Dynamically Reconfiguring Software Microbenchmarks: Reducing Execution Time Without Sacrificing Result Quality

Christoph Laaber
University of Zurich
Zurich, Switzerland
laaber@ifi.uzh.ch

Stefan Würsten
University of Zurich
Zurich, Switzerland
stefan.wuersten@uzh.ch

Harald C. Gall
University of Zurich
Zurich, Switzerland
gall@ifi.uzh.ch

Philipp Leitner
Chalmers | University of Gothenburg
Gothenburg, Sweden
philipp.leitner@chalmers.se

ABSTRACT

Executing software microbenchmarks, a form of small-scale performance tests predominantly used for libraries and frameworks, is a costly endeavor. Full benchmark suites take up to multiple hours or days to execute, rendering frequent checks, e.g., as part of continuous integration (CI), infeasible. However, altering benchmark configurations to reduce execution time without considering the impact on result quality can lead to benchmark results that are not representative of the software's true performance. We propose the first technique to dynamically stop software microbenchmark executions when their results are sufficiently stable. Our approach implements three statistical stoppage criteria and is capable of reducing Java Microbenchmark Harness (JMH) suite execution times by 48.4% to 86.0%. At the same time it retains the same result quality for 78.8% to 87.6% of the benchmarks, compared to executing the suite for the default duration. It does not require developers to manually craft custom benchmark configurations; instead it provides automated mechanisms for dynamic reconfiguration. Hence, making dynamic reconfiguration highly effective and efficient, potentially paving the way to inclusion of JMH performance tests in CI.

KEYWORDS

performance testing, software benchmarking, JMH, configuration

ACM Reference Format:
Christoph Laaber, Stefan Würsten, Harald C. Gall, and Philipp Leitner. 2020. Dynamically Reconfiguring Software Microbenchmarks: Reducing Execution Time Without Sacrificing Result Quality. In *Proceedings of The 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*. ACM, New York, NY, USA, 12 pages. https://doi.org/doi

1 INTRODUCTION

Performance testing enables automated assessment of software performance in the hope of catching degradations, such as slowdowns, in a timely manner. A variety of techniques exist, spanning from system-scale (e.g., load testing) to method or statement level, such as software microbenchmarking. For functional testing, CI has been a revelation, where (unit) tests are regularly executed to detect functional regressions as early as possible [20]. However, performance testing is not yet standard CI practice, although there would be a need for it [4, 33]. A major reason for not running performance tests on every commit is their long runtimes, often consuming multiple hours to days [22, 24, 30]. To lower the time spent in performance testing activities, previous research applied techniques to select which commits to test [22, 42] or which tests to run [2, 12], to prioritize tests that are more likely to expose slowdowns [36], and to stop load tests once they become repetitive [1] or do not improve result accuracy [18]. However, none of these approaches are tailored to and consider characteristics of software microbenchmarks *and* enable running full benchmark suites, reduce the overall runtime, while still maintaining the same result quality.

In this paper, we present the first approach to dynamically, i.e., during execution, decide when to stop the execution of software microbenchmarks. Our approach —dynamic reconfiguration— determines at different checkpoints whether a benchmark execution is stable and if more executions are unlikely to improve the result accuracy. It builds on the concepts introduced by He et al. [18], applies them to software microbenchmarks, and generalizes the approach for any kind of stoppage criteria.

To evaluate whether dynamic reconfiguration enables reducing execution time without sacrificing quality, we perform a case study evaluation on ten Java open-source software (OSS) projects with benchmark suite sizes between 16 and 995 individual benchmarks, ranging from 4.31 to 191.81 hours. Our empirical evaluation comprises of three different stoppage criteria, including the one from He et al. [18]. It assesses (1) whether dynamically reconfigured benchmarks maintain their result quality compared to the default JMH execution configuration and (2) how much time can be saved by employing dynamic reconfiguration.

We find that for the majority of studied benchmarks, the result quality remains the same after applying our approach. Depending on the stoppage criteria, between 78.8% and 87.6% of the benchmarks *do not* produce different results, with an average performance change rate between 1.4% and 3.1%. Even though computation of the stoppage criteria introduces an overhead between <1% and ~11%, dynamic reconfiguration still enables saving a total of 66.2% to 82%

ESEC/FSE 2020, 8 - 13 November, 2020, Sacramento, California, United States

of the execution time across all projects. For individual projects, benchmark suites take 48.4% to 86.0% less time to execute. Our empirical results suppport that dynamic reconfiguration of software microbenchmarks is highly effective and efficient in reducing execution time without sacrificing result quality.

Contributions. The main contributions of our study are:

- We present the first approach to dynamically stop the execution of software microbenchmark using three different stoppage criteria.
- We provide empirical evidence that demonstrates the effectiveness and efficiency of dynamic reconfiguration for ten case study OSS applications.
- We provide a fork of JMH that implements our approach as part of our replication package [32].
- To investigate whether real-world benchmark suites could benefit from our approach to save time, we collect the largest data set of JMH OSS projects (753 projects with 13'387 benchmarks) including extracted source code properties such as benchmark configurations and parameters.

2 JAVA MICROBENCHMARK HARNESS (JMH)

JMH[1] is the de-facto standard framework for writing and executing software microbenchmarks (in the following simply called benchmarks) for Java. Benchmarks operate on the same level of granularity as unit tests, i.e., statement/method level, and are similarly defined in code and configured through annotations. Different from unit tests where the outcome is binary, i.e., a test passes or fails (disregarding flakiness), benchmarks produce outputs for a certain performance metric, such as execution time or throughput. As these performance metrics are easily affected by confounding factors, such as the computer's hardware and software, background process, or even temperature, one must execute benchmarks repeatedly to obtain rigorous results that are representative of the software's true performance [16].

