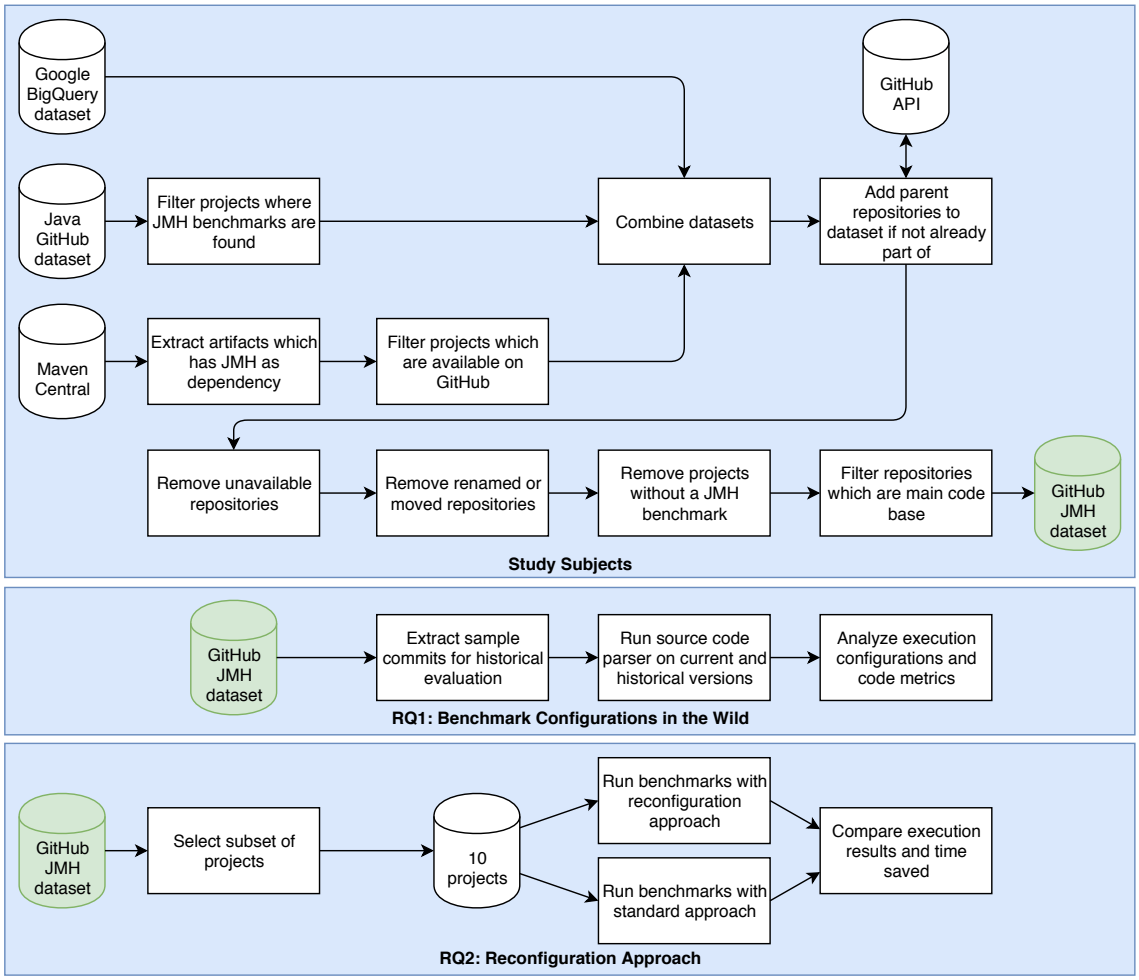


Figure 3.1: Study overview

3.3 Study Overview

This section provides a study overview. Figure 3.1 summarizes the corresponding steps. The dataset creation was described in the previous section. For the first research question, we determine a set of commits upon which the historical analysis is based. The source code parser is run on the selected code versions. We answered the first research question based on the extracted execution configurations and code metrics, such as the method body length. For the second part of this work, we selected a subset of projects, listed in Table 5.2. First, all benchmarks of these projects were executed with the standard and reconfiguration approach. Later, we compared the different stoppage criteria against the standard execution in terms of result quality and the time saved through this dynamic execution.

Chapter 4

Benchmark Configurations in the Wild

In this chapter, the data collection is first described. Following this, some background properties of the chosen projects are provided. Lastly, the results of the first research question are presented.

4.1 Approach

First, we explain how data is extracted for the first part of this study, including which tools and heuristics are employed to obtain the information. Afterward, the historical analysis is explained.

4.1.1 Extracting Data

We cloned all repositories on 22 August 2019. For the first part of this work, we performed numerous static code analyses. To this end, we utilized the Bencher tool[1] to extract several metrics and information from the source code. First, we extracted the benchmark name and class name. Next, we extracted the configuration of each benchmark and where the value was set (on a class or method). We further checked whether the annotations `@Fork`, `@Warmup`, `@Measurement` and `@BenchmarkMode` were present, such as in Listing 2.1, and we read the property values. Afterward, the Bencher checked which classes were annotated with `@State`. For these state objects, the corresponding instance variables, annotated with `@Param`, were extracted and stored. The Cartesian product could be calculated to obtain the number of parameterization combinations. Furthermore, the return value and method arguments of each annotated `@Benchmark` method are available. With this information, we can check whether a `Blackhole` or a state object is used for the benchmark. A set of benchmarks can be grouped with the annotation `@Group` and a group name that they are always executed together. The Bencher extracts the method body hash, with which we checked whether a method implementation was changed. Lastly, it provides the number of lines per benchmark method. The source code must not always be syntactically correct, so the parser may fail. However, while a few warnings occurred during parsing, the parser never crashed. We created a Comma Separated Values (CSV) file where, for each benchmark, information such as the configuration, feature usage and method hash are stored.

The Bencher project already provided a rich set of features to extract JMH information from Java byte code. However, source code parsing was not supported. First, we implemented the source code parsing with Eclipse JDT[2] and provided the same API as the byte code parser. In a

[1] `https://github.com/chrstphlbr/bencher`
[2] `https://www.eclipse.org/jdt`

second step, we added several new features to both parsers. If a benchmark uses state objects as method argument, the parser now checks whether JMH parameters are also defined in such annotated classes. The `BenchFinder` interface supports two new functionalities: On one hand, it returns a map of state objects with their defined JMH parameters. On the other hand, for a benchmark, it returns a map of JMH parameters and the class in which they are defined.

We assume that used state objects are defined in the same project and not in an external dependency that the fully qualified name resolution does not require any dependency as source or byte code. Parsing larger projects such as *apache/logging-log4j2* takes approximately 15 seconds.

JMH Version Extraction

In several analyses, the JMH version is required, because if no execution configuration is present, the default values of the corresponding JMH version are employed. To extract the used JMH version, we applied some simple heuristics. We traversed all files in the project directory, and if we found a file named `pom.xml`, we applied the Maven rules. First, the heuristic checks every single line for whether it contains both keywords *jmh* and *version*, because many projects use a property variable named, for instance, *jmh.version* to define the JMH version. If both keywords are found on the same line, it is checked for whether, on the same line, a valid version is found. The version extractor starts the check with the newest version 1.21[3] and checks all valid versions until it ends by 0.1. The first found version is returned. Searching in descending order is important, because otherwise, wrong versions are extracted. For example, if a maven project possesses a file with the line `<jmh.version>1.17.2</jmh.version>`, and the extractor first checks 1.17 before checking 1.17.2, the heuristic will not work properly. If the single line check does not reveal a JMH version, the extractor checks the dependencies block. For each dependency item, it checks whether the *groupId* is equal `org.openjdk.jmh` and the *artifactId* is equal `jmh-core`, after which the version is extracted with the same algorithm as above.

If we found a file with the extension `gradle`, we applied the Gradle heuristics. First, it checks whether a single line contains `jmhVersion` and a valid version number. This rules extracts the JMH version set in the *me.champeau.gradle.jmh* Gradle plugin[4]. As the second rule, each line is checked for whether the JMH dependency is defined. A line contains the dependency if the string `org.openjdk.jmh` was found plus a valid version number.

Java Version Extraction

The Java version extraction works similarly to the JMH version extraction. The source and target versions are individually extracted. However, the only difference is that once the keyword *target* and once *source* is used. We traversed all files in the project, and if we found a file named `pom.xml`, we applied the Maven rules. First, the extractor checked whether the `maven-compiler-plugin` contained the Java version as configuration property (e.g. `<source>1.8</source>`). Second, the properties were checked for whether they contain `<maven.compiler.source>1.8</maven.compiler.source>`. If we found a file with the extension `gradle`, we applied the Gradle heuristic. The heuristic simply checks if, after the keyword `sourceCompatibility` (respectively `targetCompatibility`), a valid Java version exists. After extracting the Java version, we unified the output. For example, `8`, `1.8` and `JavaVersion.VERSION_1_8` all describe values used to specify Java 8. We mapped all values to the same value 1.8. Later, we were only interested in the Java target version. However, if only the source code version was present, the target version was set to the same value [40].

[3] JMH version 1.22 was released after we cloned the repositories so if we speak from the current version we mean JMH version 1.21

[4] `https://plugins.gradle.org/plugin/me.champeau.gradle.jmh`

Commented out Benchmarks

As we also want to analyze whether a `@Benchmark` annotation is commented out, we created a simple heuristic. There is no option to directly obtain the content of the comment from the Eclipse abstract syntax tree. From the syntax tree parser, we receive only the line number where comments exist. With the corresponding file we can read the content of the comment. We checked line by line whether it contains `@Benchmark`. If the annotation was found, the heuristic that extracts the method name is applied. The heuristic searches the next `public` keyword in the source code. This line contains the method name before the next opening parenthesis. A method annotated with `@Benchmark` must be public, because otherwise, this produces an error when packaging the jar file. Listing 4.1 offers an example. On the second line, the corresponding `@Benchmark` annotation is found. The next line features no `public` keyword. On line four, the `public` keyword is found. The last word before the opening parenthesis, `latencySimple`, is the method name. This heuristic fails in a few cases. For example, if `@Benchmark` annotation is used in a regular comment, as in the Hello World example in line 44 from JMH[5], the heuristic thinks that a valid benchmark annotation is found.

```
1 // @BenchmarkMode(Mode.SampleTime)
2 // @Benchmark
3 // @OutputTimeUnit(TimeUnit.NANOSECONDS)
4 // public void latencySimple() {
5 //   logger.info(TEST);
6 // }
```

Listing 4.1: Commented out benchmark in *apache/logging-log4j2*

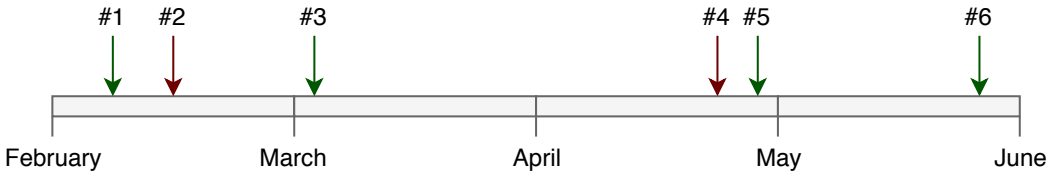


Figure 4.1: Illustrative example on how sample commits are selected

4.1.2 Historical Evaluation

For such a large dataset, we cannot evaluate all commits in the history, because this would be too time-consuming. We instead created an algorithm to reduce the number of commits. Only commits on the default branch are considered. Figure 4.1 presents an illustrative example of how sample commits are selected. The first and last commit are always selected — here, commit #1 and #6. Afterward, the algorithm traverses from the last commit to the first commit, and checks at the first day of each month whether a sample commit can be determined. The closest commit to this day is always chosen as the sample commit. Additionally, the chosen commit must be at most half a month away. For the first of May, commit #5 is selected. Commit #4 is the nearest commit to the first of April, but it is not selected because it is more than half a month away. So, for the first of April, no sample commit is selected. Lastly, for the first of March, commit #3 was chosen, because it is nearer than #2. Finally, four sample commits for the historical analysis are

[5] `https://hg.openjdk.java.net/code-tools/jmh/file/99d7b73cf1e3/jmh-samples/src/main/java/org/openjdk/jmh/samples/JMHSample_01_HelloWorld.java`

chosen. As an additional constraint, the algorithm does not traverse commits written before June 2014, because JMH then added the `@Benchmark` annotation, which is required by the parser.

On one hand, for large projects such as *elastic/elasticsearch*, the number of commits is reduced from 47'659 to 64 commits. On the other hand, analyzing only sample commits makes the evaluation more complex, because we only possess two snapshots and do not know what happens in between. The historical dataset is only used to make rough statements, such as how often a benchmark was written and the configuration was never updated. For our needs, it was sufficient if we knew that a change occurred, and not exactly when.

As previously mentioned, we are interested in the evolution of the benchmark methods. For the configuration, we can simply compare whether or not it is equal to the previous one. For the method body, several change types are possible.

- The fully qualified name of the benchmark is present for the first time in the project. So a new benchmark was added.
- An existing benchmark was updated because the method body hash was changed.
- An existing benchmark was removed.

For a simpler interaction with the dataset, we added an extra change type, because sometimes, a benchmark was temporarily removed and later visible again (e.g. a benchmark is commented out temporarily). With the change type history, we can compute how often a benchmark implementation was changed.

4.2 Dataset Description

The dataset contains 753 projects with 13'387 benchmarks and 48'107 parameterization combinations. Figure 4.2 presents the distribution of how many benchmarks the dataset projects possess. A total of 400 (53.1%) projects feature less than 10 benchmarks, and 52 (6.9%) projects possess 50 or more benchmarks. So, most of projects feature only a few written benchmarks, but there are some large projects, such as *eclipse/eclipse-collections* which has the most benchmarks with 515. A project possesses, on average, 18.9$\pm$40.8, and a median of seven benchmarks. Figure 4.3 presents the number of parameterization combinations per project. On average, a project possesses 70.6$\pm$303.3 and a median of nine combinations. The number of projects possessing more than 50 benchmarks is small. However, far more projects feature over 50 parameterization combinations. The highest number of combinations belongs to the project *msteindorfer/criterion*, with 4'132 combinations.

Table 4.1 summarizes the feature usage in the mined projects. A total of 43 (6.3%) projects feature at least one defined group where multiple benchmarks are executed together. Overall, 530 (4.0%) found benchmarks possess such a `@Group` annotation. Meanwhile, 3'529 (26.3%) of the benchmarks feature a method argument where the class is a state object. A total of 20 benchmarks set as the function argument their own class containing JMH parameters[6]. They wish to use the class where the benchmark is defined as a state object. Technically, this works, but it is not needed, as the benchmark always uses the class where it is defined as a state object if the class is annotated with `@State`. Over all projects 5'319 JMH parameters are extracted. Most of these JMH parameters (91.3%) are defined in the class belonging to the method. A total of 10'394 (77.6%) benchmarks do not use the JMH parameterization feature. If a state object features two JMH parameters, `myString1` and `myString2`, a developer can set an array of values for each.

[6] `https://github.com/devexperts/dlcheck/blob/master/benchmarks/src/main/java/com/devexperts/dlcheck/benchmarks/FineGrainedLockBenchmark.java#L96`

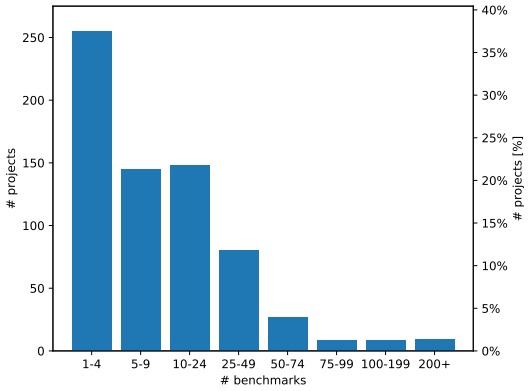


Figure 4.2: Number of benchmarks per project (100% are all 753 projects)

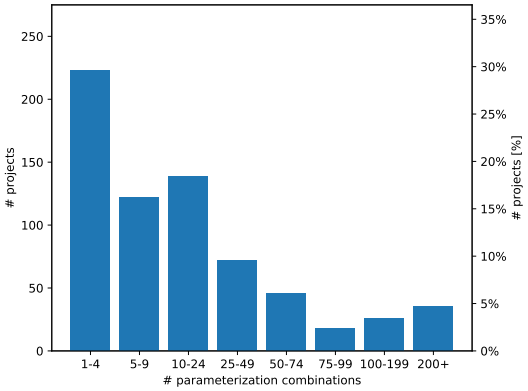


Figure 4.3: Number of parameterization combinations per project (100% are all 753 projects)

Description	Metric
Benchmark is part of a group	530 (4.0%)
Method argument of the benchmark is a state object	3'529 (26.3%)
JMH parameters are defined in the class which belongs to the method	4'855 (91.3%)
Parameterized benchmark with exact one parameterization combination	538 (4.0%)
Parameterized benchmark with multiple parameterization combinations	2'455 (18.3%)
Benchmark has a non-void return type	7'154 (53.4%)
Benchmark uses `Blackhole`	3'062 (22.9%)
Benchmark has a non-void return type or uses `Blackhole`	9'682 (72.3%)
Benchmark uses `Control`	78 (0.6%)
Benchmark uses `BenchmarkParams`	2 (0.0%)
Benchmark uses `IterationParams`	0 (0.0%)
Benchmark uses `ThreadParams`	1 (0.0%)
Benchmark is defined in inner class	156 (1.2%)
Average number of benchmarks per class	4.2±6.0
Average number of benchmarks per file	4.3±6.1

Table 4.1: Feature usage metrics (100% are all 13'387 benchmarks)

For all found JMH parameters, the Cartesian product is built. If, for both JMH parameters, exactly one value is set, only one parameterization combination is possible. A total of 538 (4.0%) benchmarks use JMH parameters, but have exactly one parameterization combination. If, for both JMH parameters, two values are set, four combinations are possible. Meanwhile, 2'455 (18.3%) benchmarks possess more than one parameterization combination.

Figure 4.4 illustrates the number of combinations if more than one parameterization combination exists. A total of 76.9% of the benchmarks feature 10 parameterization combinations or fewer. The maximum number of combinations belongs to a benchmark of *apache/hive* with 2'304[7]. Each

[7] `https://github.com/apache/hive/tree/master/itests/hive-jmh/src/main/java/org/apache/hive/benchmark/vectorization/operators/VectorGroupByOperatorBench.java`

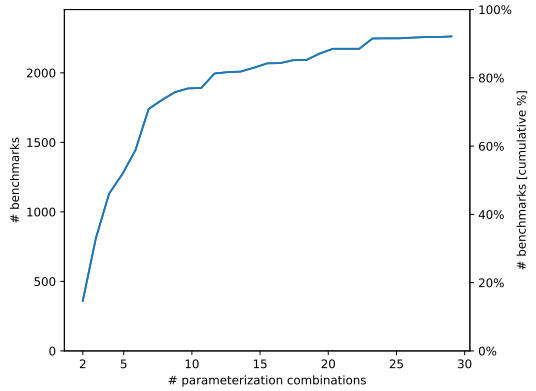


Figure 4.4: Number of parameterization combinations (100% are 2'455 benchmarks which have multiple parameterization combinations)

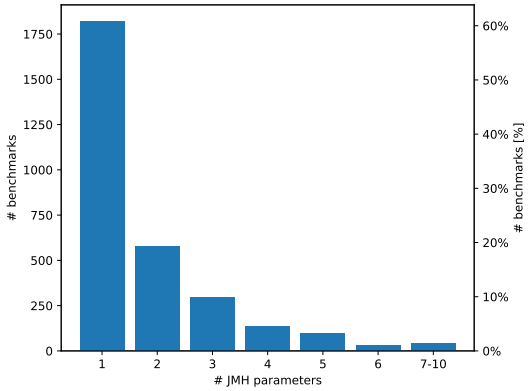


Figure 4.5: Number of JMH parameters per benchmark (100% are 2'993 benchmarks which have at least one JMH parameter)

parameterization combination is executed as an individual benchmark. As such, most projects limit the number of combinations, because otherwise, the execution takes too long. Figure 4.5 presents the distribution of how many JMH parameters a benchmark has combined over all used state objects. Over 60% of the parameterized benchmarks possess exactly one JMH parameter (which can have multiple values).

To avoid dead code elimination, a benchmark should either use a `Blackhole`, which consumes intermediate results, or the computed result should be returned by the benchmark method. 53.4% of the benchmarks feature a non-void return type, and 22.9% of the benchmarks use a `Blackhole` provided by JMH as the method argument. In total, 72.3% of the benchmarks use one option (or both) to avoid JVM optimization. We performed no-call graph analysis to check whether all intermediate results were correctly consumed. We also analyzed the usage of other classes in JMH's `infra` package. The control class is used in 78 (0.6%) cases. The three other classes — `BenchmarkParams`, `IterationParams` and `ThreadParams` — are all used less than three times.

On average, 4.2 benchmarks are defined per class. JMH enables defining benchmarks in an inner class. As such, 156 (1.2%) benchmarks are defined in a nested class. This produces an average of 4.3 benchmarks per file. As an extreme case, the project *nickman/json-benchmark* defines all its 32 benchmarks in a single file[8].

Figure 4.6 presents the length of the benchmark method bodies, where 54.3% of the benchmarks contain only a single line. As such, the median is one line. On average, a method annotated with `@Benchmark` features a length of 3.2±4.5 lines. Over 12'659 (94.6%) of the benchmarks possess less than 10 lines of code. The longest method body is 160 lines long[9]. We do not extract the method body length from the source code file. The Eclipse abstract syntax tree parser provides the method body source code. However, they use the syntax tree and always place one statement on each line. As the lines of code offer a good predictor for McCabe's cyclomatic complexity [26], we can conclude that the cyclomatic complexity of most benchmarks is low and most logic is outsourced to other methods.

[8] `https://github.com/nickman/json-benchmark/blob/master/src/test/java/test/com/heliosapm/benchmarks/TestBenchmark.java`

[9] `https://github.com/gameduell/eclipselink.runtime/blob/master/jsonb/src/test/java/org/eclipse/persistence/json/bind/defaultmapping/performance/PerformanceTest.java`

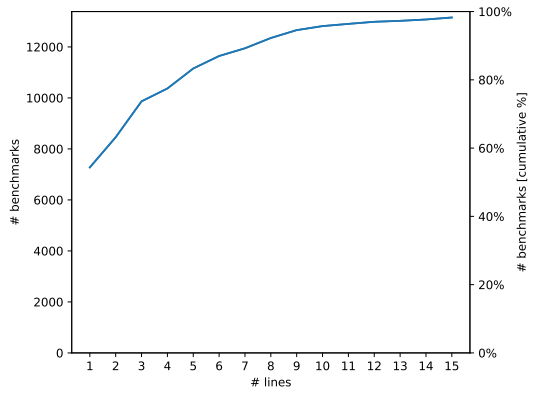


Figure 4.6: Lines of code per benchmark method body (100% are all 13'387 benchmarks)

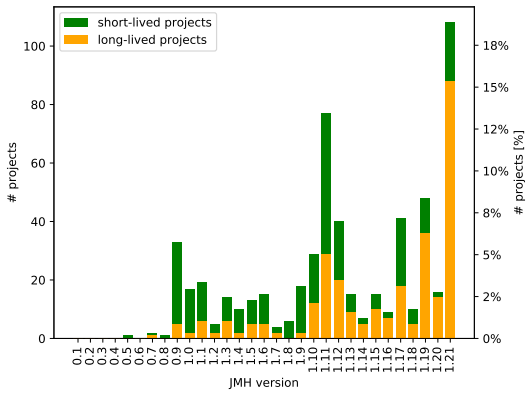


Figure 4.7: Distribution of used JMH version (100% are 573 projects with an extracted version)

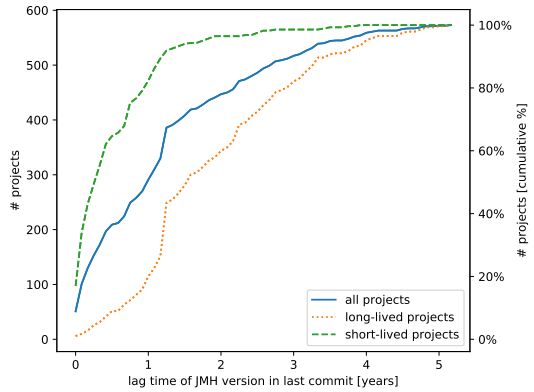


Figure 4.8: Age of the used JMH version in the last commit (100% are 573 projects with an extracted version)

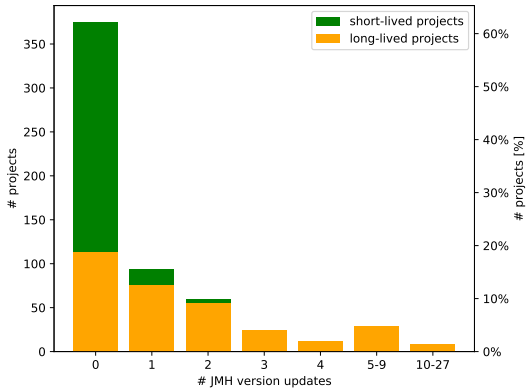


Figure 4.9: JMH version update frequency (100% are 573 projects with an extracted version)

For 180 (23.9%) projects, we cannot extract a JMH version with the heuristic and ignore them in the following analysis. Figure 4.7 presents the usage share of the JMH versions. The first public release occurred in November 2013, and the last in May 2018. Patch versions such as 1.17.1 were combined with the corresponding minor version 1.17 to simply the visualization.

The first small peak comes from version 0.9, so before the beta phase ended in August 2014. The second peak comes from version 1.11, released in September 2015. Until May 2017, and the release of version 1.19, there was normally a monthly non-patch release. However, version 1.11 was released in September 2015, and the next minor release, 1.12, occurred in April 2016. As such, the high occurrence of 1.11 is due to this long period of time before the next non-patch release. The most commonly used version is the current one. The projects are categorized into two groups: If the time between the last commit and the first (sampled) commit where JMH is used is over a year, it is classified as a long-lived project, and a short-lived project otherwise. For older versions, the short-lived projects are outnumbered, as many of the short-lived projects are written once, and then no longer actively developed (e.g., demo projects that show how to use JMH).

Some projects are no longer actively developed. As such, we compared the time of the last commit against the release date of the used JMH version. Figure 4.8 illustrates the lag time of the used version in the last commit. Few long-lived projects possess a JMH version younger than one year. The strong increase of 1.25 years for long-lived projects is exactly the distance between the current version's release date and the mining of the repositories for this study in August 2019. Previous work has revealed that numerous developers do not regularly update the used third-party dependencies in projects [11, 31]. Over 40% of the long-lived projects employ a JMH version older than two years. We can conclude that the JMH dependency is not an exception of rare updated dependencies.

Figure 4.9 demonstrates how often the JMH dependency was updated. Most of the short-lived projects are never updated. However, a small group of long-lived projects is often updated. On one hand, the repo *glowroot/glowroot* performed 27 JMH dependency updates. On the other hand, over 60% of the projects never conducted a version change. Long-lived projects updated, on average, 1.9±2.9 times, with a median of one time.

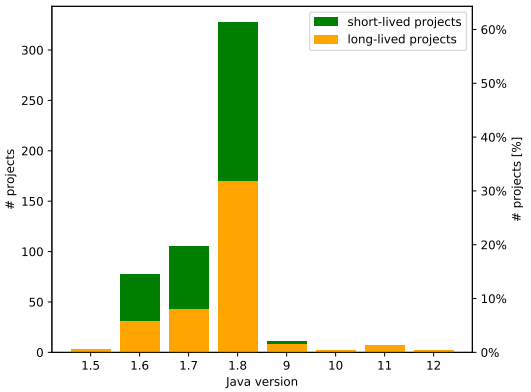


Figure 4.10: Distribution of used Java version (100% are 534 projects with an extracted version)

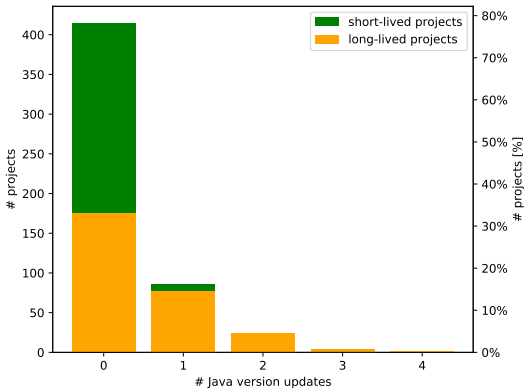


Figure 4.11: Java version update frequency since JMH is used (100% are 534 projects with an extracted version)

Figure 4.10 illustrates the used Java target version. For 219 (29.1%) projects, neither the source nor the target version can be extracted with the heuristic. No large difference exists between the two groups, short- or long-lived. Only 22 (4.1%) of the projects use a Java target version newer than 1.8. At 61.2%, version 1.8 is by far the most popular. Some projects use version 1.5 or 1.6, where the extended support has already ended in April 2015 or December 2018, respectively [41]. Figure 4.11 demonstrates how often a project updated the Java version. Short-lived projects rarely update the Java version. We can explain this by the maximum period of one year where the project was maintained. For 175 (62.3%) of the long-lived projects, the Java version was also never updated.

The Spearman correlation between the repository metrics and feature usage in Table 4.2 illustrates that no strong correlation exists anywhere. The feature metric is a percentage. As an example, if a project possesses 20 benchmarks, 15 of which possess a `@Group` annotation, the group metric would be 0.75. We only analyze commonly used features, because otherwise, for example, the `Control` class correlation value would only depend on a few projects. Projects possessing more stars, forks and so on tend to write a few more benchmarks and more often use blackholes. However, we can refute the hypothesis that projects with more contributors include someone who is an expert in performance engineering, knows how to write good software microbenchmarks, and uses the full range of the provided JMH features.

	Stars	Forks	Watchers	Commits	Contributors
Benchmark has parameterization combinations	0.06	0.07	0.07	-0.03	0.01
Benchmark is part of a group	0.05	0.10*	0.05	0.05	0.05
Benchmark uses `Blackhole`	0.10*	0.10*	0.12**	0.10*	0.13**
Benchmark has a non-void return type	-0.01	-0.04	-0.00	-0.02	-0.02
Benchmark has a non-void return type or uses `Blackhole`	-0.05	-0.08*	-0.03	-0.07	-0.05
Number of benchmarks	0.12**	0.11**	0.10**	0.17**	0.14**

Table 4.2: Spearman correlation analysis between the repository metrics as independent variables and the feature metrics as dependent variables (correlation significant at 0.05 level *, 0.01 level **)

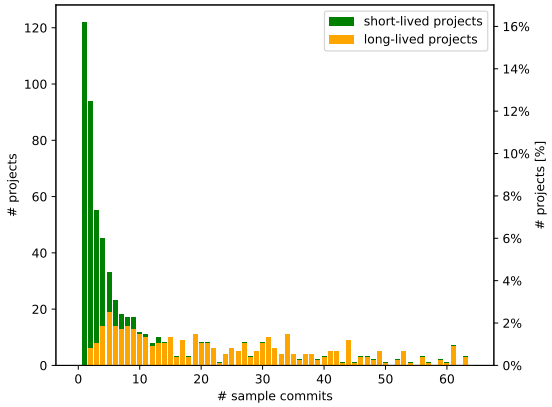


Figure 4.12: Number of chosen sample commits per project (100% are all 753 projects)

Figure 4.12 illustrates the distribution of how many commits were chosen during selecting sample commits. We count only the number of sample commits from the time when a benchmark was detected for the first time. A short-lived project can possess a maximum of 13 commits, including the first commit, last commit and one commit per month. Projects with exactly one commit are often demo projects, and not production code. Short-lived projects possess, on average, 2.6±2.0 and a median of two sample commits. For long-lived projects, we see a range of 2–63 sample commits. Some long-lived projects do not commit monthly, but exist longer than a year. On the other hand, 26 projects possess at least 50 sample commits. They committed almost every month and used JMH almost since the release of version 1.0. The long-lived projects possess, on average, 22.7±16.2 and a median of 19 sample commits.

During the static code analysis of the projects, we found 287 commented-out benchmarks in the current commit, but somewhere in the past, we tracked the benchmark as not part of a comment. For most of the benchmarks, we did not find any hints in the code as to why the benchmark was commented out. If we compare this number to the total number of benchmarks found, these equal 2.1% of all benchmarks. It could be that the benchmark failed and no one knew how to fix the issue, so it was simply commented out.

Until now, we simplified the explanation of how a developer must set the measurement and warmup time. He cannot just set a simple value in seconds. Time features two properties: On one hand, the time unit comprises an enum from the type `java.util.concurrent.TimeUnit`, and on the other hand, it comprises an integer value in the corresponding unit. Normally, a developer simply sets both properties. However, 987 benchmarks only change the integer value and not the time unit. This approach is dangerous if JMH changes the default time unit. Much worse, however, is if only the time unit is set. We found 95 benchmarks where the warmup and/or measurement time was not properly set. As an example, a developer sets the time unit to milliseconds and assumes that the execution will take 10 milliseconds per iteration, as the default integer value is 10 in version 1.21. However, if only the time unit was changed, the set time unit is ignored, and the time per iteration remains 10 seconds. Someone who reads the execution configuration will not know the effective execution configuration.

4.3 Results

The analysis of how JMH benchmarks are currently configured is divided into multiple subquestions. First, we wish to understand how often a developer sets a user-defined value and does not use the default values. Furthermore, if user-defined values are set, we seek to determine what values are chosen and whether we can identify a trend of popular values. Second, we want to analyze the execution time of an individual benchmark, including how the chosen configurations affect the execution time and whether the proportion between warmup and measurement time changed. Additionally, we investigate the project test suite execution time. Lastly, we focus on how often execution configurations are modified. As a special case, we study how developers react after the JMH default values were changed in version 1.21.

4.3.1 RQ1.1: Custom Execution Configurations

A total of 3’281 (24.5%) benchmarks possessed neither a class nor a method annotation for all seven configuration options. These benchmarks are executed exactly with the default parameters. Table 4.3 summarizes how often an annotation is used by developers. It does not matter if a class or method is annotated. Warmup iteration, measurement iteration and fork annotations are present in just under half of the cases. Warmup time and measurement time annotations are present in just over a quarter of the benchmarks, and the mode is set in a third of the cases. However, warmup forks are set in only 180 (1.3%) of the benchmarks. Therefore, most of the benchmarks use the default configuration and perform no warmup forks.

Configuration parameter	# Annotation present
Warmup iterations	6’002 (44.8%)
Warmup time	2’880 (21.5%)
Measurement iterations	5’962 (44.5%)
Measurement time	3’704 (27.7%)
Forks	5’618 (42.0%)
Warmup forks	180 (1.3%)
Mode	8’894 (66.4%)

Table 4.3: Annotation presence in the source code (100% are all 13’387 benchmarks)

The presence of an annotation does not exclude the case wherein the default value is set as a user-defined value. For this analysis, the JMH version of the project is required, because otherwise, we cannot compare the set value to the default one. For 10'816 (80.8%) benchmarks, we can extract a JMH version. In 4'115 (38.0%) of the cases, the benchmarks already use the newest JMH version 1.21 with the new default execution configurations. Table 4.4 illustrates how often, in the current commit, the set value is equal to the default value of the employed JMH version.

For the warmup time, measurement time and number of forks, the default values are rarely set as user-defined values. For the warmup and measurement iterations, behavior differs. In earlier versions, the default value is rarely set as a user-defined value by a developer. However, after the default execution configuration update, where the default value was changed from 20 to 5, over half of the benchmarks now set the default value as the user-defined value. We suppose that the number of iterations was already set to five before the project was updated to version 1.21, and the annotations that are no longer required were not removed. The mode is equal to the default mode in approximately one third of the benchmarks where an annotation is present. For the number of warmup forks, interpretation is difficult, as few benchmarks possess such an annotation. An analysis of the used value is more interesting.

Configuration parameter	JMH version 1.20 or earlier		JMH version 1.21	
	# Annotation present	Is default value	# Annotation present	Is default value
Warmup iterations	2'422	60 (2.5%)	2'108	1'218 (57.8%)
Warmup time	1'133	0 (0.0%)	899	21 (2.3%)
Measurement iterations	2'473	238 (9.6%)	2'111	1'181 (56.0%)
Measurement time	1'420	0 (0.0%)	1'246	25 (2.0%)
Forks	2'391	26 (1.1%)	1'900	7 (0.4%)
Warmup forks	150	42 (28.0%)	28	23 (82.1%)
Mode	3'971	1'317 (33.2%)	3'144	1'288 (41.0%)

Table 4.4: Frequency that user-defined values are equal to the default value of JMH

180 benchmarks specify the number of warmup forks. Figure 4.13 illustrates which values are chosen for the warmup fork configuration parameter. In 65 (36.1%) cases, the default value of zero is explicitly set as a user-defined value. A total of 115 benchmarks perform warmup forks. In 113 benchmarks, exactly one warmup fork is performed. The remaining two benchmarks are executed with two warmup forks. In the discussion, we analyze which type of projects employ warmup forks.

5'618 benchmarks specify the number of forks. Figure 4.14 presents the distribution of the fork values. A total of 56 (1.0%) benchmarks set a fork value of zero. This means that forking is disabled. Multiple trials are executed in the same JVM. However, to reduce the effect of JVM optimizations, each trial should be executed in its own VM [13,48]. JMH warns in the console that zero forks should only be utilized for debugging purposes. A total of 3'835 (68.3%) benchmarks are executed with exactly one fork. Only 282 (5.0%) benchmarks use at least five forks. Compared to the default value of five (for the current version 1.21) and 10 (for older versions), respectively, most of the benchmarks utilize fewer forks than proposed by JMH. The project *akashche/tzdata-jmh* uses 1'000 forks[10], but in combination with the singleshot mode, which reduces the execution time significantly. As such, the maximum number of forks without using the singleshot mode equals 20.