Figure 1 depicts a standard execution of a JMH benchmark suite B, where benchmarks b are sequentially scheduled. Every benchmark execution starts with a number of warmup forks wf, to bring the system into a steady state, whose results are discarded. A fork is JMH parlance for running a set of measurements in a fresh Java Virtual Machine (JVM). The warmup forks are followed by a number of measurement forks f (often simply called forks). Due to dynamic compilation, every fork is brought into steady state by running a series of warmup iterations wi, after which a series of measurement iterations mi are executed. An iteration has a specific duration – wt or mt for warmup time and measurement time, respectively– for which the benchmark is executed as often as possible, and the performance metrics for a sample of the invocations is reported. Performance metrics from warmup iterations are discarded, and the union of the measurement iterations across all forks form the benchmark's result. All these values can be configured by the developer through JMH annotations or the command line interface (CLI), otherwise default values are used. JMH supports benchmark fixtures, i.e., setup and teardown methods, as well as parameterization of benchmarks. A parameterized benchmark has a number

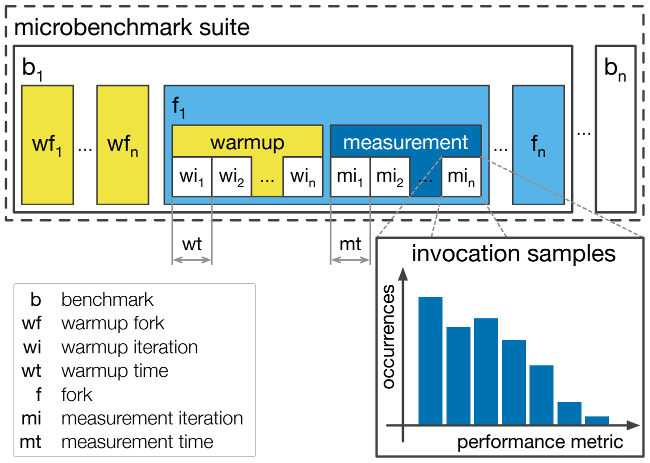


Figure 1: JMH Execution

of parameters with (potentially) multiple values; JMH then runs the benchmark once for every parameter combination, which are formed as the cartesian product of the individual parameters. JMH uses different sets of default configuration values, depending on the version: ≤ `1.20` and ≥ `1.21`. Versions until `1.20` use 10 forks (f) running 40 iterations (20 wi and mi each) with an iteration time (wt and mt) of 1s; starting from `1.21`, defaults are 5 forks (f), 5 iterations (both wi and mi), and 10s iteration time (both wt and mt) [44, 45]. JMH does not use warmup forks (wf) by default.

Consequently, and as Figure 1 depicts, we can define the overall warmup time as $t_w^b = wf * (wi * wt + mi * mt) + f * wi * wt$, the overall measurement time as $t_m^b = f * mi * mt$, and the benchmark execution time as $t^b = t_w^b + t_m^b + t_{fix}^b$, where t_{fix}^b is the time spent in benchmark fixtures. Finally, the full microbenchmark suite execution time T is the sum of all benchmark parameter combinations, defined as $T = \sum_{b \in B} \sum_{p \in P^b} t^{b_p}$, where P^b the set of parameter combinations for a benchmark b. These definitions will be used in the remainder of the paper.

3 PRE-STUDY

To motivate our work, we conduct a pre-study researching whether benchmark execution times are in fact a problem in real-world OSS projects using JMH.

3.1 Data Collection

We create, to the best of our knowledge, the most extensive OSS JMH data set to date from Github, by querying and combining three sources: (1) Google BigQuery's most recent Github snapshot[2], queried for `org.openjdk.jmh` import statements [10, 30]; (2) Github's search application programming interface (API) with an approach as outlined by Stefan et al. [46]; and (3) Maven Central searching for projects with JMH as dependency. Our final dataset consists of 753 pre-study subjects after removing duplicate entries, repositories that do not exist anymore, projects without benchmarks in the most recent commit, and forked projects.

[1]openjdk.java.net/projects/code-tools/jmh (incl. source code examples)

[2]bigquery.cloud.google.com/dataset/fh-bigquery:github_extracts

For each project, we apply the tool *Bencher* [28] to construct abstract syntax trees (ASTs) for Java source-code files and extract information related to (1) execution configuration (`@Fork`, `@Warmup`, `@Measurement`, and `@BenchmarkMode`) and (2) benchmark parameterization (`@Param`). In addition, (3) we extract the JMH version from the build script (*Maven* and *gradle*).

3.2 Summary of Pre-Study Subjects

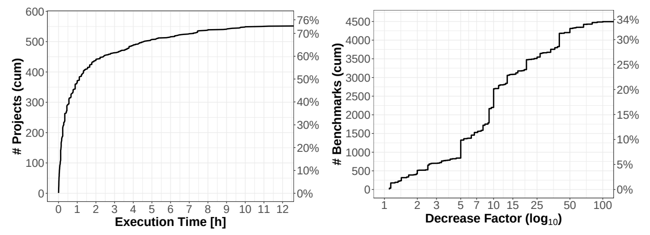
The 753 projects have in total 13'387 benchmark methods and 48'107 parameter combinations. 400 (53.1%) projects feature less than 10 benchmarks, and 52 (6.9%) projects contain 50 benchmarks or more. On average, a project has 19.7±44.9 benchmarks, with a median of 7. The project with the largest benchmark suite is *eclipse/eclipse-collections* with 515 benchmarks. Benchmark parameterization is quite common with projects having, on average, 70.6±303.3 parameterized benchmarks, with a median of 9. 76.9% of the benchmarks have 10 parameter combinations or fewer. We find the largest number of parameter combinations in the project *msteindorfer/criterion*, with 4'132 combinations; and the project with the the most parameter combinations for a single benchmark is *apache/hive*, which contains an individual benchmark with an astounding 2'304 different combinations[3]. However, the majority of the benchmarks are not parameterized, i.e., 10'394 (77.6%).

Extracting the JMH version is crucial for our analysis, as the default values of the execution configurations were changed with version `1.21` (see also § 2). However, automatically extracting the JMH version is not possible for each project. We are able to successfully extract the JMH version from build scripts for 573 (76%) of our pre-study subjects, containing 10'816, or 80.8%, of the benchmarks. About 20% of the projects (containing 4'115 (38.0%) benchmarks) already use the most recent JMH version.

3.3 Results

We use this data to analyze how much time benchmark suites in the wild take to execute. Figure 2a depicts a summary of benchmark suite execution times T for the 573 studied projects where JMH version extraction was successful. The runtimes vary greatly, ranging from 143 milliseconds for *protobufel/protobuf-el* to no less than 7.4 years for *kiegroup/kie-benchmarks* (clearly, this project does not intend to execute all benchmarks at once), with a median of 26.7 minutes. 364 (49%) benchmark suites run for an hour or less, which is probably acceptable, even in CI environments. However, 110 (15%) suites take longer than 3 hours, with 22 projects (3%) exceeding 12 hours runtime. For example, the popular collections library *eclipse/eclipse-collections* has a total benchmark suite runtime of over 16 days, executing 515 benchmarks with 2'575 parameter combinations. We conclude that at least 15% of the pre-study subjects would greatly benefit from an approach to reduce benchmark execution times given their current configuration.

The benchmark suite execution time is based on the extracted JMH configurations from the projects. We speculate that developers specifically apply custom configurations to reduce the (rather conservative) default settings of JMH. Indeed, 4'836 (36%) benchmarks have a configuration change that affects its runtime, of which 4'576



(a) Benchmark suite execution times T

(b) Decreased t^b compared to JMH defaults

Figure 2: Impact of custom configurations on the execution times of (a) benchmark suites and (b) benchmarks.

(34%) have a decreased benchmark time t^b with respect to JMH defaults (see Figure 2b). We observe that for the majority of the benchmarks the execution time is in fact drastically reduced: for 3'735 (28%) and 2'379 (18%) by a factor ≥ 5 and ≥ 10, respectively. Still 374 (3%) benchmarks are reduced by a factor ≥ 50. While only a minority of 250 (2%) of the benchmarks belonging to just 17 (3.0%) of projects are configured to increase execution time compared to the defaults.

Pre-Study Summary. OSS developers extensively customize benchmark configurations, often setting their values considerably lower than the JMH default. Despite these changes, 15% of the projects still have a benchmark suite execution time of over 3 hours. These findings indicate that developers of many projects could be supported by a data-driven way to reduce the execution time of JMH benchmarks.

4 DYNAMIC RECONFIGURATION

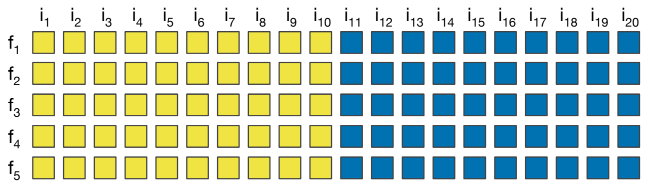
In § 3, we found that real-world OSS benchmark suites often are configured to considerably reduce runtime, with respect to JMH's defaults; still many run for multiple hours, making it effectively impossible to assess performance on every software change. We hypothesize that this time reduction is an effort by developers to keep benchmark suite runtimes reasonable without confirming that benchmark results remain stable (accurate).

This section introduces an approach to dynamically stop benchmarks when their result is stable, with the goal of saving execution time without sacrificing quality.

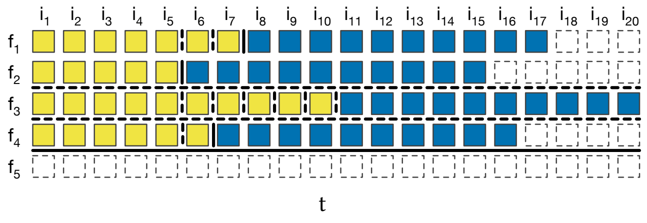
4.1 Approach

JMH allows developers to define benchmark configurations before execution, either through annotations or CLI parameters, and then executes all benchmarks according to this configuration (see § 2). We call this the "static configuration" of a benchmark execution. Figure 3a shows the static configuration where every row indicates a JMH fork (f_1–f_5) and every column/rectangle an iteration (i_1–i_{20}) of the corresponding fork. Yellow rectangles (i_1–i_{10}) indicate warmup iterations, and blue rectangles (i_{11}–i_{20}) indicate measurement iterations. This static configuration bears the problem that all forks are executed with the same configuration, irrespective of the accuracy of the results, potentially wasting precious runtime.

[3]github.com/apache/hive/blob/rel/release-3.1.2/itests/hive-jmh/src/main/java/org/apache/hive/benchmark/vectorization/operators/VectorGroupByOperatorBench.java


i1 i2 i3 i4 i5 i6 i7 i8 i9 i10 i11 i12 i13 i14 i15 i16 i17 i18 i19 i20
f1
f2
f3
f4
f5

(a) Static configuration


i1 i2 i3 i4 i5 i6 i7 i8 i9 i10 i11 i12 i13 i14 i15 i16 i17 i18 i19 i20
f1
f2
f3
f4
f5
t

(b) Dynamic reconfiguration

Figure 3: Standard JMH execution with static configuration vs. dynamic reconfiguration approach. A yellow box is a warmup iteration, a blue box is a measurement iteration, and a dashed box is a skipped iteration. A solid line indicates that the stoppage criterion is met, and a dashed line indicates the opposite.

Algorithm 1: Dynamic reconfiguration algorithm

```
Input: $b \in B$: the benchmark to execute
       $C^b = \langle wi^{min}, wi^{max}, mi, f^{min}, f^{max}, \cdots \rangle$: execution configuration for $b$
       $stable : M \times T \mapsto \{true, false\}$: stability function at threshold $t \in T$ for a set of measurements $M' \in M$
       $t$: stability threshold (specific for $stable$)
Data: $execute : B \mapsto M$: executes a benchmark iteration
Result: Measurements $M^b$ of the benchmark $b$
1  begin
2      $M^b \leftarrow \emptyset$
3      for $f \leftarrow 1$ to $f^{max}$ do
4          $M_w \leftarrow \emptyset$
           // dynamic warmup phase
5          for $wi \leftarrow 1$ to $wi^{max}$ do
6              $M_w \leftarrow M_w \cup execute(b, C^b)$
               // warmup stoppage
7              if $wi \geq wi^{min} \wedge stable(M_w, t)$ then break
           // measurement phase
8          for $1$ to $mi$ do
9              $M^b \leftarrow M^b \cup execute(b, C^b)$
           // fork stoppage
10         if $f \geq f^{min} \wedge stable(M^b, t)$ then break
11     return $M^b$
```

In order to stop benchmark executions when their result is accurate enough, we propose dynamic benchmark reconfiguration, i.e., an approach that dynamically decides, at certain checkpoints, when the benchmark results are unlikely to change with more executions. This happens at two points: (1) within a fork, when the execution reaches a steady state, i.e., the warmup phase was executed long enough, and (2) after a fork's execution, when it is unlikely that more forks will lead to different results. Figure 3b illustrates dynamic reconfiguration. Horizontal bars indicate checkpoints after iterations (line 7), vertical bars indicate checkpoints after forks (line 10), and white, dashed boxes represent iterations that are skipped.