[10] `https://github.com/akashche/tzdata-jmh/blob/master/src/main/java/com/redhat/openjdk/tzdata/TzDataBenchmark.java`

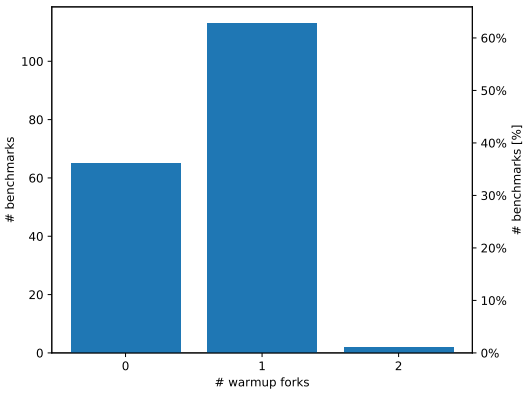


Figure 4.13: Distribution of the chosen warmup fork values (100% are 180 benchmarks where a value is present)

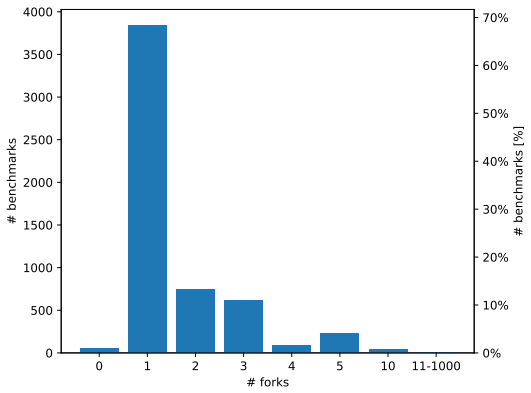


Figure 4.14: Distribution of the chosen fork values (100% are 5'618 benchmarks where a value is present)

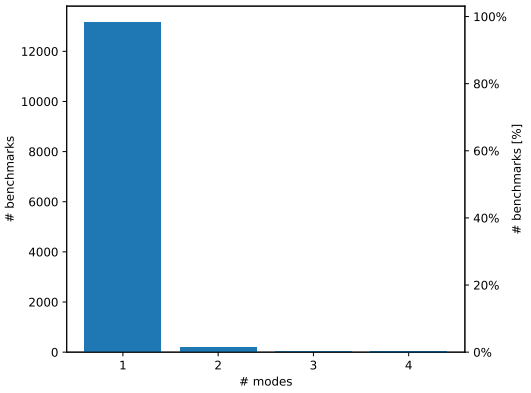


Figure 4.15: Number of modes executed (100% are all 13'387 benchmarks)

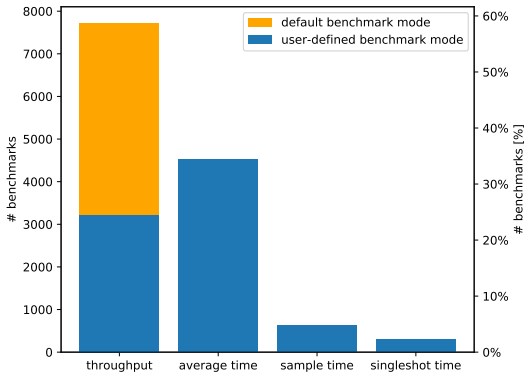


Figure 4.16: Chosen mode for executing the benchmark (100% are 13'148 benchmarks where exact one mode is performed)

JMH offers the option to select and run the benchmark with multiple modes. The `@Benchmark Mode` annotation accepts a list of modes. Figure 4.15 presents how many modes are run per benchmark. A total of 13'148 (98.2%) benchmarks are executed with exactly one mode. The information gain of running more than one mode is limited and does not justify the additional execution time.

Figure 4.16 presents how often which benchmark mode is chosen if exactly one mode is performed. In 34.2% of the cases, no `@BenchmarkMode` annotation is present, and the default mode, throughput, is selected. In 24.5%, the user-defined mode is set to throughput even if it is not necessary. In 34.4%, the mode is changed to average time. The throughput measures the number of operations executed per second, while the average time mode measures how long it takes to execute one operation on average. As such, one metric presents the inverse of the other. The sample mode works similarly to the average mode, but as output, a histogram is returned, and not the aggregated mean per iteration. A total of 4.8% of the benchmarks employ the sample mode, while 2.2% of the benchmarks utilize the singleshot mode, where only a single invocation of the method is run per iteration, and the cold start performance is measured. Lastly, 50 benchmarks

selected the "all" mode, where all four modes are executed sequentially. A point-biserial correlation analysis between number of forks and the usage of the singleshot mode produces a weak positive correlation coefficient of +0.11 under the significance level of 0.01. If the singleshot mode is used, the reduced execution time per fork is not compensated with more forks to obtain more precise results.

Except for the warmup forks and mode, all configuration options are preferably set on the class that belongs to the method, with 91.3% to 93.3%. The mode is set in 63.8% of the cases on the class. The warmup fork annotation is, at 62.2%, more often set on the method itself. In 15 benchmarks, the same value is set on the method, which was already set on the class. Meanwhile, on a method level, 38 benchmarks override the set class configuration parameter with a different value. Therefore, overriding pre-set values is extremely rare.

	Warmup iterations	Warmup time	Measurement iterations	Measurement time	Forks	Warmup forks
Warmup iterations	1.00**	-0.18**	0.69**	-0.28**	0.03	-0.38**
Warmup time		1.00**	-0.19**	0.81**	-0.30**	-0.54**
Measurement iterations			1.00**	-0.22**	0.08**	-0.81**
Measurement time				1.00**	-0.26**	-0.60**
Forks					1.00**	-0.30**
Warmup forks						1.00**

Table 4.5: Spearman correlation analysis of user-defined values between the different configuration options (correlation significant at 0.05 level *, 0.01 level **)

With a Spearman correlation analysis, we investigate whether, when one configuration option is changed, other parameters are also changed. Table 4.5 illustrates the correlation coefficient between user-defined values. First, we see that the warmup fork parameter negatively correlates with all other configuration options. As the warmup fork option is rarely present, the correlation is calculated on a small dataset. Nevertheless, the usage of warmup forks reduces all other configuration parameters, as the additional forks are compensated with less time per fork. The number of warmup iterations and the warmup time (respective measurement iterations and measurement time) weakly negatively correlate. If the number of iterations is increased, the time per iteration is rather reduced. A strong correlation exists between the number of warmup iterations and the number of measurement iterations. The same holds for the correlation between the warmup time and the measurement time, which also strongly correlates. If the number of forks is increased, the warmup and measurement time is reduced, as there is a negative correlation.

Next, we seek to analyze whether the configuration options correlate with the repository metrics. First, we use a Spearman correlation to analyze the configuration options except for the mode. Table 4.6 illustrates the correlation coefficients. Except for the warmup forks, only weak correlations exist. As only eight projects feature benchmarks with warmup forks, the results depend highly on the project metrics, and an interpretation is not meaningful. The number of iterations and forks tends to be a bit higher for projects with more stars, forks and so on, while the time per iteration is rather reduced. Table 4.7 illustrates that the mode is almost independent of the repository metrics. Projects with more stars and watchers prefer the throughput a bit more than the average time mode. We can conclude that the projects possessing higher repository metrics do not totally differ in how they configure their benchmarks.

Lastly, we analyze whether unknown projects with few stars, contributors and so on more often use the JMH default values. Table 4.8 illustrates only a small positive correlation between the use of user-defined (non-default) values and the project metrics. No large difference exists between the various configuration options.

	Stars	Forks	Watchers	Commits	Contributors
Warmup iterations	0.06**	0.08**	0.14**	0.11**	0.16**
Warmup time	-0.02	-0.15**	-0.17**	-0.07**	-0.22**
Measurement iterations	0.05**	0.09**	0.13**	0.08**	0.15**
Measurement time	-0.13**	-0.21**	-0.18**	-0.14**	-0.26**
Forks	0.07**	0.10**	0.12**	0.09**	0.21**
Warmup forks	-0.66**	-0.22**	-0.41**	-0.10	-0.42**

Table 4.6: Spearman correlation analysis between the repository metrics as independent variable and the configuration options as dependent variables (correlation significant at 0.05 level *, 0.01 level **)

	Stars	Forks	Watchers	Commits	Contributors
Mode is throughput	0.15**	0.09**	0.13**	-0.08**	0.01
Mode is average time	-0.13**	-0.08**	-0.12**	0.06**	0.00
Mode is sample time	-0.02*	-0.02	-0.02*	0.01	-0.03**
Mode is singleshot	-0.05**	-0.05**	-0.06**	-0.05**	-0.05**

Table 4.7: Point-biserial correlation analysis between the repository metrics as independent variable and the configuration options as dependent variables (correlation significant at 0.05 level *, 0.01 level **)

	Stars	Forks	Watchers	Commits	Contributors
At least one user-defined value is present	0.10**	0.09**	0.10**	0.10**	0.15**
Non-default warmup iteration value	0.19**	0.17**	0.18**	0.17**	0.23**
Non-default warmup time value	-0.02**	0.01	-0.00	0.11**	0.15**
Non-default measurement iteration value	0.18**	0.17**	0.18**	0.11**	0.22**
Non-default measurement time value	0.17**	0.12**	0.16**	0.07**	0.17**
Non-default fork value	0.15**	0.12**	0.15**	0.09**	0.20**
Non-default warmup fork value	-0.03**	-0.03**	-0.03**	-0.02**	-0.04**
Non-default mode	0.07**	0.04**	0.06**	-0.02*	0.09**

Table 4.8: Point-biserial correlation analysis between the repository metrics as independent variable and the binary variable if the default value was used as dependent variables (correlation significant at 0.05 level *, 0.01 level **)

To summarize the findings on how often default execution configuration are employed and what other values are used, in our dataset, 24.5% of the benchmarks use the default values for all seven configuration options. Warmup forks are set extreme rarely, while the number of iterations and forks is set in barely half of the benchmarks as a user-defined value. We found that some benchmarks explicitly set the default value as a user-defined value. Most of it can be traced back to default value change in JMH version 1.21. If the number of forks is set, the chosen number is usually smaller than the default value. In most cases, just one benchmark mode is used. The two most popular modes consist of throughput and average time. A correlation analysis between the configuration options revealed that they are not changed in isolation. Therefore, if the number of warmup iterations is increased, the number of measurement iterations is also increased. However, only a weak correlation is found between the execution configuration options and the repository metrics, such as stars, forks and contributors.

4.3.2 RQ1.2: Benchmark Execution Time

In the previous section, we analyzed the configuration options as single isolated components. However, all different configuration parameters influence the execution time, as demonstrated in Formula 2.3. A developer can decrease the number of iterations while increasing the time per iteration. In total, the execution time remains constant, but the other parameters are set. In this section, we investigate how the execution time in total as well as the ratio between warmup and measurement time have been changed. We ignore cases where only the singleshot benchmark mode is used, because there, we cannot estimate the used time. For the other modes, during an iteration, as many invocations as possible are performed in the pre-defined period, while the singleshot mode only invocates the method once.

		Measurement time		
		Decreased	Equal	Increased
Warmup time	Decreased	4'555 (94.2%)	0 (0.0%)	27 (0.6%)
	Equal	21 (0.4%)	10 (0.2%)	65 (1.3%)
	Increased	0 (0.0%)	0 (0.0%)	158 (3.3%)

Table 4.9: Measurement warmup time matrix (100% are 4'836 benchmarks where non-default configurations are used)

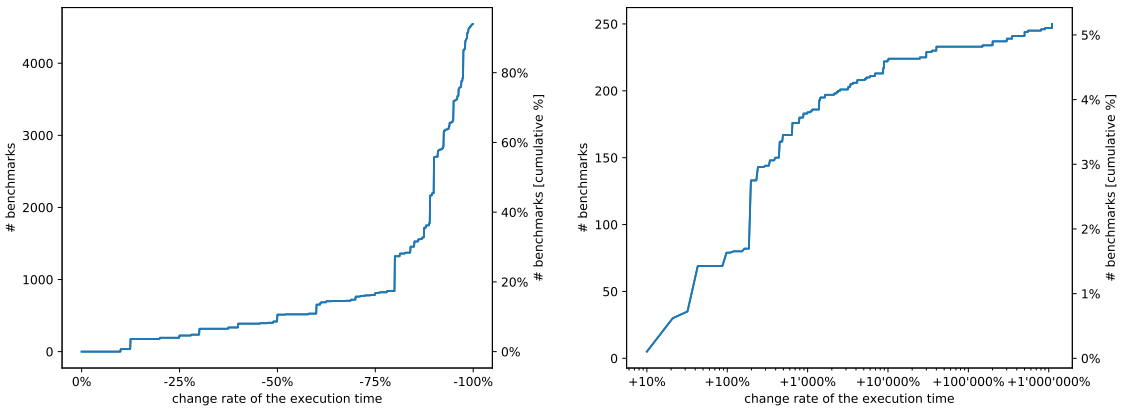
First, we created a measurement warmup time matrix in Table 4.9. We only analyzed a benchmark configuration if at least one parameter (except the mode) was changed. As such, the analyses in this section are always based on 4'836 benchmarks. In 94.2% of the cases, the measurement as well as the warmup time were decreased. As the second most common option for both phases, the time consumption increased.

We wish to compare the consumed execution time with the default execution time. Equation 4.1 defines the ratio between the two values. A ratio of 0.25 means that the execution time was decreased to 25% of the initial execution time.

$$executionTimeRatio = \frac{execution\ time\ with\ chosen\ execution\ configuration}{execution\ time\ with\ default\ execution\ configuration} \tag{4.1}$$

Figure 4.17 illustrates how the execution time change rate compared to the default values. In 10 cases, some configuration parameters are changed, but the execution time remains exactly the same. Meanwhile, 4'576 (94.6%) benchmarks reduce the execution time. Figure 4.17a illustrates how much the execution time of these benchmarks was decreased. The change rate on the x-axis is $1 - executionTimeRatio$ as percentage. In 3'762 (77.8%) benchmarks, the execution time was reduced by over 75%. For 1'879 (38.9%) benchmarks, the execution time diminished by over 90%. As an extreme case, for some benchmarks, the project *tzaeschke/distributed-phtree* reduced the total execution time per benchmark to 25 microseconds[11]. We can conclude that if the execution time is reduced, only a few benchmarks make moderate modifications, and in most cases, the time is massively reduced.

On the right side, we see in Figure 4.17b how much the execution time is increased. In 250 (5.2%) benchmarks, it takes longer with the chosen configuration than the default execution time. Only 17 (3.0%) projects feature at least one benchmark where the execution time is increased. However, if the execution time is increased, it is not just increased by a small amount. In 67 (1.3%) cases, the execution time takes 10 or more times longer. The project *kiegroup/kie-benchmarks* features some benchmarks where the execution time lasted over 63 days[12]. We also want to note other suspicious configurations. First, 49 benchmarks employed no warmup time. Second, two benchmarks set the number of measurement iterations to zero[13].



(a) Decreased execution time

(b) Increased execution time

Figure 4.17: Effective execution time of a benchmark compared to the default execution time (100% are 4'836 benchmarks where non-default configurations are used)

Next, we investigate how the measurement time was changed compared to the warmup time. In Formula 4.2, we defined warmup proportion. $warmupTime$ and $executionTime$ are calculated as defined in Formula 2.1 and 2.3. With the default configuration, both phases consume the same amount of time ($warmupProportion = 50\%$).

$$warmupProportion = \frac{warmupTime}{executionTime} \tag{4.2}$$

[11] https://github.com/tzaeschke/distributed-phtree/blob/master/benchmark/src/main/java/ch/ethz/globis/distindex/cloning/CloningBenchmark.java

[12] https://github.com/kiegroup/kie-benchmarks/blob/master/drools-benchmarks/src/main/java/org/drools/benchmarks/session/EmptySessionWithInsertionsAndFireBenchmark.java

[13] https://github.com/atlanmod/NeoEMF/blob/master/benchmarks/core/src/main/java/fr/inria/atlanmod/neoemf/benchmarks/runner/WriteOnlyRunner.java

Table 4.10 illustrates how the $warmupProportion$ has been changed. A total of 46.8% of the benchmarks keep the same ratio — in other words, both parts consume the same amount of time. For 35.4% of the benchmarks, the measurement time is increased compared to the warmup time. Meanwhile, 17.9% of the benchmarks increase the warmup time proportionally compared to the measurement time. Figure 4.18 demonstrates how much of the execution time is spent warming up the benchmark. In 638 (13.24%) benchmarks, the warmup time takes 20% or less of the total execution time. If the warmup proportion is increased compared to the default, a proportion of 66.7% or 90.9% is often chosen. There is a wide range for how the warmup proportion is chosen. In half of the cases, the ratio is not modified. In the other cases, the chosen measurement time takes between 0.02%[14] and 99.2%[15] of the total time.

Category	# Benchmarks
$warmupProportion < 50\%$	1'711 (35.4%)
$warmupProportion = 50\%$	2'262 (46.8%)
$warmupProportion > 50\%$	863 (17.9%)

Table 4.10: Proportion between warmup and measurement time (100% are 4'836 benchmarks where non-default configurations are used)

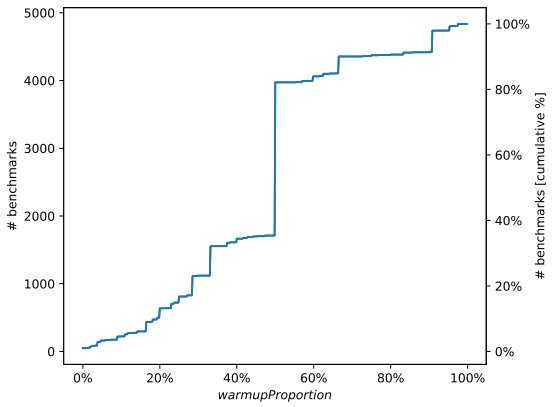


Figure 4.18: Visualization of the warmup proportion (100% are 4'836 benchmarks where non-default configurations are used)

Next, we analyze the correlation of the used execution time, warmup proportion and project metrics. Only a weak correlation is found. Table 4.11 illustrates the correlation values. Projects with more stars, forks, contributors and so on allow the benchmarks run a bit longer. For the warmup proportion, we did not detect any correlation.

[14] https://github.com/apache/hive/blob/master/itests/hive-jmh/src/main/java/org/apache/hive/benchmark/vectorization/mapjoin/AbstractMapJoin.java#L72

[15] https://github.com/CrispOSS/prime-sieves/blob/master/src/main/java/com/github/crisposs/sieves/benchmark/SieveBenchmark.java

	Stars	Forks	Watchers	Commits	Contributors
$executionTimeRatio$	0.12**	0.15**	0.16**	0.18**	0.19**
$warmupProportion$	-0.06**	-0.06**	-0.00	-0.05**	-0.02

Table 4.11: Spearman correlation analysis with the repository metrics as independent variables and the ratios as dependent variables (correlation significant at 0.05 level *, 0.01 level **)

Lastly, we analyze a single benchmark, but unlike this section's previous analyses, benchmarks with different parameterizations are grouped together and analyzed as a single unit. Therefore, the execution time is multiplied by the number of parameterization combinations. Figure 4.19 illustrates the execution time per benchmark with all parameterization combinations. The median execution time equals 6.7 minutes. A total of 88.3% of these benchmarks feature a total execution time less than 10 minutes. However, for 6.3% of the benchmarks, it takes longer than half an hour to execute all parameterization combinations. The number of a benchmark's parameterization combinations does not correlate with the execution time of a single parameterization combination. As such, if more parameterization combinations are defined, the total time per parameterization is not reduced.

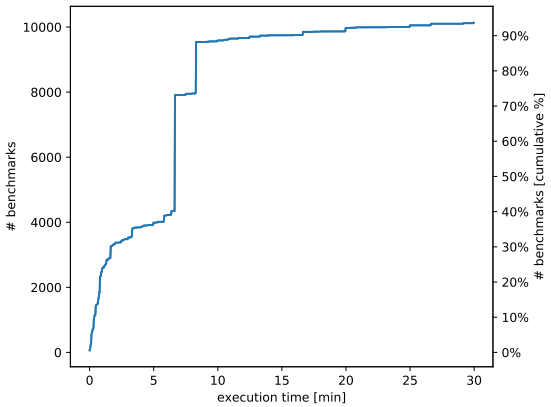
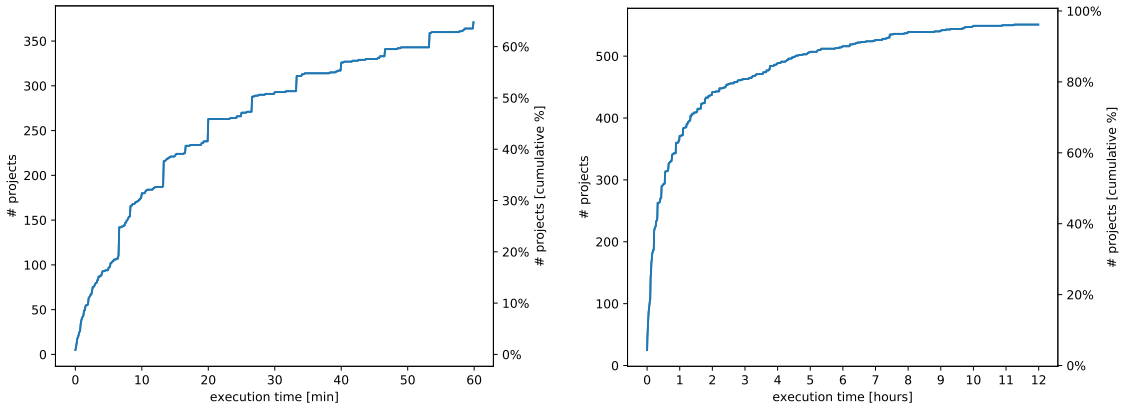


Figure 4.19: Execution time to run one benchmark with all parameterization combinations (100% are all 10'816 benchmarks with a JMH version)

We can conclude most developers reduce both measurement and warmup time. The total execution time is often drastically reduced if something in the execution configuration is changed. The ratio between measurement and warmup time is, in 46.8% of the cases, not changed. If it is modified, oftentimes, the warmup time is diminished compared to the measurement time. We did not identify a strong correlation between the execution time and repository metrics.