Algorithm 1 depicts the pseudo code for our dynamic reconfiguration approach. The algorithm takes the benchmark to execute b, its extended JMH execution configuration C^b, a stability function *stable* that is executed at every checkpoint, and a threshold t for deciding what is considered stable. C^b is a tuple of configuration values defined as $C^b = \langle wi^{min}, wi^{max}, mi, f^{min}, f^{max}, wf, wt, mt \rangle$ (see also § 2). Note that checkpoints only happen after i_5 and f_2 in the example, defined as wi^{min} and f^{min}. If a benchmark is not stable at a checkpoint, the bar is dashed (solid otherwise) and the warmup phase continues or another fork is spawned. To circumvent the situation where a benchmark's warmup phase never reaches a steady state or the overall measurements are never accurate enough, our approach takes a maximum number of warmup iterations (wi^{max}) and forks (f^{max}), e.g., f_3 has a dashed bar after the last warmup iteration. This guarantees that a single benchmark execution never exceeds a configurable time budget, which defaults to JMH's warmup iterations (wi) and forks (f). Benchmarks often exhibit multiple steady states resulting in multi-modal distributions, and outliers, due to non-deterministic behavior, might still occur even after *stable* considered a fork to be in a steady state [17]. Therefore, our approach uses a fixed number of multiple measurement iterations mi (lines 8–9), because a single measurement iteration would not accurately represent a fork's performance.

4.2 Stoppage Criteria

To decide whether a fork reached a steady state (line 7) or the gathered measurements are stable (line 10), our approach needs to decide whether more measurements provide significantly more accurate results. For this, we rely on statistical procedures on the performance measurement distributions. That is, if more measurements (i.e., data points) are unlikely to change the result distribution, we consider the measurement stable. There are three key aspects to consider: (1) a stability criteria $sc : M \mapsto \mathbb{R}^+$ that assigns a stability value $s \in \mathbb{R}$ to a set of measurements $M' \in M$; (2) a threshold $t \in T$ that indicates whether a stability value s is considered stable; and (3) a stability function $stable : M \times T \mapsto \{true, false\}$ that, based on a set of stability values (extracted from a set of measurements $M' \in M$) and a threshold $t \in T$, decides whether a set of performance measurements is stable or not.

4.2.1 Stopping Warmup Phase. The first stoppage point (line 7) decides whether a fork is in a steady state, which indicates the end of the warmup phase and the start of the measurement phase. For this, the dynamic reconfiguration approach uses a sliding-window technique where the measurement distributions at the last iterations are compared along a stability criteria. Let us consider the set of warmup measurements M_w (across multiple warmup iterations) such that $m_i \in M_w$ at the i^{th} iteration. We then define the sliding-window warmup vector $W_{i''}$ after a current iteration i'', a sliding-window size s^W, and the resulting start iteration of the window $i' = i'' - s^W : i' \geq 1$ in Equation 1.

$$W_{i''} = \left\langle sc\Big(\bigcup_{i=i'}^{x} m_i\Big) \;\Big|\; i' \leq x \leq i'' \right\rangle \tag{1}$$

4.2.2 *Stopping Forks.*

The second stoppage point (line 10) decides whether the benchmark measurement results M^b are sufficiently stable and no additional fork needs to be spawned, therefore stopping the execution of benchmark b. Let us consider the set of measurements M^b (across multiple forks) such that $M^b_f \subseteq M^b$ is the subset of measurements at fork number f. We then define the fork vector $F_{f''}$ after a current fork f'' in Equation 2.

$$F_{f''} = \left\langle sc\Big(\bigcup_{f=1}^{x} M^b_f \Big) \;\Big|\; 1 \leq x \leq f'' \right\rangle \tag{2}$$

4.2.3 *Stability Criteria and Function.*

The dynamic reconfiguration approach allows for different stability criteria (sc) and functions ($stable$), and we identified and evaluated three:

Coefficient of variation (CV): Coefficient of variation (CV) is a measure of variability under the assumption that the distribution is normal. However, performance distributions are usually non-normal, e.g., multi-modal or long-tailed [9, 34]. As a stability criteria sc, CV might still be a “good enough” proxy to estimate a benchmark's stability, especially due to its low computational overhead. Depending on the benchmark, the stability values in the vector $v \in \{W_{i''}, F_{f''}\}$ converge towards different values, making a global threshold t for all benchmarks unrealistic. Instead, we compare all stability values from v such that the delta between the largest and the smallest is at most t. Formally, $stable^{var}(M', t) = true \iff max(v) - min(v) \leq t$.

Relative confidence interval width (RCIW): The second stability criteria sc —relative confidence interval width (RCIW)— is similar to CV, as it estimates a benchmark's variability, hence $stable^{var}$ also applies here. Different from CV, we employ a non-parametric technique based on Monte Carlo simulation called bootstrap [11, 19] to estimate the RCIWs for the mean. For this, we utilize the tool *pa* [29] that implements a technique by Kalibera and Jones [25]. It uses hierarchical random resampling [40] with replacement, which is tailored to performance evaluation. The hierarchical levels are (1) invocations, (2) iterations, and (3) forks (we refer to *pa* [29] and Kalibera and Jones [25] for details).

Kullback-Leibler divergence (KLD): The third stability criteria sc uses a technique outlined by He et al. [18] that constructs a probability that two distributions d_1 and d_2 are similar based on the Kullback-Leibler divergence (KLD) [27]. sc computes this probability (for every element of the vector v) where d_1 is the measurement distribution *without* the last measurement (warmup iteration i or fork f) and d_2 is the measurement distribution *with* the last measurement. Consequently and different from a variability-based stability criteria, the vector v consists of probabilities rather than variabilities. The stability function $stable$ checks whether the mean probability of the stability values from v are above the threshold t. Formally, $stable^{prob}(M', t) = true \iff mean(v) > t$.

4.3 Modified JMH Implementation

We implemented the dynamic reconfiguration approach with the three stoppage criteria for JMH version `1.21`, by adding a reconfiguration benchmark mode with stoppage criteria (sc and $stable$) and threshold (t) properties, annotation properties for wi^{min} and f^{min}, and corresponding CLI flags. Additionally, we adapted JMH's console and JavaScript Object Notation (JSON) result file output to include the new configuration options and added a warning if the stability criterion has not been met for a benchmark. The modified fork of JMH is part of our replication package [32].

5 EMPIRICAL EVALUATION

To assess whether dynamic reconfiguration is effective and efficient, we conduct a case study evaluation on a subset of the Java OSS projects identified in our pre-study (see § 2). Our evaluation compares three dynamic reconfiguration approaches (one for every stoppage criterion). As a baseline for comparison, we use standard JMH with static configuration and the default values.

To support open science, we provide all evaluation data and scripts in a replication package [32].

5.1 Research Questions

First, we want to ensure that dynamic reconfiguration *does not* change the results compared to static configuration. If the results of the same benchmark executed with static configuration and with dynamic reconfiguration are equal, we conclude that dynamic reconfiguration is effective in preserving result quality. For this, we formulate RQ 1:

RQ 1 How does dynamic reconfiguration of software microbenchmarks affect their execution result?

Second, we want to evaluate if dynamic reconfiguration improves the overall runtime of a benchmark suite, compared to static configuration, including the overhead imposed by the stoppage criteria computation. For this, we formulate RQ 2:

RQ 2 How much time can be saved by dynamically reconfiguring software microbenchmarks?

As a benchmark's result quality (accuracy) and runtime are competing objectives, the combination of the results from RQ 1 and RQ 2 validates whether dynamic reconfiguration enables “reducing execution time without sacrificing result quality”.

5.2 Study Subjects

Evaluating the dynamic reconfiguration approach on all 753 pre-study subjects (see § 3) is infeasible as executing benchmark suites potentially takes a long time. Hence, we select a subset of ten projects from a wide variety of domains with small (16) to large (994) benchmark suites. Table 1 lists the study subjects with their number of benchmarks (“# Benchs.”) and benchmark parameter combinations (“# Param. Benchs.”), *git* version used for the evaluation (“Version”), and execution time when using JMH default values (“Exec. Time”). Our evaluation executes all 3'969 benchmark parameter combinations of the ten study subjects[4] , which is 8.2% of the 48'107 parameter combinations from the pre-study.

5.3 Study Setup

We execute all benchmarks, retrieve the benchmark results, and afterwards apply dynamic reconfiguration and the stoppage criteria to the obtained data set. This allows us to experiment with

[4]Except failing benchmarks. See replication package for those [32].

Table 1: Selected study subjects. All projects are hosted on Github except the ones indicated. [1] Repository: hg.openjdk.java.net/code-tools/jmh-jdk-microbenchmarks/. [2] Module directory in repository: hg.openjdk.java.net/code-tools/jmh.

Name	Project	Version	# Benchs.	# Param. Benchs.	Exec. Time	Domain
byte-buddy	*raphw/byte-buddy*	`c24319a`	39	39	5.42h	Bytecode manipulation
JCTools	*JCTools/JCTools*	`19cbaae`	60	148	20.56h	Concurrent data structures
jdk	*jmh-jdk-microbenchmarks*[1]	`d0fab23`	994	1'381	191.81h	Benchmarks of the JDK
jenetics	*jenetics/jenetics*	`002f969`	40	40	5.56h	Genetic algorithms
jmh-core	*jmh-core-benchmarks*[2]	`a07e914`	110	110	15.28h	Benchmarks of JMH
log4j2	*apache/logging-log4j2*	`ac121e2`	358	510	70.83h	Logging
protostuff	*protostuff/protostuff*	`2865bb4`	16	31	4.31h	Serialization
RxJava	*ReactiveX/RxJava*	`17a8eef`	217	1'282	178.06h	Asynchronous programming
SquidLib	*SquidPony/SquidLib*	`055f041`	269	367	50.97h	Visualization
zipkin	*openzipkin/zipkin*	`43f633d`	61	61	8.47h	Distributed tracing

thresholds and parameters without having to rerun the full benchmark suites with our modified JMH implementation (with dynamic reconfiguration).

5.3.1 Execution and Data Gathering. As performance measurements are prone to confounding factors [9, 13, 16, 34, 37], we follow a rigorous methodology to increase result reliability. All projects are (1) patched with JMH `1.21` and (2) compiled and executed with AdoptOpenJDK and Java HotSpot virtual machine (VM) version `1.8.0_222-b10`, except *log4j2* which requires a Java Development Kit (JDK) version ≥ 9, hence we employ version 13+33. (3) We run the benchmarks on a bare-metal machine [3, 43] with a 12-core Intel Xeon X5670 @2.93GHz CPU, 70 GiB memory, and a Samsung SSD 860 PRO SATA III disk, running ArchLinux with a kernel version `5.2.9-1-1-ARCH`. (4) All non-mandatory background processes except ssh are disabled, without explicitly disabling software/hardware optimizations. Regarding benchmark suite execution, we (5) configure and execute all benchmarks with five forks f, 100 measurement iterations mi, 1s measurement time mt, and JMH's `sample` mode, set through JMH's CLI. This configuration corresponds to the JMH `1.21` defaults, only mt changes from 10s to 1s but, at the same time, mi increases by a factor of 10, which grants our approach more checkpoints. Note that warmup iterations wi are set to zero but mi is doubled (from 50 to 100), which is required to obtain results for every iteration to dynamically decide when to stop the warmup phase. Last, (6) we remove outliers that are a magnitude larger than the median.

5.3.2 Approach. With the obtained performance results from the suite executions, we evaluate dynamic reconfiguration with the following parameters. The baseline, i.e., JMH with static configuration, uses the JMH `1.21` default configuration. For this, we remove the first 50 iterations (corresponding to wi) from each fork and use the 50 remaining iterations as mi.