4.3.3 RQ1.3: Project Test Suite Execution Time

This section analyzes the execution time of not just a single benchmark, but the entire test suite. As in the previous section, if we cannot extract a JMH version, an analysis of the test suite execution time is not possible, because we do not know the default execution configuration. If we summarize the execution time of all benchmarks in a project, we obtain the execution time of the project test suite. Figure 4.20 presents the project test suite execution time. For 63.7% of the test suites, it takes less than one hour to execute all benchmarks. As numerous projects feature only a few benchmarks and not many parameterization combinations, these microbenchmarking test suites possess only an execution time of minutes, as illustrated in Figure 4.20a. For 19.3% of the project test suites, the execution takes more than three hours, and 3.8% of them take longer than 12 hours, as illustrated in Figure 4.20b. For example, for *eclipse/eclipse-collections*, executing the test suite takes over 16 days, because the project possesses 515 benchmarks and a total of 2'575 parameterization combinations. The median execution time of a project test suite equals 26.7 minutes. On one hand, the minimum test suite execution time equals 143 milliseconds for the project *protobufel/protobuf-el*. On the other hand, the maximum execution time is 7.4 years for *kiegroup/kie-benchmarks*. Compared to the median execution time of 138.9 minutes using the default execution configuration, the execution time is reduced by more than a factor of five. However, a considerable amount of time is still needed to execute the entire project test suite.



(a) < 1 hour (b) all

Figure 4.20: Visualization of the execution time over project test suites (100% are 573 projects with an extracted JMH version)

We can summarize that the number of parameterization combinations is multiplied with the execution time of one parameterization combination. As the Cartesian product of all JMH parameters is built, this number can quickly grow. The median execution time of 26.7 minutes disguises the fact that some projects possess numerous benchmarks, resulting in a time-consuming microbenchmarking lasting multiple hours.

4.3.4 RQ1.4: Execution Configuration Modification Frequency

Last, we want to investigate how often a benchmark is modified after its creation. As we perform the historical analysis only on sampled commits, we do not see every change. We select only one commit per month. If, during a month, a benchmark was modified more than once, we can only count it as one change. If, during a month, a benchmark was changed and the change was undone immediately afterwards, we do not see any change. However, we can analyze how the benchmark was changed in the long-term perspective. Additionally, we want to note that if the fully qualified name of a benchmark was changed, the heuristic believes that a benchmark was removed and a new one is created.

Throughout the project history, we can detect 19'211 benchmarks. Only 216 (1.1%) have another benchmark configuration present somewhere in the history. Figure 4.21 illustrates the distribution of how often the configuration was changed. If a benchmark configuration was modified, there was usually exactly one modification in the history. On average, 0.01 ± 0.11 configuration modifications were performed. A total of 2'041 (10.6%) benchmarks modified the benchmark method body after a benchmark was created. Figure 4.22 presents how often the code was changed. On average, a code modification occurred 0.15 ± 0.64 times. Often, a method body was also only modified once. We can conclude that benchmarks are usually written once, and then never updated. If a benchmark is modified more often, the implementation is changed.

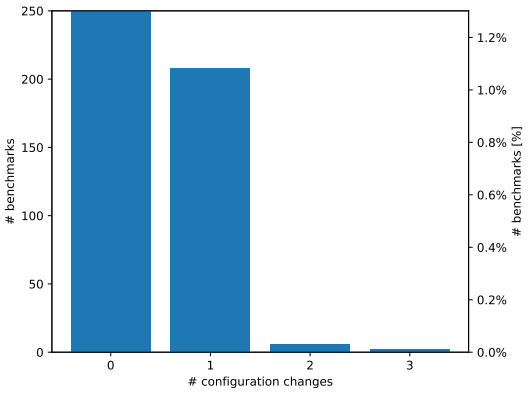


Figure 4.21: Execution configuration modification frequency after the benchmark creation (100% are all 19'211 benchmarks)

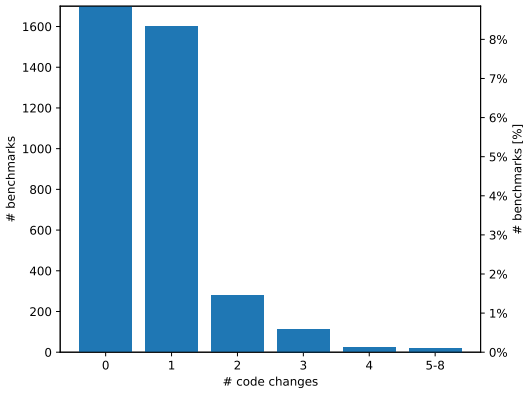


Figure 4.22: Implementation modification frequency after the benchmark creation (100% are all 19'211 benchmarks)

With the JMH update 1.21, new default execution configurations are introduced. Accordingly, we are interested in how developers react to these new default values. The analysis is limited to 3'682 benchmarks where we have at least one commit with a pre-1.21 version and one commit with JMH version 1.21. We take and compare the last sample commit before the version change to 1.21 against the current commit.

In total, 3'469 (94,2%) benchmarks did not perform an active configuration change of any parameter. In Table 4.12, we grouped the benchmarks depending on how they reacted to the version update. The first group contains all type of changes where the developer performed an active change to the source code. The time per warmup iterations is, with 5.05% of the configuration options, most often actively changed. We see that for some benchmarks, the warmup and measurement time were actively set to the old default value ($\rightarrow d_o$). Other projects using the old default value changed to a user-defined non-default value ($d_o \rightarrow u_2$). A few benchmarks had already set the new default value before and removed the annotation after the update, as the annotation was no longer required ($d_n \rightarrow$).

	Warmup iterations		Warmup time		Measurement iterations		Measurement time		Forks	
Active change	**51**	**(1.39%)**	**186**	**(5.05%)**	**49**	**(1.33%)**	**181**	**(4.92%)**	**31**	**(0.84%)**
$\rightarrow d_o$	0	(0.00%)	68	(1.85%)	0	(0.00%)	63	(1.71%)	0	(0.00%)
$u_1 \rightarrow u_2$	14	(0.38%)	0	(0.00%)	12	(0.33%)	0	(0.00%)	5	(0.14%)
$d_o \rightarrow u_2$	13	(0.35%)	96	(2.61%)	11	(0.30%)	96	(2.61%)	4	(0.11%)
$u_1 \rightarrow d_n$	0	(0.00%)	0	(0.00%)	0	(0.00%)	0	(0.00%)	22	(0.60%)
$d_o \rightarrow d_n$	2	(0.05%)	22	(0.60%)	4	(0.11%)	22	(0.60%)	0	(0.00%)
$d_n \rightarrow$	22	(0.60%)	0	(0.00%)	22	(0.60%)	0	(0.00%)	0	(0.00%)
Passive change	**2'124**	**(57.69%)**	**3'159**	**(85.80%)**	**2'119**	**(57.55%)**	**2'811**	**(76.34%)**	**2'281**	**(61.95%)**
$\rightarrow$	2'124	(57.69%)	3'159	(85.80%)	2'119	(57.55%)	2'811	(76.34%)	2'281	(61.95%)
No change	**1'507**	**(40.93%)**	**337**	**(9.15%)**	**1'514**	**(41.12%)**	**690**	**(18.74%)**	**1'370**	**(37.21%)**
$d_o \rightarrow d_o$	57	(1.55%)	266	(7.22%)	29	(0.79%)	514	(13.96%)	0	(0.00%)
$d_n \rightarrow d_n$	864	(23.47%)	1	(0.03%)	834	(22.65%)	3	(0.08%)	5	(0.14%)
$u_1 \rightarrow u_1$	586	(15.92%)	70	(1.90%)	651	(17.68%)	173	(4.70%)	1'365	(37.07%)

Table 4.12: Benchmark configuration reaction on the update to JMH version 1.21, 100% are 3'682 benchmarks (d_o = pre 1.21 default value, d_n = 1.21 default value, u = user-defined value which is not d_o or d_n, = no annotation is present)

A majority of the benchmarks did not possess an annotation present before and after the update ($\rightarrow$). The developer did not actively change something, but as the default execution configuration was passively changed, a configuration modification occurred. Depending on the configuration parameter, no user-defined value is set for between 57.55% and 85.80% of the benchmarks. The last category features benchmarks where a value had already been set as a user-defined value, and the value did not change. In one quarter of all benchmarks, the new default value for the number of iterations was already set as a user-defined value and the annotation was not removed ($d_n \rightarrow d_n$). For the time per iteration, some benchmarks had manually set the old default value before ($d_o \rightarrow d_o$). In the future, if we conduct a survey with developers, we should ask the developers why they set the old default value as user-defined values, as this was not necessary. Lastly, we wish to note that, for the number of iterations and forks, some projects set user-defined values that are not equal to the old and new default values ($u_1 \rightarrow u_1$).

To summarize, only a few benchmarks changed the execution configuration to the old default values after the update. More often, the old default value was already previously set as user-defined value. Most of the project either did not set a user-defined, non-default value and simply used the default values where a passive change happened, or else they had already manually set a non-default value and nothing changed. For the warmup and measurement iterations, developers already often used the new default values before the JMH default value update.

Chapter 5

Reconfiguration Approach

The following chapter introduces the reconfiguration approach. Afterwards, the project selection and data collection is briefly described. Lastly, the different aspects of the second research questions are investigated.

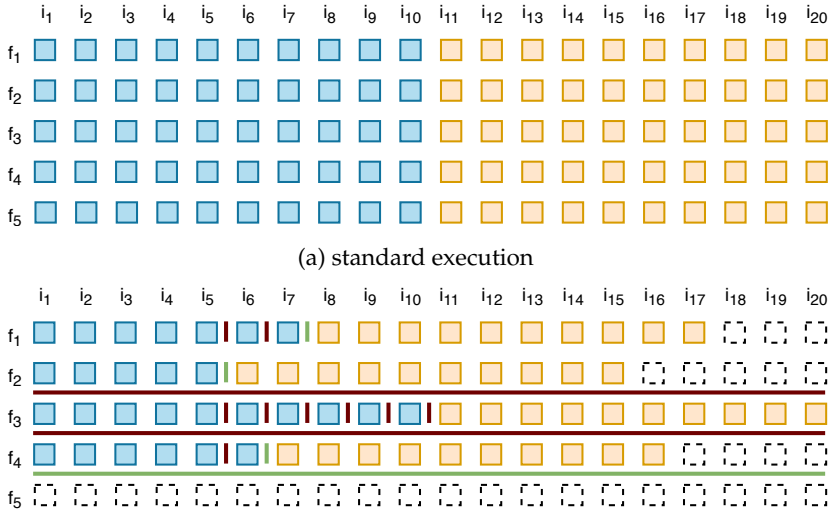
5.1 Approach

Traditionally in JMH, a benchmark possesses a fixed warmup and measurement time. Each phase is divided into multiple iterations. The question thus concerns how a developer can determine the optimal length of the warmup phase. If the warmup phase is too short, no steady state is reached, and the results are not representative. If the warmup phase is too long, time is wasted. Theoretically, a developer can determine the optimal warmup time for each benchmark. However, depending on the hardware, the same benchmark does not require the same warmup time. The JMH core development team solved this issue using pessimistic default values that, in most benchmarks, wastes time but produces good results.

Our approach consists of dynamically assessing how long the warmup phase should be. A developer simply sets an lower-bound and upper-bound warmup execution time. Between each iteration, we check the stoppage criteria to verify whether a steady state is reached. If the stability threshold is reached, the warmup phase is stopped and the measurement phase begins. The measurement phase remains a fixed amount of time. We do not solely optimize the number of warmup iterations; we also optimize the number of forks. If at least two forks are executed, we check the stability between the forks. We apply the same stoppage criteria as for iterations in order to check the result characteristic. If different forks possess the same measurement result distribution, it is not necessary to execute five forks as proposed by the JMH default values.

Our approach of dynamically reconfiguring the number of warmup iterations requires several decision points where the warmup phase is either stopped or continued. The JMH default number of warmup iterations is five in the current JMH version. The problem is that the decision of whether the warmup phase is continued is based on the stability of the last few data points. We decided to compare the stability of the last five data points to reduce the effect of outliers. However, if at least five data points are required, we cannot reduce the number of warmup iterations. This results in a change of the default configuration from five 10-seconds warmup iterations to 50 one-second warmup iterations to achieve sufficient decision points. We decided to use one-second iterations, as they were the default value in previous JMH versions.

Figure 5.1 visualizes how the reconfigured approach differs from the standard execution. In this illustrative example, five forks with 10 warmup and 10 measurement iterations are performed. The execution of the standard approach is summarized in Figure 5.1a. The visualization



(a) standard execution

(b) reconfigured execution

Figure 5.1: Execution strategy visualization: A blue box is a warmup iteration, an orange box is a measurement iteration, a dotted box is a skipped iteration, a green line indicates that the stoppage criteria is reached, and a red line indicates that the result is not stable

should be read from left to right and from top to bottom. As such, the first fork is spawned and 10 warmup iterations are executed. Afterward, 10 measurement iterations are performed, and the result of the measurement phase is stored. The next four forks are executed in the same way. In the reconfigured execution, the number of warmup iterations and forks are dynamically determined, as presented in Figure 5.1b. However, some restrictions remain. At least five warmup iterations and two forks are executed. Therefore, the first fork is spawned and five warmup iterations are executed. From now on, after each warmup iteration, the stoppage criteria are computed and compared with the threshold. The vertical line indicates the result of the statistical evaluation. After the fifth and sixth iteration, the result is not sufficiently stable, meaning that the result characteristic still changes after each new executed iteration. However, after the seventh iteration, the stability threshold is reached and the measurement phase begins. Exactly 10 measurement iterations are executed. Therefore, in the first fork, three iterations are saved, because the warmup phase is stopped earlier. In the first statistical evaluation of the second fork, stable results are already produced. After the second fork's measurement phase, we now additionally check whether the results from the measurement phases over all executed forks are stable and no more forks are necessary. The horizontal red line indicates that this is not the case, and that more forks are required. In the third fork after 10 warmup iterations, the results remain unstable. However, the maximum number of warmup iterations is reached, and the measurement phase is started. The command line and the output file contain a warning that, in a fork, the warmup phase was too short and the stability threshold was not reached. After the fourth fork, the results are stable, and the fifth forks is skipped.

5.1.1 Stoppage Criteria

In this section, we seek to explain the stoppage criteria. This includes how we measure the stability of a performance distribution, based on which we decide on the result's quality. A distribution is stable if adding new data points does not change the distribution characteristics. As an exam-

ple, Figure 5.2 presents a significant difference between the two distributions of benchmark A. The blue coloring represents the first n iterations. Between the two distributions, one new iteration $n + 1$ is executed. The orange coloring represents the first $n + 1$ iterations. A significant mean shift occurs as the invocation time in iteration $n + 1$ is smaller than the previous ones. For benchmark B, the characteristic change is too small to be significant. However, small differences remain; for example, the standard deviation is a bit higher after $n + 1$ iterations.

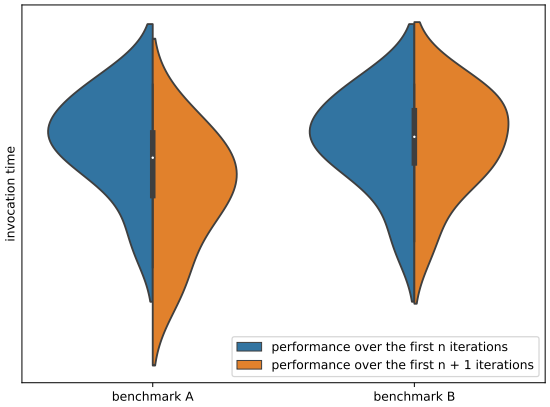


Figure 5.2: Illustrative example on what stability of a distribution means

We decided to support three different stoppage criteria: The first is the CoV, where the calculation is explained in Section 2.2.1. Figure 5.3 illustrates the CoV values over the number of iterations. The value in iteration nine is the variation over the data points of the first nine iterations. Depending on the benchmark task, the CoV converges to different values. We do not analyze the absolute value. Therefore, we compute the delta between the last five points and check whether it is below a pre-defined threshold. As an example, the CoV in iteration 13 equals 0.052. We compute the absolute difference to iteration 12 (0.039), iteration 11 (0.052), iteration 10 (0.045) and iteration 9 (0.050). The maximum delta value over the last data points equals 0.007. We terminate the warmup phase if the maximum delta value of the last five data points is below 0.01 for the CoV.

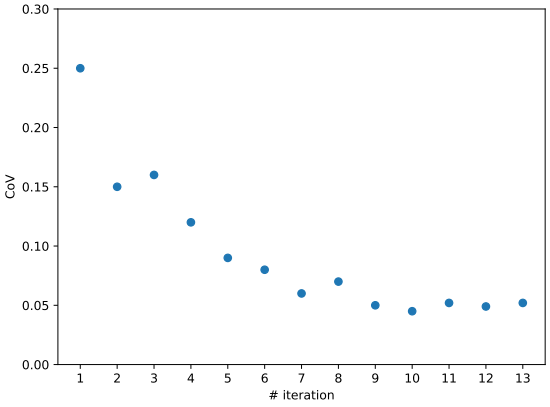


Figure 5.3: Illustrative example on how the stoppage criteria is computed

The second criteria is the CI width. We modified the practice proposed by Georges [20]. They stopped the measurement if CI width divided by the mean was smaller than 0.03. Instead, we used bootstrapping CI, as explained in Section 2.2.2. Such CIs are more variable and often larger. As for the CoV, we calculated the absolute delta values over the last five data points and used 0.03 as threshold. We reused the Performance (Change) Analysis tool[1], which computes the CI using bootstrapping. We always used a confidence level of 99%. The stoppage criteria employs 1'000 bootstrap simulation, as the performance overhead of the statistical evaluation would otherwise be too large.

The final criteria is the Kullback-Leibler divergence, as previously employed by He et al. for load tests [23] and previously explained in Section 2.1.7. We modified the computation a bit, though. They partitioned each distribution into 1'000 strips. However, if the distribution possesses outliers, 1'000 strips are not enough. Increasing the number of strips produces a higher calculation time for the kernel density estimation technique with the Gaussian kernel. We decided to take 1'000 strips, but limit the analysis to a certain range. Our minimum point is $Q_1 - 1.5 * IQR$, and the maximum point is $Q3 + 1.5 * IQR$, where IQR is the Interquartile Range (IQR), Q_1 is the first quartile, and Q_3 the third quartile. If the average p-value of the last five comparison is larger than the threshold of 0.99, the execution is stopped.

As the statistical evaluation is always performed between iterations, the total execution time is higher. To reduce the computational overhead, the number of data points is reduced by only taking a weighed sample of 1'000 points for the CoV and divergence criteria and 10'000 data points for the CI width criteria. Additionally, outliers that are more than one magnitude higher than the median are removed, because they exert too much influence on the mean.