For the dynamic reconfiguration approaches, we employ the configuration $C^b = \langle 5, 50, 10, 2, 5, 0, 1s, 1s\rangle$ (see § 4.1). Note that we also reduce mi to 10 instead of 50 à 1s, which the baseline uses. Initial experiments showed that an increase in measurement iterations, *after* a steady state is reached, has only a minor effect on result accuracy *but* with considerably longer runtimes. To reduce computational overhead at checkpoints, we draw a weighted sample of 1'000 invocations per iteration. We use the following parameters for the three dynamic reconfiguration approaches (one per stoppage criterion). (1) The sliding-window size is set to $s^W = 5$. (2) **CV** uses a threshold $t = 0.01$, which corresponds to a maximum variability difference in the sliding window of 1%. (3) For **RCIW**, we use a 99% confidence level, 1'000 bootstrap simulations (which is a good tradeoff between runtime overhead and estimation accuracy), and a threshold $t = 0.03$ following best practice [16]. (4) For **KLD**, we partition the distributions d_1 and d_2 into 1'000 strips [18] for the KLD calculation and remove outliers that are more than $1.5 \times IQR$ away from the median. More strips would result in longer calculation times for the kernel density estimation and, consequently, in a higher runtime overhead. We use a threshold $t = 0.99$, which corresponds to a mean probability within the sliding window of 99% or larger.

5.4 Results and Analysis

We now present the results of our empirical evaluation by comparing the benchmark results of the static configuration to the ones of our dynamic reconfiguration approaches with the three stoppage criteria.

5.4.1 RQ 1: Result Quality. To assess whether applying dynamic reconfiguration changes benchmark results and to answer RQ 1, we perform two analyses between results coming from static configuration and each of the three dynamic reconfiguration approaches: (1) statistical A/A tests and (2) mean performance change rate.

A/A Tests. An A/A test checks whether results from two distributions are *not* significantly different, where no difference is expected. In our context, this means if an A/A test between static configuration and dynamic reconfiguration (for each stoppage criterion) does not report a difference, we conclude that dynamic reconfiguration does not change the benchmark result. Following performance engineering best practice [6, 8, 25, 31], we estimate the confidence interval for the ratio of means with bootstrap [11], using 10'000 iterations [19], and employing hierarchical random resampling with replacement on (1) invocation, (2) iteration, and (3) fork level [25] (again relying on *pa* [29]). If the confidence interval (of the ratio) straddles 1, there is no statistically significant difference. Note that this procedure is different from the stoppage criteria RCIW (see § 4); here we compare the results (all measurement iterations *mi* from all forks f) of two techniques, whereas RCIW uses confidence interval widths as a variability measure of a single technique.

The first row of Table 2 shows the A/A results. For a majority of the 3'969 benchmark parameter combinations, applying dynamic reconfiguration *does not* result in significantly different distributions. About 80% or more of the benchmarks have similar result distributions compared to the static configuration. RCIW achieves the best result with 87.6%, while CV and KLD perform similarly well with 78.8% and 79.6%, respectively. Note that the static approach uses 50 measurement iterations (*mi*) while the dynamic approach "only" runs 10, indicating that if a steady state is reached (which is one goal of dynamic reconfiguration) more measurement iterations have a negligible impact on the overall result.

Change Rate. In addition to A/A tests, we assess the performance change rate between the static configuration approach and each of the dynamic reconfiguration approaches, i.e., by how much the means of the performance result distributions differ. The change rate augments the A/A tests' binary decision, by showing *how* different the benchmark results become when applying dynamic reconfiguration.

The second row of Table 2 shows the mean change rate across all benchmarks in percent and its standard deviation. The mean change rate between the three stoppage criteria and the static approach is ~3% or lower for all three. Note that, following a rigorous measurement methodology, ~3% could still be caused by JVM instabilities unrelated to our approach [17]. Again, RCIW is the best criterion with 1.4%±3.8%. Finally, the last three rows show how many benchmarks have a change rate below 1%, 2%, and 3% for all stoppage criteria. We observe that RCIW outperforms the other two significantly, followed by KLD. ~73% of the benchmarks have a change rate below 1%, ~87% below 2%, and ~92% below 3%. This suggests that RCIW is a highly effective technique for stopping benchmark executions.

Figure 4 depicts the change rate distributions per project and stoppage criterion, where every data point corresponds to a benchmark's mean performance change. Considering the median change rate of a project's benchmarks, RCIW performs best for all projects except *jenetics*, *jmh-core*, and *SquidLib* where KLD is slightly superior. CV consistently has the largest change rates of the three stoppage criteria; nonetheless, it performs only slightly worse in most cases. Considering the mean change rate, RCIW is the most accurate stoppage criteria for 9/10 projects, with only *jmh-core* being more stable when KLD is employed. Note, that for the projects where RCIW is not the best stoppage criterion, both mean and median change rates are below 1%. The projects with the most diverging benchmarks between static configuration and dynamic reconfiguration execution are *byte-buddy*, *JCTools*, *log4j2*, and *SquidLib*. The benchmarks of these projects are less stable compared to the other projects, likely due to executing non-deterministic behaviour such as concurrency and IO. Results from benchmarks that are less stable will potentially have statistically different distributions and, therefore, not maintaining the same result quality.

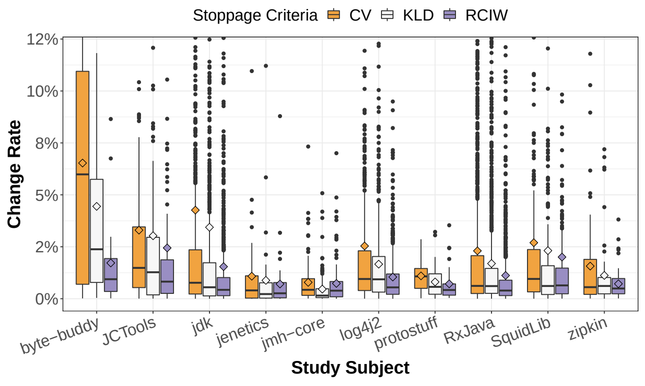


Figure 4: Mean change rate per study subject and stoppage criteria. The bar indicates the median, the diamond the mean, the box the IQR, and the whiskers $[Q_1|Q_3] + 1.5 * IQR$.