5.1.2 Modified JMH Implementation

On GitHub, we provided the modified JMH implementation for version 1.21[2]. We added an additional benchmark mode named "reconfiguration." We provided the three previously described reconfiguration modes: CoV, CI width and divergence. To use the modified JMH implementation, the version must be published to the local maven repository with `mvn install`. Now, other projects can reuse this version by defining a dependency with the *groupId* `org.openjdk.jmh`, *artifactId* `jmh-core` and the *version* `1.21-Reconfigure`.

We added new annotations, annotation properties and CLI flags. Table 5.1 lists the new possibilities. First, the mode enum is extended with the option `Mode.Reconfigure`. Additionally, a developer can set the minimal number of forks and warmup iterations. Then the first *n - 1* statistical evaluations are skipped. The reconfiguration mode is the stoppage criteria where the options — `ReconfigureMode.COV`, `ReconfigureMode.CI` and `ReconfigureMode.DIVERGENCE` — can be chosen. For each, the threshold can be set as a property in the `@Reconfigure` annotation. The CoV and CI threshold value can be any non-negative double. The divergence threshold value must be a probability between zero and one. The existing annotation properties `@Fork(value=5)`, `@Fork(warmups=0)` and `@Warmup(iterations=50)` are used as upper-bound execution configuration if the `Mode.Reconfigure` is executed.

Lastly, we modified the console and JSON output. Similar information is added. If a fork is terminated earlier, the current iteration number, the stability metrics and the corresponding threshold are printed on the console. In the file output, the stability values of the stoppage criteria are stored. If the maximum number of forks or warmup iterations is reached, and the results are still not stable, a warning is printed.

[1] `https://github.com/chrstphlbr/pa`
[2] `https://github.com/stewue/jmh`

Description	Annotation	CLI
Minimum number of forks	`@Fork(minValue=2)`	*-mf 2*
Minimum number of warmup forks	`@Fork(minWarmups=0)`	*-mwf 0*
Minimum number of warmup iterations	`@Warmup(minIterations=5)`	*-mwi 5*
Reconfiguration mode	`@Reconfigure(mode=COV)`	*-rm cov*
CoV threshold	`@Reconfigure(covThreshold=0.01)`	*-rcov 0.01*
CI width threshold	`@Reconfigure(ciThreshold=0.03)`	*-rci 0.03*
Divergence threshold	`@Reconfigure(kldThreshold=0.99)`	*-rkld 0.99*

Table 5.1: Added and modified JMH annotations and CLI flags

5.1.3 Project Selection

Executing benchmarks represents a time-consuming task. We cannot execute all benchmarks found in RQ1. Instead, we selected a subset of projects where all benchmarks are executed[3]. Table 5.2 summarizes the chosen projects. In total, 3'969 benchmarks are chosen to analyze RQ2. This comprises 29.6% of the 13'387 benchmarks. The selected projects derive from different domains. On one hand, there are popular projects with numerous forks and stars, and on the other hand, smaller projects are also selected. Additionally, we also executed the benchmarks of JMH itself and the official Java Development Kit (JDK) benchmarks from OpenJDK. The analysis is performed on the latest released version where the commit hash is specified in the version column. The Java target version specifies the Java version employed by the project. All projects are executed with JDK version 8, except *apache/logging-log4j2*, where JDK version 13 is used for the benchmarking. The execution time column contains the time to execute all benchmarks with the standard configuration. Some projects feature considerable difference between the effective execution time and the execution time with the default configuration.

5.1.4 Data Collection

Previous literature has proposed executing performance tests on a bare-metal machine [5, 47]. Resources should be exclusively reserved for the benchmark execution. Our machine possesses a 12-core Intel Xeon X5670 @2.93GHz CPU with 70 GiB memory, and it runs ArchLinux with a Linux kernel version 5.2.9-1-1-ARCH. It uses a Samsung SSD 860 PRO SATA III disk. During the benchmark execution, no other user-space application except ssh was run. We did not explicitly disable hardware optimizations. We also want to note that the benchmark execution never fully utilized the resources.

We overrode all benchmark execution configurations via CLI arguments to obtain comparable results between different benchmarks. We executed each benchmark once with the configuration of five forks, zero warmup iterations and 100 one-second measurement iterations. We do not reuse the annotations present in the code, because the approach can only save time if sufficient warmup iterations and forks are performed. For only 4 benchmarks the execution time is reduced. For the benchmarks `org.openjdk.bench.java.security.MessageDigests.digest` and `org.openjdk.bench.javax.crypto.Crypto.{encrypt, decrypt}` from the project *jmh-jdk-microbenchmarks* the execution time is reduced from 750 to 500 seconds. For the benchmark `org.openjdk.jmh.benchmarks.ScoreStabilityBench.test` from the project *jmh-core-benchmarks* the custom execution time is 5'000 seconds. Later, we read the JSON output file to obtain the data for evaluation.

[3]Except failing benchmarks (see Appendix C)

Project	Abbreviation	# Benchmarks	# Parametrization combinations	Execution time	JMH version	Version	Java target version	Area
ReactiveX/RxJava	*RxJava*	217	1'282	178.06h	1.20	17a8eef	1.6	Asynchronous, event-based programming
jmh-jdk-microbenchmarks[a]	*jdk*	994	1'381	191.81h	1.21	d0fab231915f[b]	1.8	JDK
apache/logging-log4j2	*log4j2*	358	510	70.83h	1.19	ac121e2	9	Logging
jmh-core-benchmarks[c]	*jmh-core*	110	110	15.28h	1.21	a07e91499e97[d]	1.7	JMH
raphw/byte-buddy	*byte-buddy*	39	39	5.42h	1.16	c24319a	1.7	Bytecode
JCTools/JCTools	*JCTools*	60	148	20.56h	1.21	19cbaae[e]	1.6	Concurrency
protostuff/protostuff	*protostuff*	16	31	4.31h	1.6.1	2865bb4	1.6	Serialization
jenetics/jenetics	*jenetics*	40	40	5.56h	1.21	002f969	1.8	Genetic algorithm
SquidPony/SquidLib	*SquidLib*	269	367	50.97h	1.21	055f041	1.7	Visualization
openzipkin/zipkin	*zipkin*	61	61	8.47h	1.21	43f633d	1.8	Distributed tracing

Table 5.2: Selected projects for the second research question

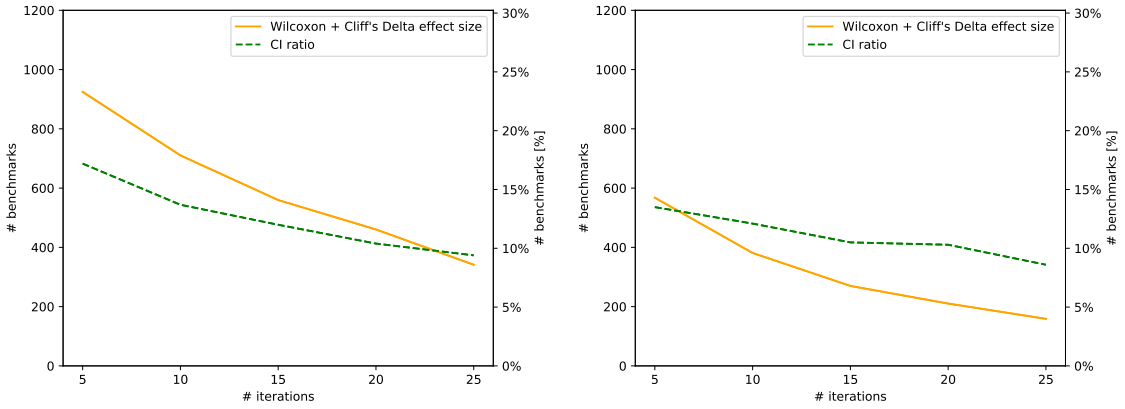
[a] Source code repository `http://hg.openjdk.java.net/code-tools/jmh-jdk-microbenchmarks/`
[b] The repository does not have release versions. We just use the last revision
[c] module in directory *jmh-core-benchmarks* of repository `https://hg.openjdk.java.net/code-tools/jmh`
[d] The repository does not have release versions. We just use the last revision
[e] The repository does not have regularly release versions. The last release was over a year old. We just use the last commit

5.2 Results

In this section, the different aspects of the performance result are presented. First, the effect of shortening the measurement phase is illustrated. Next, the result quality of the reconfiguration approach is explored. Lastly, the influence on the execution time is discussed.

5.2.1 RQ2.1: Length of Measurement Phase

Theoretically, after the warmup phase, the benchmark has reached a steady state, and one iteration is sufficient to obtain a sound result. As described by Gil et al. [21], there is no single steady state, and the results vary between different iterations. The question thus becomes how many measurement iterations are needed to achieve a result possessing a negligible difference to the default of 50 measurement iterations. To this end, we computed two A/A tests: First, for how many benchmarks the Wilcoxon and Cliff's Delta effect size test detect a significant distribution change compared to the execution with more measurement iterations. We employ a confidence level of 99%. If the p-value of the Wilcoxon test is smaller than 0.01, there is a significant difference. However, a significant difference between two distributions is not classified as significant if the Cliff's Delta effect size is negligible ($|d| < 0.147$), as defined by Romano et al. [45]. Second, the CI ratio between first n measurement iterations and all 50 iterations is computed with 10'000 bootstrap simulations. If the ratio is at least 1% significantly different, we classify the distribution as significantly different. We compare 5, 10, 15, 20 and 25 iterations with the baseline of 50 measurement iterations.



(a) Single fork

(b) Five forks

Figure 5.4: How often does a shortened measurement phase end in a significant different distribution

Figure 5.4 summarizes the number of benchmarks for which the A/A tests revealed a significant difference between the first n iterations and the baseline of 50 iterations. First, we analyze a single fork in Figure 5.4a. In 17.2% of the benchmarks, the CI ratio test revealed a significant difference of at least 1% between the first 5 and 50 measurement iterations. The more iterations are added, the fewer benchmarks produce significant changes. A trade-off exists between the number of measurement iterations that result in a higher execution time and the result quality. Running 25 measurement iterations ends in only 9.4% of benchmarks where a significant change is detected. With only 10 iterations, the CI ratio criteria detect a difference greater than 1% for only 13.7% of benchmarks. The combination of Wilcoxon and Cliff's Delta effect size is a bit worse with

17.9% after 10 iterations. Figure 5.4b demonstrates that multiple forks reduce the detection rate of significant changes for both A/A tests. Running multiple forks is important, as different steady states are considered. Especially for the combination of Wilcoxon and Cliff's Delta, only half as many significant different distributions are detected. The CI ratio detection rate is also decreased by running more forks. With the decision to run 10 measurement iterations to evaluate the reconfiguration approach in 12.1% of the benchmarks, the shortened measurement phase results in a significantly different distribution using the CI ratio test. After 10 iterations, the CI ratio curve is quite flat, and running more iterations reduces the number of significant differences only slightly.

5.2.2 RQ2.2: Result Quality

For each stoppage criteria, two A/A tests are computed to compare the result quality between the standard and reconfiguration approaches. We apply two A/A tests — the confidence interval ratio and the combination of Wilcoxon and Cliff's Delta effect size — as in the previous section. There is no significant difference in CI ratio if the ratio is smaller than 1%. For the other A/A test, the two groups significantly differ if the effect size $|d| \geq 0.147$ and the p-value of the Wilcoxon test is $p < 0.01$.

In Table 5.3, the results of the A/A tests are summarized to illustrate how often, for each stoppage criteria, a significantly different result is detected between the reconfigured and standard approach. Both tests identify the CoV stoppage criteria as the worse criteria of them all. In general, the Wilcoxon and Cliff's Delta test detect more significant changes than the CI ratio comparison. If we examine the CI width as stoppage criteria, the CI ratio is larger than 1% in only 12.4% of the benchmarks. The Kullback-Leibler divergence produces the best results for the Wilcoxon and effect size test. During the evaluation, we noted that the null hypothesis of the Wilcoxon test is often rejected, as the p-value is much smaller than 0.01, but the effect size is often negligible.

	CoV	CI width	Divergence
Wilcoxon + Cliff's Delta effect size test	36.4%	28.2%	24.3%
CI ratio test	21.2%	12.4%	20.4%

Table 5.3: Significantly different result distribution between the standard and reconfigured execution

	CoV	CI width	Divergence
Change rate	3.1%±8.1%	1.4%±3.8%	2.4%±7.4%

Table 5.4: Change rate between standard and reconfigured execution

Table 5.4 illustrates the mean change rate per stoppage criteria. Again, the CoV stoppage criteria is the worst, with an average change rate of 3.1%. If we examine the CI width as stoppage criteria, it offers the best average change rate at 1.4%. In between lies the divergence criteria. Next, the CI ratio shift between the standard and reconfigured execution result is analyzed. The Performance (Change) Analysis tool returns the mean, lower bound of the CI CI_L and upper bound of the CI CI_U at the confidence level of 99%. The minimal and maximal CI ratio shift is calculated. First, the absolute lower and upper bound is calculated in Formula 5.1 and 5.2.

$$CI_{L_{ABS}} = abs(1 - CI_L) \tag{5.1}$$

$$CI_{U_{ABS}} = abs(1 - CI_U) \tag{5.2}$$

This results in the minimum and maximum CI ratio shift as defined in Formula 5.3 and 5.4.

$$min = \begin{cases} CI_L \leq 1 \text{ and } CI_U \geq 1, & 0 \\ \text{else}, & min\{CI_{L_{ABS}}, CI_{U_{ABS}}\} \end{cases} \tag{5.3}$$

$$max = max\{CI_{L_{ABS}}, CI_{U_{ABS}}\} \tag{5.4}$$

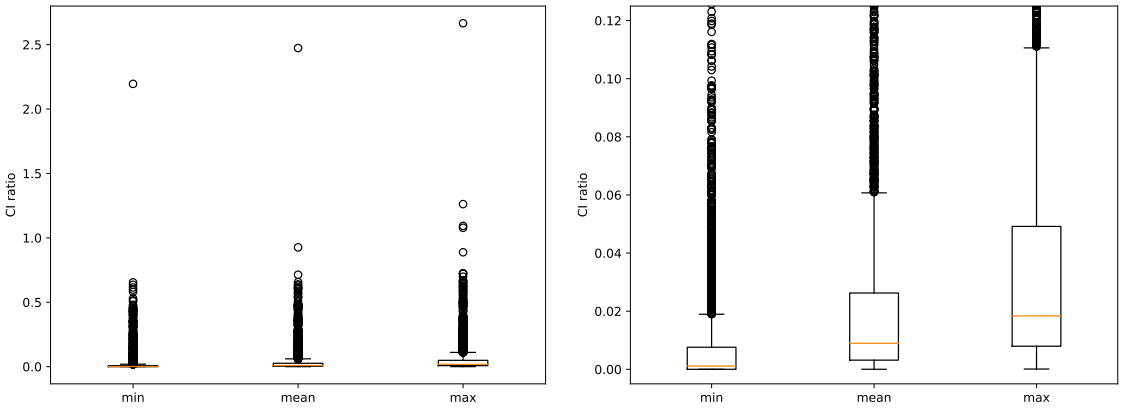
In Figure 5.5, the minimum, mean and maximum CI ratios are visualized. The CoV have a mean change rate of 3.3%±8.2%, a median of 0.9% and a IQR of 2.3%. The average minimum is 1.8%±6.8% and the average maximum is 5.2%±10.3%. The average mean for the CI width stoppage criteria is 1.6%±3.9%, while the average minimum is 0.7%±3.1% and the average maximum is 2.8%±5.3%. The median is 0.6% and the IQR is 1.1%. The divergence have an mean change rate of 2.6%±7.8%, a median of 0.7% and a IQR of 1.7%. The average minimum is 1.5%±5.5% and the average maximum is 4.1%±1.2%. The shift of the CI ratio demonstrates a pattern similar to the change rate. We can conclude that the CoV criteria produces the most different results compared to the standard approach. The CI width stoppage criteria is often a bit better than the divergence analysis.

Figure 5.6 presents the change rate per project and stoppage criteria over all evaluated benchmarks. In the remainder of this section, we primarily analyze the divergence stoppage criteria. The other two stoppage criteria possess a similar characteristic. The CI width usually has a lower change rate, while the CoV criteria is worse. For the divergence, the median change rate over all projects is just 0.6% compared to the mean of 2.4%. Meanwhile, 78.2% of the benchmarks feature a change rate smaller than 2%. However, the remaining benchmarks often possess change rates of over 5%. This is possibly due to the benchmarks not being in a steady state and the warmup phase being stopped too early. These benchmarks explain the high standard deviation of the change rate.

Clear differences can be seen between different projects. The *jmh-jdk-microbenchmarks* and *jmh-core-benchmarks* projects are both written by the OpenJDK development team, but by different authors. We can expect that the OpenJDK team possesses sufficient knowledge on how the JDK internally works and optimizes the execution in order to write good state-of-art JMH benchmarks. However, the third quartile Q_3 from the *jmh-core-benchmarks* project is 0.5%, while for the *jmh-jdk-microbenchmarks* projects, Q_3 is 1.7%. Most benchmarks of the *jmh-core-benchmarks* project are simple, so no complex state objects are involved, and the benchmark method is normally only one line long. Benchmarks of the *jmh-jdk-microbenchmarks* project often use larger state objects and perform complex interactions. A high change rate does not imply that a benchmark is poorly written. Depending on the task performed by a benchmark, the scattering of results is sized differently.

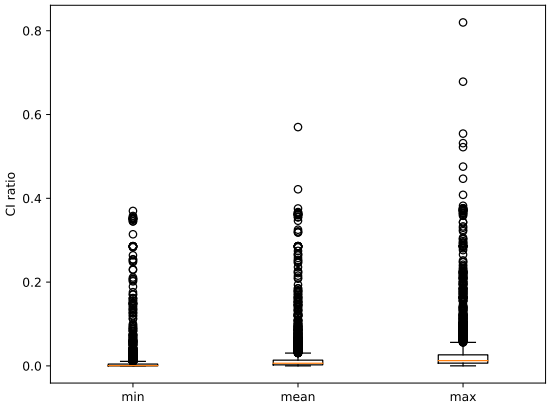
The *JCTools/JCTools* project provides concurrent data structures not offered by the JDK. Literature warns that synchronization and memory sharing can affect the benchmarking performance [25]. Wakart demonstrated that false sharing influences the JMH results [54]. The benchmarks of *JCTools/JCTools* are more variable than the average benchmark. The project with by far the highest change rate is *raphw/byte-buddy*. With 39 benchmarks, the project represents a small test suite. A manual analysis of the results indicates that 18 (46%) of the benchmarks feature a change rate smaller than 2%. We can conclude that, depending on which resources a benchmark is using and the task performed by the benchmark, the result quality differs.

Next, we analyze whether a benchmark's change rate correlates with the number of executed forks. If such a correlation exists, this indicates that dynamically determining the number of forks negatively influences result distribution. For the divergence stoppage criteria, the correlation value equals -0.08 at a significant level of 0.01. We can conclude that dynamically reconfiguring the number of forks exerts no negative effect on the performance result.

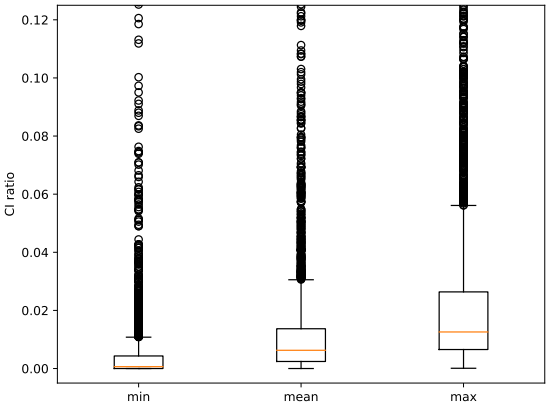


(a) CoV

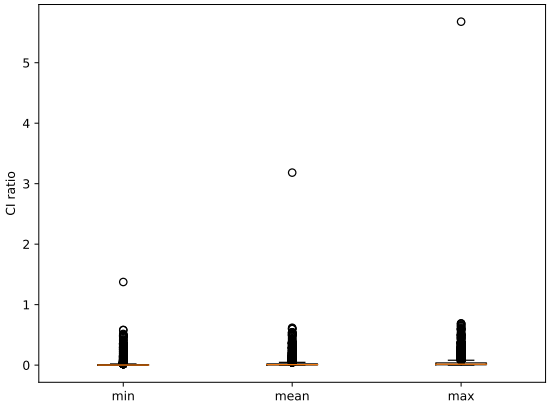
(b) CoV (zoomed)



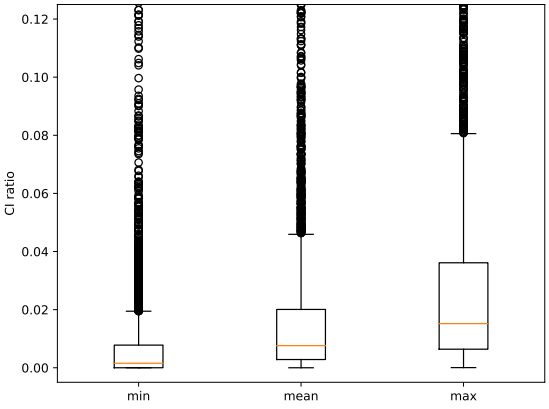
(c) CI width



(d) CI width (zoomed)

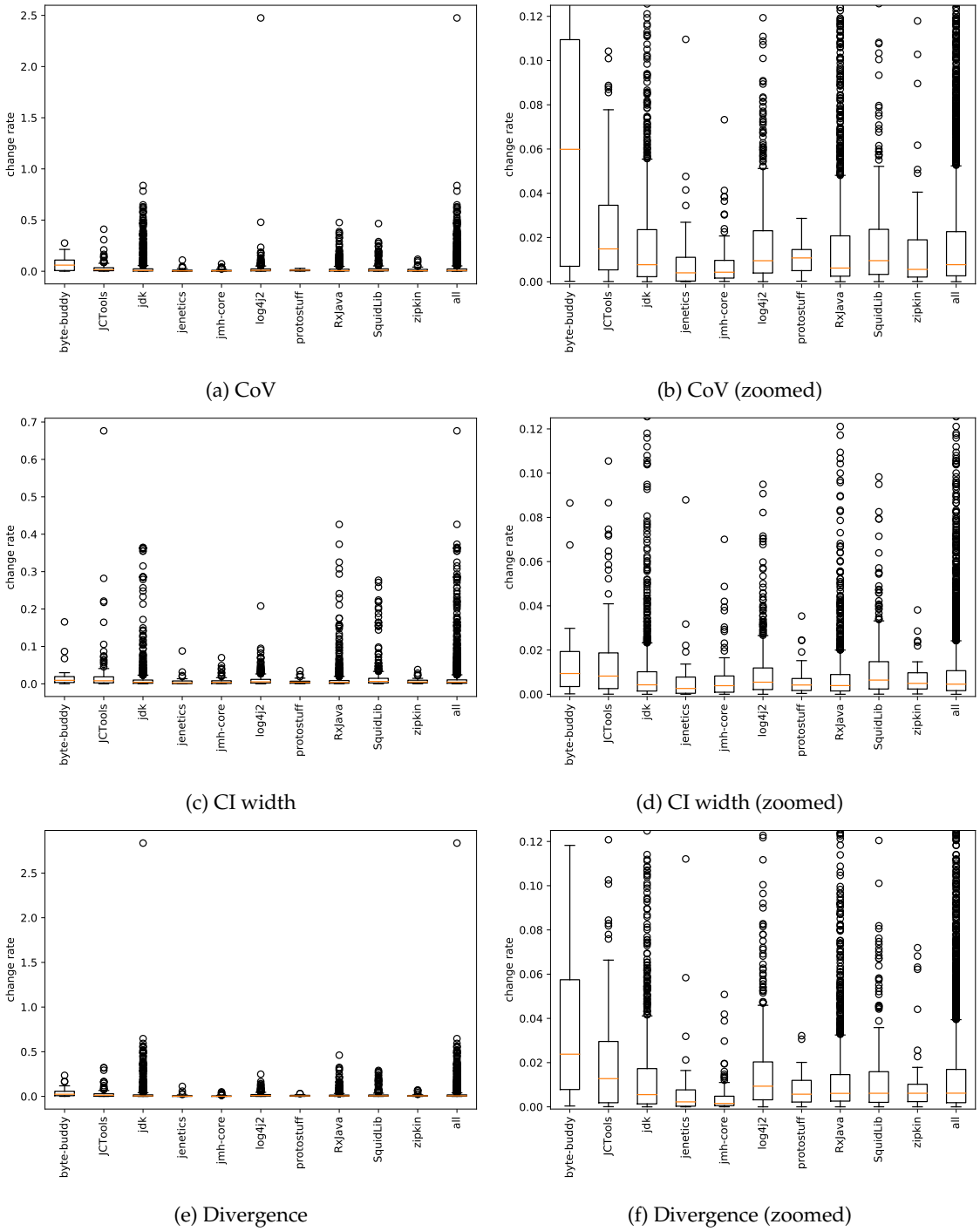


(e) Divergence



(f) Divergence (zoomed)

Figure 5.5: CI ratio shift per stoppage criteria



(a) CoV

(b) CoV (zoomed)

(c) CI width

(d) CI width (zoomed)

(e) Divergence

(f) Divergence (zoomed)

Figure 5.6: Change rate per project and stoppage criteria

Following this, we analyze the correlation between mean invocation time and change rate. If a positive correlation exists, this means that a benchmark with a short invocation time produces more stable results. For the CoV and divergence stoppage criteria, we can detect a weak correlation of 0.20 and 0.25, respectively, at a significant level of 0.01. A part of this can be explained by the fact that, for benchmarks that can only be invoked a few times per second, insufficient data points are available, producing an unstable mean invocation time from which a higher change rate follows.

	CoV	CI width	Divergence
After 50 warmup iterations too variable	1.0%	46.7%	0.8%
After 5 forks too variable	12.0%	37.9%	46.4%

Table 5.5: Reconfigured execution never reaches stable point

Lastly, we analyze how often the statistical evaluation does not stop within 50 iterations. Table 5.5 presents the results. Either the benchmark does not produce stable results, or the stability threshold is too restrictive. On one hand, the divergence and CoV stoppage criteria rarely reach the limit of 50 warmup iterations. In only 1.0% and 0.8% of the benchmarks, respectively, the evaluation revealed that the results are too variable inside a fork. On the other hand, in 46.7% of the benchmarks, the CI width delta is too large. A similar analysis can be conducted to check if, after five forks, the evaluation indicates that more forks are required to obtain a stable result. The results inside the forks are often less variable than between forks. The maximum number of forks is, at five, relatively small. Only one fork that is significantly different blocks the evaluation from stopping earlier. For 12.0% of the benchmarks, the CoV is too large after five forks. The CI width delta is, at 37.9%, still larger than the threshold. The divergence criteria compares the distribution curve. Adding one fork that is significantly a bit different exerts a greater influence on the probability density function than on the mean and standard deviation. Therefore, for 46.4% of the benchmarks, the divergence criteria would like to run more forks.

For most benchmarks, the result quality is not significantly different when using the reconfiguration execution approach. The results of CI width criteria differ the least compared to the standard execution, followed by the divergence stoppage criteria. Overall, the CoV analysis produces the worst results. Depending on the benchmark task, the difference is measurably stronger or weaker.

5.2.3 RQ2.3: Time Saving

First, we wish to measure the performance overhead of the statistical evaluation during the execution. To this end, we execute *apache/logging-log4j2* to approximate the overhead. As baseline, we execute the benchmark with JMH version 1.21 and sample mode. Next, the test suite is executed for each stoppage criteria with the `1.21-reconfigure` version. We measured the end-to-end execution time from the command line on a per-benchmark level. By dividing the end-to-end value with the stoppage criteria by the baseline, we obtain the performance overhead as a percentage. We expect that the performance overhead is similar for all benchmarks, as the number of data points should not strongly influence the computation time of the statistical evaluation. It is important to note that the measured overhead factors are only valid if one-second iterations are used and may differ with other hardware.

Table 5.6 presents the performance overhead o for each stoppage criteria. For all stoppage criteria, the standard deviation is less than 1%. This means that no large difference exists between benchmarks. As such, we can use the measured performance overhead o to estimate the execution time of any benchmark.

CoV	CI width	Divergence
0.88%±0.34%	10.92%±0.63%	4.32%±0.65%

Table 5.6: Performance overhead per stoppage criteria

If we compare the effective execution time of the standard execution with the estimation in Formula 2.3, we obtain the inaccuracy of the approximation. The difference equals 1.8%±2.3%. There are various reasons for this difference: First, the time per iteration is a minimal value. If this limit is reached, no new invocation is started, but the running invocation is not stopped. Second, this time also includes the overhead of the JMH framework for parsing, communication between threads and so on.

We extended the Formula 2.3 to estimate the execution time with the reconfiguration approach. From this, we obtain Formula 5.5. The variable $forks$ contains the set of all executed forks, while wi_f refers to the number of warmup iterations taken in this specific fork.

$$effectiveExecutionTime = \sum_{f \in forks} (1 + o) * wi_f * wt + mi * mt \tag{5.5}$$

As an example using the CoV stoppage criteria, we set o to 0.88%. After two forks, the execution is terminated by the stoppage criteria. In the first fork, 15 warmup iterations are required until the statistical evaluation stops the warmup phase. In the second fork, only nine warmup iterations are performed. The warmup iteration time as well as the measurement iteration time is one-second. In the measurement phase, 10 iterations are executed. Therefore, the effective execution time would be as follows:

$$\big[(1 + 0.0088) * 15 * 1 + 10 * 1\big] + \big[(1 + 0.0088) * 9 * 1 + 10 * 1\big] = 44.21$$

Project	Time saved		
	CoV	CI width	Divergence
byte-buddy	4.42h (81.7%)	2.62h (48.4%)	4.22h (77.8%)
jctools	17.42h (84.8%)	11.45h (55.7%)	17.13h (83.3%)
jmh-jdk-microbenchmarks	157.32h (82.0%)	135.57h (70.7%)	154.41h (80.5%)
jenetics	4.78h (86.0%)	3.37h (60.7%)	4.52h (81.4%)
jmh-core-benchmarks	12.76h (83.5%)	12.69h (83.1%)	12.42h (81.3%)
log4j2	54.56h (77.0%)	39.12h (55.2%)	55.96h (79.0%)
protostuff	3.43h (79.6%)	2.91h (67.7%)	3.44h (79.8%)
rxjava	147.91h (83.1%)	121.55h (68.3%)	138.68h (77.9%)
squidlib	43.07h (84.5%)	30.70h (60.2%)	41.11h (80.7%)
zipkin	6.17h (72.8%)	4.93h (58.2%)	6.59h (77.8%)
Total	**451.84h (82.0%)**	**364.92h (66.2%)**	**438.48h (79.5%)**

Table 5.7: Time saved per project and stoppage criteria

Formula 5.5 enables calculating the execution time with the reconfiguration approach for each stoppage criteria. In Table 5.7, we aggregate this information per project and stoppage criteria. The divergence and CoV saved a similar amount of time with 79.5% and 82.0%, respectively, compared to the standard execution. This means that around one fifth of the initial time is utilized. For both criteria, no large differences are identified between individual projects. The largest outlier is *openzipkin/zipkin*, where the CoV can save only 72.8% of the execution time. With the CI width stoppage criteria, an average of 66.2% of the time can be saved. Additionally, between the projects, large differences exist. For *jmh-core-benchmarks*, over 83.1% of the time be saved, but for

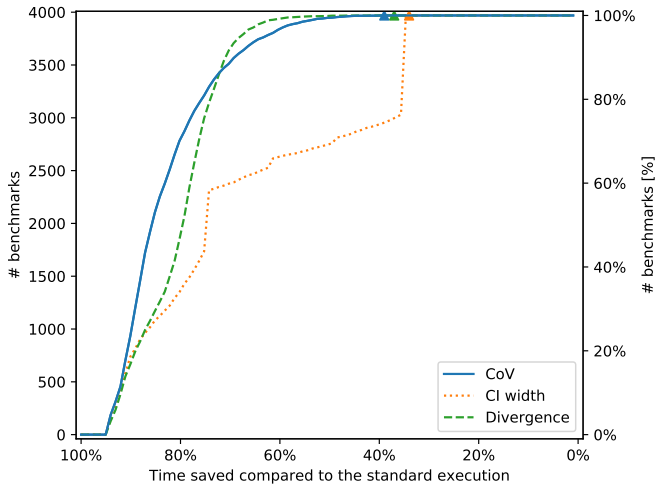


Figure 5.7: Time saved compared to the default execution per stoppage criteria

the *raphw/byte-buddy* project, this equals only 48.4%. The CI width stoppage criteria reacts more quickly to small characteristic changes, and thus takes longer until it stops.

Figure 5.7 illustrates how much time is saved for each stoppage criteria. As an example, 50% or more time is saved using the CI width stoppage criteria for 69% of the benchmarks. If we compare the different curves, we notice the following: For 25% of the benchmarks where the most execution time can be saved, all three criteria perform equally. From now on, the CoV saves a bit more time compared to the divergence criteria. Until 60% of the benchmarks, the CI width criteria is not much worse than the divergence. For the last 40% of the benchmarks, the CI width takes much more time than the other two criteria. The jump by 35% time saved for the CI width occurs because, even if the warmup phase is never stopped earlier in all forks, time is still saved, because we always execute only 10 measurement iterations. The triangle marker in the plot reveals the time saved due to the reduced number of measurement iterations.

	CoV	CI width	Divergence
Average number of warmup iterations	18.5±9.4	34.6 ±16.6	14.1±6.9
Average number of forks	3.1±1.2	3.3± 1.4	4.1±1.2

Table 5.8: Reconfigured execution characteristic

Table 5.8 presents how many iterations and forks on average are performed by each stoppage criteria. The divergence criteria employs the fewest iterations with 14.1. A few more iterations are conducted by the CoV criteria with 18.5. Almost twice as many iterations are performed by the CI width approach with 34.6 iterations. The standard deviation of all three criteria is large, because depending on the benchmark, the number of iterations varies considerably. The number of iterations cannot be considered in isolation. If 50 iterations are necessary per fork, but only two forks are needed, this may be better than five forks using 40 iterations. The CoV and CI width criteria require, on average, 3.1 and 3.3 forks, respectively. It is important to note here that we always execute at least five iterations and two forks. With 4.1 forks, the divergence criteria executes more forks on average.

To conclude, the CoV stoppage criteria demonstrates the smallest overhead, while the CI width possesses the largest one. Depending on the stoppage criteria, between 66.2% and 82.0% of the execution time can be saved using the novel reconfiguration mode in JMH. Often, the number of warmup iterations is reduced more than the number of forks.

Chapter 6

Discussion

In this chapter, the findings of the evaluation are discussed first. In the second segment, the threats of validity and limitations of this thesis are addressed.

6.1 Implications and Main Lessons Learned

This section discusses the behavior patterns for how developers configure their benchmarks and deal with long execution times. Following this, we assess the JMH implementation and its default values, and whether reconfiguration can optimize the result quality execution time trade off. Lastly, we discuss the discovered iteration length phenomenon.

6.1.1 Behavior Patterns

We expected popular projects with numerous contributors to include someone who is an expert in performance engineering. Such a performance engineer knows how to write and configure state-of-the-art benchmarks. The correlation analysis revealed that projects with more stars, forks, contributors and so on do not configure benchmarks differently and use the full power of the JMH features. The question thus becomes why some projects consequently use features such as blackholes and return values, while others do not. An estimated 27.7% of the benchmarks possess neither a non-void return type nor a `Blackhole`. This raises the question of how JVM optimizations are avoided. A further study should investigate how developers learned to write software microbenchmarks and why numerous projects do not use the full power of performance testing frameworks. A possible explanation is that not all tutorials focus on the framework's conceptual details and only explain the basics on how to write benchmarks.

For 24.5% of the benchmarks, the developer did not set any execution configuration. A possible explanation is that they did not know what good configurations are and did not modify the default configuration. Another open question concerns why default values are explicitly set as user-defined values. This can partly be explained by the default configuration update of JMH version 1.21, where previously set execution parameters become useless, as after the update, they are equal to the default value, but the annotations are not removed. A total of 2'268 (16.9%) of the benchmarks set all configuration parameters (except warmup forks) even if some of the parameters were equal to the default value. The advantage of this is that, if JMH updates the default configuration, the execution configuration of these benchmarks will not passively change.

The analysis of the number of forks revealed that, if a user-defined non-default value is used, a developer often decreases the value. The most commonly chosen value is only one fork. However, in the analysis of the reconfiguration approach, one finding was that multiple forks are important

to cover different steady states. This raises the question of whether all developers understand why forking is performed or whether they underrate the importance of multiple forks. We see here a clear difference between the state-of-the-art [13, 21] and the state-of-practice. It must be better explained to practitioners why they should be careful when reducing the number of forks. JMH should add warning messages if the execution configurations contradict best practices.

Table 6.1 lists all projects that use warmup forks. Only eight (1.1%) of the projects feature at least one benchmark running a warmup fork. The projects can be categorized into two groups: The first group comprises one-shot performance tests comparing different implementations of the same functionality. For example, *dryuf/dryuf-concurrent* compares the listable future implementation of the JDK with Guava[1] and Spring[2]. Such projects are written once and not regularly executed. The execution time is not a critical factor as in a continuous integration pipeline. We want to note that *dryuf/dryuf-concurrent*, *kvr000/zbynek-concurrent-pof* and *kvr000/zbynek-java-expimplemented* are all written by the same author. The second group consists of database-related systems seeking to warm up system caches. For the remaining two projects, we cannot explain why warmup forks should be helpful. Further analysis is necessary to investigate whether the benchmarks which perform warmup forks produce better results compared to cases where the same benchmarks are executed without warmup forks. In any case, the JavaDocs and help command description of JMH should be improved, because they do not sufficiently explain why warmup forks are helpful and why they are disabled by default.

Project	# Benchmarks performing warmup forks	Total # benchmarks	Area
coder-press/crazy-soap	2	2	Implementation comparison
dryuf/dryuf-concurrent	50	50	Implementation comparison
kvr000/zbynek-concurrent-pof	44	44	Implementation comparison
kvr000/zbynek-java-exp	6	22	Implementation comparison
hazelcast/hazelcast	4	66	In-memory data grid
spotify/sparkey-java	5	10	Key/value storage library
svenruppert/rapidpm-microservice	2	2	Microservice framework
pdemanget/examples	2	2	Demo project

Table 6.1: Projects which perform warmup forks

The analysis of the benchmark update frequency reveals that execution configurations are chosen once, and then no longer questioned. The method body of a benchmark is also rarely updated. A developer can argue that if a benchmark is properly configured, no configuration change is necessary. However, after the default configuration was changed with JMH version 1.21, we expected most projects check and modify the configurations. We did not see this phenomenon in practice. However, the default execution time per benchmark was increased from 400 to 500 seconds. If a 400-second execution time was sufficient in the past, it is unclear why the benchmark should run longer and waste time. From this, we can conclude that the developers simply updated the JMH dependency without realizing the effect on the test suite execution time and result quality.

It is beyond the scope of this study to examine how regular performance tests are executed. If benchmarks are regularly executed and evaluated, incorrect configuration is more likely to be detected and fixed. For example, the test suite of *kiegroup/kie-benchmarks* features a total execution

[1] `https://github.com/google/guava`
[2] `https://spring.io`

time of 7.4 years. In the survey of Bezemer et al., one third of the participants performed performance testing on at least a weekly basis [7]. An open question concerns how often microbenchmarks are executed. We have doubts that all projects regularly execute their benchmarks and evaluate their results, as some execution configurations are obviously wrong.