Table 2: Result quality differences between static configuration approach and dynamic reconfiguration approaches

	CV	KLD	RCIW
A/A tests *not* different	78.8%	79.6%	87.6%
Mean change rate	3.1% ± 8.1%	2.4% ± 7.4%	1.4% ± 3.8%
# benchs < 1%	57.4%	62.3%	73.2%
# benchs < 2%	72.4%	78.2%	87.0%
# benchs < 3%	79.6%	84.6%	91.9%

Unreachble Stability Criteria. If the stability function $stable$ never evaluates the measurements after a warmup iteration or a fork as stable, the maximum number of warmup iterations (wi^{max}) or forks (f^{max}) are executed. This corresponds to the static configuration of JMH. We analysed how often stability is not achieved according to the three stoppage criteria across all study subjects. CV is the most lenient criterion with only 1.0% of the benchmarks' forks not considered stable after 50 warmup iterations and 12% of the benchmarks insufficiently accurate after five forks. KLD achieves similar numbers (0.8%) for warmup iterations, however 46.4% of the benchmarks were not considered stable after five forks. RCIW is even more restrictive where 46.7% and 37.9% of the benchmarks do not reach the stability criteria after wi^{max} and f^{max}, respectively. This restrictiveness impacts the A/A test and mean change rate results, leading to benchmark results with higher quality. Not reaching the stability criteria can either happen if the threshold t is too restrictive or the benchmark is inherently variable, which is a common phenomenon [30, 31].

RQ 1 Summary. Applying dynamic reconfiguration *does not* change the result quality of the majority of the benchmarks, when compared to the static configuration. The RCIW stoppage criteria outperforms KLD and CV, with 87.6% of the benchmarks maintaining their result quality and a mean performance change rate of 1.4%.

5.4.2 *RQ 2: Time Saving.* The main goal of dynamic reconfiguration is to save time executing benchmark suites. For this, and to answer RQ 2, we (1) measure the runtime overhead of the three stoppage criteria, (2) estimate the time saving for all projects compared to the static configuration, and (3) show at which checkpoint (warmup or fork) more time can be saved.

Runtime Overhead. To measure the runtime overhead of the three stoppage criteria, we execute the benchmark suite of *log4j2* once with standard JMH 1.21 (i.e., static configuration) and once for each stoppage criteria with our JMH fork implementing dynamic reconfiguration. To make the two comparable, we use a configuration for static and dynamic approaches of $C^b = \langle 5, 90, 10, 2, 5, 0, 1s, 1s\rangle$, but do not stop at the stoppage checkpoints. We measure the end-to-end execution time $t^{b\prime}$ of every benchmark b when executed through JMH's CLI. Note that this time includes JVM startup, benchmark fixtures, benchmark execution, and stoppage criteria computation. The overheads $o \in O$ of all benchmarks for a stoppage criteria is $O = \bigcup_{b \in B} t^{b\prime}_{dyn} / t^{b\prime}_{sta} - 1$, where $t^{b\prime}_{dyn}$ is the execution time of the dynamic reconfiguration with a specific stoppage criteria, and $t^{b\prime}_{sta}$ is the execution time of the static configuration.

The overheads we measure are $\boldsymbol{o^{CV} = 0.88\% \pm 0.34\%}$ for CV, $\boldsymbol{o^{RCIW} = 10.92\% \pm 0.63\%}$ for RCIW, $\boldsymbol{o^{KLD} = 4.32\% \pm 0.65\%}$ for KLD. Note that changing the iteration time of 1s and executing benchmarks on different hardware might affect the overhead. The considerable difference in overhead is explained by the complexity of the stoppage criteria calculations. Whereas CV is computationally cheap (it only needs to compute standard deviation, mean, and their difference), RCIW is computationally intensive due to the simulations required for bootstrap. Because there is hardly any overhead variability (< 1%) among all benchmarks, we consider the overhead constant and use the mean value in the remainder of the experiments.

Time Saving Estimation. To estimate the overall time that can be saved with dynamic reconfiguration, we adapt the execution time equation t^b (see § 2) to incorporate the stoppage criteria. The dynamic reconfiguration benchmark execution time is then $t^b_{dyn} = \sum_{f \in forks} [(1+o) * wi_f * wt + mi * mt]$. $forks$ corresponds to the number of forks f a benchmark had according to the stoppage criterion, wi_f to the number of warmup iterations in this fork f, and the rest according to C^b from § 4.1. For simplicity and because of the low variability between benchmark overheads, we disregard benchmark fixture times. The total benchmark suite execution time when using dynamic reconfiguration is then $T_{dyn} = \sum_{b \in B'} t^b_{dyn}$, where B' is the set of benchmark parameter combinations.

Table 3 shows the time saving per project and stoppage criteria in absolute numbers (hours) and relative to the static configuration. We observe that dynamic reconfiguration with all three stoppage criteria enables drastic time reductions compared to static configuration. In total, CV and KLD save ~80% and RCIW ~66% of the benchmark suite execution times of all projects combined. For individual projects, the time saving ranges between 72.8% and 86.0% for CV, 48.4% and 83.1% for RCIW, and 77.8% and 83.3% for KLD. Even with the computationally most expensive technique, i.e., RCIW, we can save at least 48.4% of time. In total numbers, the savings are between 3.43h and 157.32h for CV, 2.62h and 135.57h for RCIW, and 3.44h and 154.41h for KLD.

Table 3: Time saving per project and stoppage criteria

Project	Time Saving					
	CV		RCIW		KLD	
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%)
jdk	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	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%)**

Stoppage Criteria Checkpoints. Dynamic reconfiguration defines two points during benchmark execution when to stop: (1) after the warmup phase if measurements are stable within a fork and (2) after a fork if measurements across forks are stable. In our analysis, the range of warmup iterations is from five (wi^{min}) to 50 (wi^{max}), and forks are between two (f^{min}) and five (f^{max}) (see C^b in § 4.1). Although CV and KLD save a similar amount of time, they have different stoppage behavior. Where CV requires more warmup iterations (18.5 ± 9.4) than KLD (14.1 ± 6.9), the opposite is the case for forks with 3.1 ± 1.2 vs. 4.1 ± 1.2, respectively. RCIW, which saves considerably less time, demands more warmup iterations (34.6 ± 16.6) to consider a fork stable but lies between CV and KLD in terms of forks (3.3 ± 1.4). The reported numbers are arithmetic means with standard deviations. Generally, warmup iterations are more reduced than forks in our setup, indicating that fork-to-fork variance is more present than within-fork variance. Dynamic reconfiguration enables finding the sweet spot between shortening warmup iterations and forks in combination with a certain stoppage criteria.