To execute the reconfigured benchmarks, we reduced the measurement phase from 50 to 10 seconds, which for most benchmarks ends in a negligible performance change. Therefore, with the reconfigured approach, more warmup time than measurement time is performed. However, in the analysis of RQ1.2, the warmup time took longer than the measurement time in only 17.9% of the benchmarks. Developers tend to neglect the importance of warmup phase.

A number of projects utilized an outdated JMH version. Previous research has indicated a general problem of updating third-party dependencies [46]. An update of the JMH dependency not only influences the library's API; bugs are also fixed, which may affect the performance. If benchmarks are executed on bare-metal machines, the results can be compared to historical executions in the same environment. However, if a dependency update affects the performance, the results after the update can then no longer be compared with historical information that uses an old version. Practitioners solved this problem by skipping some versions, but they occasionally update to the newest version. For example, *apache/logging-log4j2* updated to version 1.1.1 in September 2014. In January 2018, they directly updated to version 1.19, and the last update in July 2019 switched to 1.21. This raises an open questions of how many projects execute their tests on bare-metal machines so that rarely updating JMH makes a difference.

6.1.2 Dealing with Time Intensive Executions

When evaluating the execution time, we noted that if the configuration was changed, it was often not just moderate changes. Either the default values are extremely pessimistic, the developers lack sufficient knowledge to select good configurations, or they realized that executing the entire test suite takes too long, and so they deliberately reduced the parameters. The reconfiguration approach reveals that there remains considerable room to reduce the execution time. However, determining a priori how long the benchmark execution should take is non-trivial. For most benchmarks, the default values are pessimistic and time is wasted. The state of practice analysis revealed that numerous benchmarks reduce the execution time by over a factor of four. However, an execution of one benchmark often still takes more than one minute. Executing the entire project test suite also requires considerable time, and executing them as part of the continuous integration pipeline remains unfeasible.

As mentioned in the previous section, the number of forks represents a popular configuration option employed by developers to reduce the total execution time. JMH supports different benchmark modes to measure performance. As each additional mode increases the total execution time, usually only one mode is employed. Analyzing benchmarks where non-default execution configurations are set, we observe that the median warmup time per fork equals five seconds, while the median measurement time per fork equals eight seconds. To conclude, all configuration parameters are somehow utilized to reduce a benchmark's total execution time. Typically, the number of forks and time per iteration are reduced. However, the combination of all modified execution parameters often ends in a considerably smaller execution time.

6.1.3 Recommendations to Developers

Based on the identified behavior patterns, we would like to remind developers of a few things: First, always avoid JVM optimizations. Use either the `Blackhole` class provided by JMH or the return value of the benchmark method to consume all intermediate results. Multiple forks also represent an important factor for obtaining sound results, as different forks reach different steady

states. RQ2.1 revealed that if a steady state is reached, a short measurement phase is sufficient to obtain a sound result. Updating the JMH dependency makes the comparison with historical execution difficult, as bugs in the framework are fixed, which may influence the performance. Additionally, if a developer updates to a newer JMH version, he should check whether the default execution configuration was changed. If a benchmark does not possess user-defined values, the execution configuration will be passively changed. A developer should also consider whether the old default values should be set as user-defined values, or if the benchmarks should be executed in the future with the new default values. The JMH sample mode offers an advantage of enabling more rigorous performance evaluation. Microbenchmarks are not normally distributed. The performance distribution features more details than the mean and standard deviation and can be used as input for numerous statistical tests.

6.1.4 JMH Implementation and Default Values

We identified 94 benchmarks that run both modes, sample and average time. Technically, both modes work similarly—only the output format differs. Additionally, the throughput can be calculated from the average time, because it is the inverse. A total of 203 benchmarks run the throughput mode first, and afterwards the average and/or sample mode. The problem is that JMH sequentially processes multiple modes. However, if all four modes are selected, only two separate executions are necessary: First, the sample mode from which the throughput and average time can be calculated, and second, the singleshot mode, which measures the cold start performance. We suggest that the JMH implementation should be modified.

With JMH version 1.21, the default number of forks is decreased from 10 to 5 [50]. For most benchmarks, five forks are sufficient. However, we cannot rule out that some benchmarks may produce better results if more than five forks are executed. Before JMH update 1.21, a single fork possessed a warmup time of only 20 seconds. The OpenJDK development team stated two reasons for changing the values with version 1.21 [51]: First, 20 one-second iterations produce noisy data. We can confirm that 20 second warmup times per fork is not sufficient, as demonstrated by the evaluation of the number of warmup iterations required for the reconfiguration approach. Second, some environments feature bad time-to-performance. In some rare cases, one-second iterations are too short due to the large workload. However, this raises the question of whether the default value should thus be modified for some rare cases. The evaluation of the number of measurement iterations revealed that millions of data points are not necessary to achieve a sound result. Therefore, the measurement phase can be shortened. One-second iterations compared to 10-second iterations offer the advantage of enabling more fine-tuned configuration modifications.

6.1.5 Result Quality Execution Time Trade-Off

Choosing good execution parameters is difficult, because developers desire to obtain meaningful results on one hand, but on the other hand, to also not waste time. In the best case, benchmarks are executed until sound results are produced. However, JMH does not provide any mechanism to easily identify good execution configurations. We thus implemented an optimization that applies stoppage criteria during the execution.

If we compare the reconfigured results with the baseline, we can determine that, depending on the necessary result quality, such a reconfiguration is feasible. Depending on the stoppage criteria, the performance difference varies in strength. The CI width possesses the smallest distribution change compared to the default execution. On average, there is a mean change rate of 1.4%, but 66.2% of the time is still saved. We want to note that the utilized bootstrapping simulation was written in Go, and a native Java implementation would require more computation time, because Go code is directly compiled to machine code, which is faster [43].

The CoV and the Kullback-Leibler Divergence saved a similar amount of time, while the divergence results are closer to the results of the standard execution. We do not see any advantages in the CoV stoppage criteria, because the divergence criteria are better across all aspects. The decision of choosing the divergence or CI width stoppage criteria depends on how accurate the result should be.

For the CI width and divergence criteria, in over one third of the benchmarks, the stoppage criteria warns that the stability threshold was not reached after five forks. This indicates that either the benchmark results are not stable or the stoppage criteria already reacts to small characteristic differences. We compute the mean change rate for the CI width criteria and split them into two groups. The first group comprises benchmarks where we receive a warning, and the second group consists of benchmarks that are stable. Stable benchmarks possess a mean change rate of 1.3%±4.0%, while the other benchmarks feature a mean change rate of 1.4%±3.4%. As no large difference exists in the mean, we suppose that the CI width stoppage criteria is sensitive to small characteristic changes and that most benchmarks classified as not stable can be stopped after five forks.

6.1.6 Iteration Length Phenomenon

We investigate the influence of reducing the execution time of a single iteration while simultaneously increasing the number of iterations. Concretely, we compare 10 10-seconds iterations against 100 one-second iterations. The following analysis is based on the *ReactiveX/RxJava* and *apache/logging-log4j2* projects with a single fork.

On one hand, we compared the result quality between the two groups. In the best case, there is no difference. Shorter iterations potentially end in more variable results. The difference between the two CoV is, on average, 0.009±0.082. Over all benchmarks, the results are minimally more stable with shorter iterations. Next, we analyze the CI width divided by the mean with 10'000 bootstrap simulations. The average difference of the width equals 0.026±0.064. Over all benchmarks, the CI width is smaller, but for a single benchmark, the width may grow. However, in only 7.4% of the benchmarks, the CI width is enlarged by at least 0.01 with shorter iterations. Conversely, it follows that in most cases, the width is roughly the same, or the width is smaller. Lastly, we interpret the divergence criteria. We compute the p-value between the 9th and 10th 10-second iteration and between the 90th and 100th one-second iteration. The value distribution should not change, as we are in a steady state, so we expect large p-values and a small difference between the two groups. In 85.9% of the benchmarks, the p-value difference remains less than 0.01. We can conclude that, for most benchmarks, the result quality is on the same level if more but shorter iterations are used. Only for some benchmarks this execution configuration modification negatively influences the result characteristic.

On the other hand, we also compare the benchmark's average invocation time. We expect there to be no difference. For 71.3% of the benchmarks, the average invocation time differs by more than 1%. Even 32.0% of the benchmarks feature a difference exceeding 5%. This behavior is unexpected. We do not have an explanation for why changing the iteration length influences the average invocation time. We can conclude that the average invocation time is significantly changed if the time per iteration is shortened. However, sound results (with another mean) are still produced.

For performance regression testing between two versions of a benchmark, this phenomenon implies that we can only compare versions sharing the same time per iteration. Otherwise, the mean invocation time differs solely because the benchmark execution configuration was changed.

6.2 Threats of Validity

In this section, we discuss various topics that threaten the validity of the results.

6.2.1 Construction Validity

Construction validity concerns the study's construction. In the first part of this work, we sought to analyze as many projects as possible. However, we did not analyze every commit, because such a procedure would be too time consuming. For the historical analysis, we only selected a subset of commits from the default branch in regular intervals. Only one commit per month and project was selected. We do not know what happened between the sampled commits. Therefore, our findings focus on the long-term behavior. Additionally the number of code changes in the historical evaluation does not always represent the effective number, because some projects—for example, *melix/jmh-gradle-plugin* the same fully qualified benchmark name multiple times in different folders with different method hash. In this case, the number of code changes is overestimated. Furthermore, we have not considered that a benchmark can be renamed or moved. For many evaluations, the JMH version of the project is required to determine the default execution configuration. However, the heuristic cannot always extract the version. We often ignored projects where the extraction failed and do not analyze whether a systematic error was produced.

We estimate the execution time of benchmarks in the evaluation. However, the execution time describes a lower-bound limit, because the set time per iteration is the minimum time spent at each iteration [14]. In Section 5.2.3, we analyzed this factor. However, this factor remains roughly the same for most benchmarks and can be ignored. To achieve comparable results in the second part of this thesis, we changed the JMH version of all projects to 1.21. We cannot rule out that the project's development team would modify the benchmarks before updating to a new version.

To evaluate the reconfigured approach, we simply executed each benchmark once with five forks, zero warmup iterations and 100 one-second measurement iterations. Afterward, we constructed the dataset for each stoppage criteria and the baseline case. As such, the number of forks and warmup iterations of a stoppage criteria was calculated and the corresponding data points stored. The constructed data would be similar to a separate execution. However, if a slowdown occurred during the execution, all data sets are affected.

6.2.2 Internal Validity

The internal validity concerns the confidence of the findings. We ignored the case wherein execution configurations are overridden by CLI arguments or user-defined values are set in the code. Potentially, there would be a readme file explaining the required CLI flags. We did not search these. A follow-up study is required to investigate how often CLI execution configuration arguments are set and whether it is a good practice to set configurations via the CLI. For only four benchmarks the execution with the custom configuration takes longer. For most benchmarks we increased the execution time which results in a more sound result. The repository metrics are not always representative of a project's popularity. For example, *apache/logging-log4j2* is only a mirror repository and features a relatively low number of stars and forks. As GitHub is the most popular version control hosting service, it is likely that the majority of projects possess their major code base on GitHub, and the repository metrics offer an effective indicator of project popularity.

We followed the guidelines of Georges et al. to statistically rigorous test the performance of software microbenchmarks [20]. The tests are executed on a bare-metal machine. However, we cannot fully exclude measurement bias. The CIs are approximated via a bootstrapping algorithm. To mitigate Monte-Carlo noise, we utilized 10'000 bootstrap iterations in the evaluation scripts.

However, for the stoppage criteria, 10'000 bootstrap iterations end in an over-large performance overhead, and the sequential testing only performed 1'000 bootstrap iterations.

The modified JMH implementation and the new features in the Bencher tool can generate functional or performance bugs. The core functionality of both is unit tested to address this threat. For the Cliff's Delta effect size, we always applied the threshold proposed by Romano et al. [45]. Some benchmarks may require different stoppage criteria thresholds. A further study should evaluate the end-to-end differences of other thresholds. Additionally, we want to note that the measured performance overhead is hardware dependent. We did not analyze the overhead factor on other machines.

6.2.3 External Validity

External validity concerns the work's generalizability. We only selected open-source projects on GitHub. Further work should analyze industrial projects and other data sources for open-source projects. Previous work has also demonstrated that the majority of projects on GitHub are either personal or inactive [30]. We selected only one repository per project, so we ignored the forks. Otherwise, they would skew the findings. We also did not check whether a forked project modified the execution configuration or implementation, which would be a reason to not exclude them. For RQ2, we evaluated only a subset of all mined projects. We attempted to select an effective mix of application area and project size. Running more benchmarks was infeasible, because the benchmarks possess a long execution time. We focused our analysis on Java projects, as they are long-running [32] and benefit from reducing the execution time. We selected only projects using JMH as a microbenchmarking framework, which is the most popular one for JVM-based languages. However, we should verify whether our findings hold for other programming languages and frameworks. We always used the default Just-In-Time (JIT) compiler and did not analyze other compilers, such as Graal[3]. Additionally, hardware optimizations were not disabled. For the second research question, we excluded some benchmarks, as their execution failed. As only a small number of benchmarks were ignored, we did not analyze whether a systematic error occurred.

6.2.4 Limitations

This study's limitations concern technical details of the implementation. In the first part of this work, we employed a source code parser to investigate the state of practice regarding execution configurations. However, such a parser does not always work perfectly. A main issue is that the parser must resolve the fully qualified names. As the source code must not be syntactically correct, the parsing process may fail. Resolving the name inside the same repository is normally not a problem. However, resolving the fully qualified name in external dependencies requires the dependency as a source code or jar file. We simplified the name resolution in that only names in the same repository or the required JMH names are resolved. If a benchmark uses a state object from an external dependency, we do not resolve it. The source code parser adds a warning message to the log. We want to note that such a case does not occur often during the parsing.

The source code parser does not support all JMH features. For example, if a class called `AbstractStateObject` features a JMH parameter, and a class `ConcreteStateObject` extends the abstract class and possesses a `@State` annotation, the parser thinks that the `ConcreteStateObject` class possesses no JMH parameters, but the JMH framework searches for parameters in the class hierarchies. With the release of version 0.9, some annotations are renamed. In previous versions, the `@Benchmark` annotation was named `@GenerateMicroBenchmark`. The

[3] `https://www.graalvm.org`

source code parser does not support these old names, and such benchmarks are not found. An analysis of the JMH versions reveals that only a few projects possessed an older version as 0.9 as dependency.

The JMH version extractor uses some simple heuristics to extract the JMH version. However, false results are sometimes produced. If a project features sub-modules, and not all modules employ the same JMH version, a wrong version is extracted, because after a valid JMH version was found, the heuristic stops. We assumed that all benchmarks in a project utilize the same JMH version. For example, *alesharik/AlesharikWebServer* have two modules database and api, which use JMH version 1.18, and a one module utils, which uses 1.21. Additionally, Gradle enables declaring dynamic versions such as `1.10.+`, but the heuristic thinks that this is version 1.10. The Java version extractor works similarly to the JMH version extractor. It also faces problems with different modules that have different Java versions defined. For example, in *apache/logging-log4j2*, the benchmarks are in a sub-module utilizing Java 9 as the target version, but other sub-modules feature a lower target version. All extractors face some problems with comments. For example, in the gradle file of *puniverse/quasar*[4], Java 10 is set as target version, but on the same line, a comment contains 11 as a character sequence, so the heuristic thinks that the project runs Java 11.

[4] `https://github.com/puniverse/quasar/blob/master/build.gradle`

Chapter 7

Closing Remarks

This chapter summarizes this study's contribution and provides a brief overview of the findings. Moreover, future work is briefly explained that has not been addressed so far.

7.1 Conclusion

This thesis focused on the research gap regarding how developers configure software microbenchmarks and whether reconfiguring software microbenchmarks can reduce the execution time while maintaining result quality at the same level. In chapter 4, we extracted the execution configuration from 753 open-source projects on GitHub that use JMH as a performance-testing framework. From the configuration, we computed the resulting execution time. In order to answer the first research question, we answered the following sub-questions:

RQ1.1: Which custom software microbenchmark configurations are defined by developers?

Depending on the execution parameter, the developer does not override the default execution parameter between 55.2% and 98.7% of the time. For 24.5% of the benchmarks, not even one of seven execution parameters changed. In some cases, developers explicitly set the default values as user-defined values. If developers modify the number of forks, often just one fork is chosen. We did not find any strong correlation between the repository metrics and how developers configure benchmarks.

RQ1.2: How do custom configurations affect benchmark execution time?

If the default execution configuration is not used, the benchmarks' initial execution time is reduced for over 94.6% of the benchmarks. For over 77.8% of the benchmarks, the execution time is reduced by over 75%. The proportion between warmup and measurement time remains unchanged in 46.8% of the benchmarks. However, 35.4% of the benchmarks spend more time in the measurement phase than in the warmup phase. More popular projects reduce the execution time only slightly less.

RQ1.3: How long does it take to run the full benchmark suites of open-source projects?

For over 37.3% of the test suites, it take longer than an hour to execute all benchmarks of a project. One reason is that some benchmarks possess parameterization combinations. The median execution time for the projects equals 26.7 minutes. Compared to the project test suite execution with the default configuration, the execution time is reduced by more than a factor of five.

RQ1.4: How often are execution configurations modified?

Once a benchmark is written, neither the benchmark configuration nor the implementation are often modified. Only for 1.1% of the benchmarks, we detected an execution configuration change over the project history. Projects that updated to the newest JMH version 1.21, where the default values are modified, also rarely changed the execution configuration.

RQ1: How are software microbenchmarks configured and what are their resulting execution times?

Typically, benchmarks are written once and then never updated. Many benchmarks reuse some of the proposed default parameters. If the execution configuration is modified, this often results in a lower execution time. However, most project test suites still need several minutes or hours to execute all benchmarks.

In the second part of this report, we evaluated a statistics-based performance testing concept for software microbenchmarks. Compared to the standard execution, we applied sequential testing during the execution and decided whether a sound result was produced and if the execution could be stopped. To evaluate the usefulness of this reconfiguration approach, we focused on the following aspects:

RQ2.1: How much does the length of the measurement phase matter after a steady state is reached?

For most benchmarks, reducing the length of the measurement phase ends in an insignificant change compared to the execution using the default configuration if the benchmark was previously warmed up. Multiple forks help to reduce the number of significant differences. A measurement phase lasting longer than 10 seconds per fork usually does not produce significantly better results.

RQ2.2: How does reconfiguration of software microbenchmarks affect the execution result?

Depending on the stoppage criteria, the characteristic difference varies in strength. The CoV always produces the worse results. The CI width criteria is often a bit better than the divergence. Even if an A/A test detects a significant difference, for most benchmarks, the mean shift is smaller than 2%. Meanwhile, 20% of the benchmarks are not just slightly negatively affected by the reconfiguration, and the mean shift is often over 5%.

RQ2.3: How much time can be saved with dynamic reconfiguration of software microbenchmarks?

Depending on the stoppage criteria, between 66.2% and 82.0% of the time can be saved using the proposed reconfiguration approach. The performance overhead of the statistical evaluation ranges between 0.88% and 10.92%. The gain in time from skipping iterations and forks is larger than the time needed to calculate the stoppage criteria.

RQ2: Can dynamic reconfiguration of software microbenchmarks reduce the execution time without sacrificing result quality?

Our analysis indicated that it is possible to reduce execution time by reconfiguring the execution configuration of software microbenchmarks. Depending of the stoppage criteria, the execution time can be reduced up to 82.0%. For most benchmarks ($\sim$80%), the performance distribution is only slightly changed, which is often insignificant. For the remaining benchmarks, the reconfiguration can result in significantly different results.

Additionally, we extended the Bencher tool that, on one hand, now provides a source code parser, and on the other hand, offers more features, such as supporting parsing state objects. We modified the JMH implementation and added a new benchmark mode called "reconfigure," which supports the three stoppage criteria CoV, CI width and Kullback-Leibler divergence. The source code of the implementations and evaluation plus the intermediate result files of the evaluation are publicly available on GitHub[1][2][3].

We can conclude that developers do not configure benchmarks as proposed by the default JMH values. Often, the long execution time is reduced. The reconfiguration of software microbenchmarks reduces the execution time while maintaining similar result quality. Such an approach simplifies the configuration of software microbenchmarks, because a developer simply chooses the reconfiguration mode and the tool itself dynamically optimizes the execution configuration. Practitioners can use the results of our study to better configure benchmarks, such as not reducing the number of forks or reducing the length of the measurement phase more than the warmup phase. Framework developers on one hand possess an overview where they can add warning messages to check the execution configurations. On the other hand, we proposed a new benchmark mode that they can support.

7.2 Future Work

This section elaborates on possible future work. Our entire evaluation is only performed on software microbenchmarks written with JMH. Other benchmarking implementations, such as the framework in the Go standard library[4], do not support the same configuration options. However, the reconfiguration approach describes an abstract concept that can also be implemented with slight modifications in other performance testing frameworks and is not limited to JVM-based languages.

Our state of practice analysis is only conducted on source code, and developers are not interviewed. We could only analyze the results and do not know the decision behind the chosen execution configuration. A developer survey helps to understand why a strange behavior occurs and how performance testing frameworks should be modified so that developers can more easily utilize their full power.

During the analysis of the reconfigured benchmarks, we realized that some projects employ parameterization combinations which produce results similar to other executions. Therefore, the question becomes whether a tool can, based on historical information, decide which parameterization combinations of a benchmark should be executed. If two parameterization combinations of a benchmark share the same invocation time and a similar result characteristic, we gain no additional information from executing both combinations; it is sufficient to simply execute one of them.

We executed the reconfigured software microbenchmarks only in a controlled bare-metal environment. An open question is thus raised as to whether such an approach is also applicable on virtualized resources, such as public clouds. Furthermore, it is unclear whether the same stoppage criteria threshold can be used, or if they are too strict and the benchmarks will never stop.

During the analysis of the reconfiguration approach, we executed all benchmarks of the selected projects[5]. We never verified whether the benchmarks are well written and feature no bad practices, as defined by Costa et al. [13]. A possible reason why some benchmarks are not stable after 50 warmup iterations could be that they never reached a steady state.

[1] `https://github.com/stewue/masterthesis-evaluation`
[2] `https://github.com/chrstphlbr/bencher`
[3] `https://github.com/stewue/jmh`
[4] `https://golang.org/pkg/testing`
[5] Except failing benchmarks (see Appendix C)

A possible improvement of our approach would be to perform a sensitivity analysis. In this analysis, a configuration parameter such as the number of measurement iterations or the stoppage criteria threshold should be modified, and the end-to-end change should be evaluated in terms of result quality and execution time. As an example, this could explore how would the mean change rate and the two A/A tests react if the number of measurement iterations was reduced or increased.

Another option would be to test other stoppage criteria or modify the existing stoppage criteria. For example, the CI width metric is currently calculated over all executed iterations in the current fork. Better results might be produced if only the last 10 iterations are considered.

We revealed that the iteration length influences the average invocation time. However, we did not investigate the reason behind this behavior. We also did not determine whether we can reproduce the same phenomenon with other performance testing frameworks and programming languages, or if this represents a JMH-specific behavior.

Appendix A

Acronyms

API Application Programming Interface

CI Confidence Interval

CLI Command Line Interface

CoV Coefficient of Variation

CPU Central Processing Unit

CSV Comma Separated Values

IDE Integrated Development Environment

IQR Interquartile Range

JDK Java Development Kit

JIT Just-In-Time

JMH Java Microbenchmark Harness

JSON JavaScript Object Notation

JVM Java Virtual Machine

REST Representational State Transfer

URL Uniform Resource Locator

VM Virtual Machine

XML Extensible Markup Language

Appendix B

Bibliography

All the following link and links in the footnotes were verified on 23 January 2020.

[1] H. Abdi. Coefficient of variation. *Encyclopedia of research design*, 1:169–171, 2010. doi:10.4135/9781412961288.n56.

[2] A. Abedi and T. Brecht. Conducting repeatable experiments in highly variable cloud computing environments. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ICPE '17, pages 287–292, New York, NY, USA, 2017. ACM. doi:10.1145/3030207.3030229.

[3] H. M. Alghmadi, M. D. Syer, W. Shang, and A. E. Hassan. An automated approach for recommending when to stop performance tests. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 279–289, Oct 2016. doi:10.1109/ICSME.2016.46.

[4] M. M. Arif, W. Shang, and E. Shihab. Empirical study on the discrepancy between performance testing results from virtual and physical environments. *Empirical Software Engineering*, 23(3):1490–1518, Jun 2018. doi:10.1007/s10664-017-9553-x.

[5] E. Bakshy and E. Frachtenberg. Design and analysis of benchmarking experiments for distributed internet services. In *Proceedings of the 24th International Conference on World Wide Web*, WWW '15, pages 108–118, Republic and Canton of Geneva, Switzerland, 2015. International World Wide Web Conferences Steering Committee. doi:10.1145/2736277.2741082.

[6] B. Beizer. *Software Testing Techniques*. John Wiley & Sons, Inc., New York, NY, USA, 1990. ISBN: 978-8177222609.

[7] C.-P. Bezemer, S. Eismann, V. Ferme, J. Grohmann, R. Heinrich, P. Jamshidi, W. Shang, A. van Hoorn, M. Villavicencio, J. Walter, and F. Willnecker. How is performance addressed in devops? In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, ICPE '19, pages 45–50, New York, NY, USA, 2019. ACM. doi:10.1145/3297663.3309672.

[8] L. Bulej, V. Horký, and P. Tůma. Do we teach useful statistics for performance evaluation? In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, ICPE '17 Companion, pages 185–189, New York, NY, USA, 2017. ACM. doi:10.1145/3053600.3053638.

[9] J. Cito, P. Leitner, T. Fritz, and H. C. Gall. The making of cloud applications: An empirical study on software development for the cloud. In *Proceedings of the 2015 10th Joint Meeting*

on Foundations of Software Engineering, ESEC/FSE 2015, pages 393–403, New York, NY, USA, 2015. ACM. doi:10.1145/2786805.2786826.

[10] N. Cliff. *Ordinal methods for behavioral data analysis*. Psychology Press, 1996. ISBN: 978-0805813333.

[11] J. Cox, E. Bouwers, M. v. Eekelen, and J. Visser. Measuring dependency freshness in software systems. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 109–118, May 2015. doi:10.1109/ICSE.2015.140.

[12] C. Curtsinger and E. D. Berger. Stabilizer: Statistically sound performance evaluation. *SIGARCH Comput. Archit. News*, 41(1):219–228, Mar. 2013. doi:10.1145/2490301.2451141.

[13] D. E. Damasceno Costa, C. Bezemer, P. Leitner, and A. Andrzejak. What's wrong with my benchmark results? studying bad practices in jmh benchmarks. *IEEE Transactions on Software Engineering*, pages 1–1, 2019. doi:10.1109/TSE.2019.2925345.

[14] O. development team. Jmh command line help, 2019.

[15] P. M. Duvall, S. Matyas, and A. Glover. *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007. ISBN: 978-0321336385.

[16] T. Dybå, V. B. Kampenes, and D. I. Sjøberg. A systematic review of statistical power in software engineering experiments. *Information and Software Technology*, 48(8):745 – 755, 2006. doi:10.1016/j.infsof.2005.08.009.

[17] M. Fagerström, E. E. Ismail, G. Liebel, R. Guliani, F. Larsson, K. Nordling, E. Knauss, and P. Pelliccione. Verdict machinery: On the need to automatically make sense of test results. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 225–234, New York, NY, USA, 2016. ACM. doi:10.1145/2931037.2931064.

[18] K. C. Foo, Z. M. J. Jiang, B. Adams, A. E. Hassan, Y. Zou, and P. Flora. An industrial case study on the automated detection of performance regressions in heterogeneous environments. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ICSE '15, pages 159–168, Piscataway, NJ, USA, 2015. IEEE Press. doi:10.1109/ICSE.2015.144.

[19] R. W. Frick. A better stopping rule for conventional statistical tests. *Behavior Research Methods, Instruments, & Computers*, 30(4):690–697, Dec 1998. doi:10.3758/BF03209488.

[20] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous java performance evaluation. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 57–76, New York, NY, USA, 2007. ACM. doi:10.1145/1297027.1297033.

[21] J. Y. Gil, K. Lenz, and Y. Shimron. A microbenchmark case study and lessons learned. In *Proceedings of the Compilation of the Co-located Workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, & VMIL'11*, SPLASH '11 Workshops, pages 297–308, New York, NY, USA, 2011. ACM. doi:10.1145/2095050.2095100.

[22] A. S. Harji, P. A. Buhr, and T. Brecht. Our troubles with linux and why you should care. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, APSys '11, pages 2:1–2:5, New York, NY, USA, 2011. ACM. doi:10.1145/2103799.2103802.

[23] S. He, G. Manns, J. Saunders, W. Wang, L. Pollock, and M. L. Soffa. A statistics-based performance testing methodology for cloud applications. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, pages 188–199, New York, NY, USA, 2019. ACM. doi:10.1145/3338906.3338912.

[24] T. C. Hesterberg. What teachers should know about the bootstrap: Resampling in the undergraduate statistics curriculum. *The American Statistician*, 69(4):371–386, 2015. doi:10.1080/00031305.2015.1089789.

[25] V. Horký, P. Libič, A. Steinhauser, and P. Tůma. Dos and don'ts of conducting performance measurements in java. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ICPE '15, pages 337–340, New York, NY, USA, 2015. ACM. doi:10.1145/2668930.2688820.

[26] G. Jay, J. Hale, R. Smith, D. Hale, N. Kraft, and C. Ward. Cyclomatic complexity and lines of code: Empirical evidence of a stable linear relationship. *JSEA*, 2:137–143, 01 2009. doi:10.4236/jsea.2009.23020.

[27] J. Jenkov. Jmh - java microbenchmark harness, 2015. `http://tutorials.jenkov.com/java-performance/jmh.html`.

[28] T. Kalibera and R. Jones. Quantifying performance changes with effect size confidence intervals. Technical Report 4–12, University of Kent, June 2012. `http://www.cs.kent.ac.uk/pubs/2012/3233`.

[29] T. Kalibera and R. Jones. Rigorous benchmarking in reasonable time. *SIGPLAN Not.*, 48(11):63–74, June 2013. doi:10.1145/2555670.2464160.

[30] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian. The promises and perils of mining github. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 92–101, New York, NY, USA, 2014. ACM. doi:10.1145/2597073.2597074.

[31] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue. Do developers update their library dependencies? *Empirical Software Engineering*, 23(1):384–417, Feb 2018. doi:10.1007/s10664-017-9521-5.

[32] 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*, MSR '18, pages 119–130, New York, NY, USA, 2018. ACM. doi:10.1145/3196398.3196407.

[33] C. Laaber, J. Scheuner, and P. Leitner. Software microbenchmarking in the cloud. how bad is it really? *Empirical Software Engineering*, 24(4):2469–2508, Aug 2019. doi:10.1007/s10664-019-09681-1.

[34] P. Leitner and C.-P. Bezemer. An exploratory study of the state of practice of performance testing in java-based open source projects. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ICPE '17, pages 373–384, New York, NY, USA, 2017. ACM. doi:10.1145/3030207.3030213.

[35] P. Leitner and J. Cito. Patterns in the chaos—a study of performance variation and predictability in public iaas clouds. *ACM Trans. Internet Technol.*, 16(3):15:1–15:23, Apr. 2016. doi:10.1145/2885497.

[36] Q. Luo, D. Poshyvanyk, A. Nair, and M. Grechanik. Forepost: A tool for detecting performance problems with feedback-driven learning software testing. In *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, pages 593–596, New York, NY, USA, 2016. ACM. doi:10.1145/2889160.2889164.

[37] T. Mytkowicz, A. Diwan, M. Hauswirth, and Sweeney. Producing wrong data without doing anything obviously wrong!, 2016. http://aftermath.rocks/2016/04/11/wrong_data/.

[38] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Producing wrong data without doing anything obviously wrong! *SIGARCH Comput. Archit. News*, 37(1):265–276, Mar. 2009. doi:10.1145/2528521.1508275.

[39] A. B. D. Oliveira, S. Fischmeister, A. Diwan, M. Hauswirth, and P. F. Sweeney. Perphecy: Performance regression test selection made simple but effective. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 103–113, March 2017. doi:10.1109/ICST.2017.17.

[40] Oracle. Java platform, standard edition tools reference - javac, 2019. https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javac.html.

[41] Oracle. Oracle java se support roadmap, 2019. https://www.oracle.com/technetwork/java/java-se-support-roadmap.html.

[42] J. Ponge. Avoiding benchmarking pitfalls on the jvm. 2014. https://www.oracle.com/technical-resources/articles/java/architect-benchmarking.html.

[43] S. Radadiya. A simple performance test and difference: Go v/s java, 2019. https://medium.com/@radadiyasunny970/a-simple-performance-test-and-difference-go-v-s-java-e6f29ad65293.

[44] M. Rodriguez-Cancio, B. Combemale, and B. Baudry. Automatic microbenchmark generation to prevent dead code elimination and constant folding. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 132–143, New York, NY, USA, 2016. ACM. doi:10.1145/2970276.2970346.

[45] J. Romano, J. D. Kromrey, J. Coraggio, J. Skowronek, and L. Devine. Exploring methods for evaluating group differences on the nsse and other surveys: Are the t-test and cohen'sd indices the most appropriate choices. In *annual meeting of the Southern Association for Institutional Research*, pages 1–51. Citeseer, 2006.

[46] A. A. Sawant and A. Bacchelli. fine-grape: fine-grained api usage extractor – an approach and dataset to investigate api usage. *Empirical Software Engineering*, 22(3):1348–1371, Jun 2017. doi:10.1007/s10664-016-9444-6.

[47] M. Selakovic and M. Pradel. Performance issues and optimizations in javascript: An empirical study. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 61–72, New York, NY, USA, 2016. ACM. doi:10.1145/2884781.2884829.

[48] A. Shipilev. Java microbenchmark harness (the lesser of two evils), 2013. https://shipilev.net/talks/devoxx-Nov2013-benchmarking.pdf.

[49] A. Shipilev. Nanotrusting the nanotime, 2014. https://shipilev.net/blog/2014/nanotrusting-nanotime/.

[50] A. Shipilev. Reconsider defaults for fork count, 2018. https://bugs.openjdk.java.net/browse/CODETOOLS-7902170.

[51] A. Shipilev. Reconsider defaults for warmup and measurement iteration counts, durations, 2018. https://bugs.openjdk.java.net/browse/CODETOOLS-7902165.

[52] 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*, ICPE '17, pages 401–412, New York, NY, USA, 2017. ACM. doi:10.1145/3030207.3030226.

[53] N. Tillmann and W. Schulte. Parameterized unit tests. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 253–262, New York, NY, USA, 2005. ACM. doi:10.1145/1081706.1081749.

[54] N. Wakart. Using jmh to benchmark multi-threaded code, 2013. http://psy-lob-saw.blogspot.com/2013/05/using-jmh-to-benchmark-multi-threaded.html/.

[55] A. Wald. Sequential tests of statistical hypotheses. *The annals of mathematical statistics*, 16(2):117–186, 1945. ISBN: 978-1114632806.

[56] E. J. Weyuker and F. I. Vokolos. Experience with performance testing of software systems: issues, an approach, and case study. *IEEE Transactions on Software Engineering*, 26(12):1147–1156, Dec 2000. doi:10.1109/32.888628.

Appendix C

Ignored Benchmarks

Table C.1 contains all benchmarks which are excluded from the evaluation of RQ2. The reason column summarizes why a specific benchmark was skipped. If a benchmark does not have any JMH parameters, the JMH parameter column is empty. Else it is stated which parameterization combinations were skipped.

Project	Benchmark	JMH parameter	Reason
apache/logging-log4j2	org.apache.logging.log4j.perf.jmh .Log4j2AppenderComparisonBenchmark.appenderMMap		Execution stucks
apache/logging-log4j2	org.apache.logging.log4j.perf.jmh .Log4j2AppenderComparisonBenchmark.end2endMMap		Execution stucks
apache/logging-log4j2	org.apache.logging.log4j.perf.jmh.ReflectionBenchmark .test05_getStackTraceClassForClassName		Fails if more than 16 iterations are executed
jenetics/jenetics	io.jenetics.util.RandomEnginePerf.Base.nextInt		Benchmarks defined in abstract class
jenetics/jenetics	io.jenetics.util.RandomEnginePerf.Base.nextLong		Benchmarks defined in abstract class
jenetics/jenetics	io.jenetics.util.RandomEnginePerf.Base.nextFloat		Benchmarks defined in abstract class
jenetics/jenetics	io.jenetics.util.RandomEnginePerf.Base.nextDouble		Benchmarks defined in abstract class
openzipkin/zipkin	zipkin2.SpanBenchmarks.deserialize_kryo		Execution stucks
openzipkin/zipkin	zipkin2.SpanBenchmarks.serialize_kryo		`com.esotericsoftware.kryo.KryoException`
openzipkin/zipkin	zipkin2.internal.WriteBufferBenchmarks.writeVarint_32		`java.lang.ArrayOutOfBoundsExeption`
openzipkin/zipkin	zipkin2.internal.WriteBufferBenchmarks.writeVarint_64		`java.lang.ArrayOutOfBoundsExeption`
openzipkin/zipkin	zipkin2.internal.WriteBufferBenchmarks.writeLongLe		`java.lang.ArrayOutOfBoundsExeption`
openzipkin/zipkin	zipkin2.collector.MetricsBenchmarks .incrementBytes_shortSpans_Actuate		`java.lang.IllegalStateException`
openzipkin/zipkin	zipkin2.collector.MetricsBenchmarks .incrementBytes_mediumSpans_Actuate		`java.lang.IllegalStateException`
openzipkin/zipkin	zipkin2.collector.MetricsBenchmarks .incrementBytes_longSpans_Actuate		`java.lang.IllegalStateException`
openzipkin/zipkin	zipkin2.codec.ProtoCodecBenchmarks .bytebuffer_zipkinDecoder		`java.lang.NoSuchMethodError`
ReactiveX/RxJava	io.reactivex.FlowableFlatMapCompletableAsyncPerf .flatMapCompletable	all parameterization combinations	Execution of a single invocation takes over 60 seconds
ReactiveX/RxJava	io.reactivex.FlatMapJustPerf.flowable	times=1	`java.lang.NullPointerException`
ReactiveX/RxJava	io.reactivex.FlatMapJustPerf.observable	times=1	`java.lang.NullPointerException`

Table C.1: Ignored Benchmarks for RQ2

Appendix D

CD-ROM Content

- **master_thesis.pdf**
  Master Thesis as PDF
- **abstract.txt**
  Abstract in English
- **zusfsg.txt**
  Abstract in German
- **git-bencher/**
  Copy of the Bencher git repository[1]
- **git-jmh/**
  Copy of the JMH git repository[2]
- **git-evaluation/**
  Copy of the git repository which contains the evaluation data and scripts[3]
- **raw-data/**
  Raw data of RQ2
- **meetings/**
  Meeting protocols

[1] https://github.com/chrstphlbr/bencher
[2] https://github.com/stewue/jmh
[3] https://github.com/stewue/masterthesis-evaluation