RQ 2 Summary. With runtime overheads between <1% and ~11%, dynamic reconfiguration enables reducing benchmark suite runtimes by 48.4% to 86.0% compared to JMH's default runtime.

6 DISCUSSION AND RECOMMENDATIONS

Our pre-study (see § 3) shows that developers often drastically reduce benchmark execution times. We see two potential reasons for this: (1) JMH defaults are overly conservative, and benchmarks with shorter runtimes often still produce results that are considered sufficiently accurate; or (2) the benchmark suite runtimes are too long, and, consequently, developers trade shorter runtimes for inaccurate results. We hypothesize that the latter is more likely, but leave the developer perspective for configuration choices for future work. In any case, the proposed dynamic reconfiguration approach enables reducing time while maintaining similar benchmark results, as our empirical evaluation shows.

Recommendations for Developers. Developers are advised to either assess their benchmark accuracies when executed in their environment and adjust configurations accordingly, or employ dynamic reconfiguration which is able to adjust to different execution environments. The choice of stoppage criteria depends on the required result quality and, therefore, the performance change sizes desired to be detected. For slightly less accurate results but more time reduction, we recommend using KLD, otherwise RCIW is preferred. The exact threshold t depends on the stability of the execution environments the benchmarks are run in. If a controlled, bare-metal environment is available, we suggest the thresholds of our study. In a virtualized or cloud environment, the thresholds need to be adjusted (see also He et al. [18]). The effectiveness of our technique in non-bare-metal environments, such as in the cloud, is subject to future research.

Microbenchmarks in CI. The long benchmark execution times (see § 3 and [22, 30, 42]) are a major obstacle for including microbenchmarks in CI [4]. To overcome this hurdle, a combination of our technique with benchmark selection [12], benchmark prioritization [36], and risk analysis on commits [22] would reduce the required time for microbenchmarking and potentially enable CI integration. Continuously assessing software performance would increase confidence that a change does not degrade performance and likely be beneficial for performance bug root cause analysis.

Choosing Configuration Parameters. Choosing configuration parameters that keep execution time low and result accuracy high is non-trivial, and developers decrease configurations drastically. Our results show the importance of setting the warmup phase correctly and utilizing multiple forks for benchmark accuracy. With a large number of benchmarks, expecting developers to pick the "right" values becomes unrealistic. Our dynamic reconfiguration approach helps in this regard by deciding based on data and per benchmark when the results are accurate enough.

Iteration Time and Forks. The warmup and measurement times affect benchmark result accuracy and control the frequency with which stability checkpoints occur. JMH 1 . 21 changed the iteration time from 1s to 10s, and reduced the number of forks from ten to five [44, 45]. The OpenJDK team argued that 1s is too short for large workloads [45]. We performed an additional analysis whether result accuracy changes when switching from 10s to 1s but did not observe differences in most cases. Hence, we decided for 1s iterations to give the dynamic reconfiguration approach more checkpoints to assess a benchmark's stability. Whereas 10s is a safe choice for static configurations, we believe that 1s provides more flexibility and works better with dynamic reconfiguration. Our results support reducing to five forks, which indicates that most fork-to-fork variability is captured.

Unreachable Stability Criteria. Although the stability criteria is frequently not met for warmup iterations or forks of individual benchmarks, at least when using KLD and RCIW, the overall runtime of the full benchmark suites is considerably reduced (see § 5.4). Dynamic reconfiguration uses upper bounds for warmup iterations (wi^{max}) and forks (f^{max}); therefore, it does not exceed the runtime of standard JMH with static configuration. In case of an unreachable stability criteria, our JMH implementation warns the developer, who can then adjust this benchmark's upper bounds to obtain better results.

7 THREATS TO VALIDITY

Construct Validity. Our pre-study (see § 3) relies on information extracted from source code, i.e., configurations based on JMH annotations. We do not consider overwritten configurations through CLI arguments, which might be present in build scripts or documentation in the repositories. Reported runtimes do not consider fixture (setup and teardown) times, JVM startup, and times spent in the benchmark harness of JMH; and assume iteration times are as configured, while in reality they are minimum times. Therefore, reported times might slightly underestimate the real execution times.

The results and implications from RQ 1 are based on the notion of benchmark result similarity. We assess this by means of statistical A/A tests (based on bootstrap confidence intervals for the ratio of means) and mean performance change rate, similar to previous work [6, 31]. Other tests for the similarity of benchmark results, such as non-parametric hypothesis tests and effect sizes [10, 31], might lead to different outcomes. We further base the time savings from RQ 2 on overhead calculations from a single project and assume this overhead is constant for all stoppage points and benchmarks. There is hardly any reason to believe that overheads change between projects, benchmarks, and stoppage points, because the number of data points used for stoppage criteria computation should be similar. Further, we perform post-hoc analysis on a single benchmark execution data set for all stoppage criteria. That is, we execute the benchmark suites with five forks and 100 measurement iterations à 1s and then compute the stoppage points. Computing the stoppage points while executing test suites might lead the slightly different results. Finally, we use a sliding window approach for determining the end of the warmup phase with a window size of five. Different window sizes might impose a larger runtime overhead and change the stoppage point outcomes.

Internal Validity. Internal validity is mostly concerned with our performance measurement methodology and the employed thresholds. We follow measurement best practice [16] and run experiments on a bare-metal machine [3, 43] to reduce measurement bias [9, 13, 34, 37]. We did not explicitly turn off software and hardware optimizations, which might affect benchmark variability and, therefore, our results. Regarding the thresholds, we started from previous works [16, 18] and adapted them to fit the context of microbenchmarks. As we used the same thresholds for all benchmarks and projects, we are confident that they are generally applicable for Java microbenchmarks executed on a similar machine to ours. Further, the times reported in § 3 rely on the JMH version of a benchmark; we applied simple heuristics to extract the version, which might not be fully accurate in case of, for instance, multi-module projects or dynamic JMH version declarations.

External Validity. Generalizability might be affected with respect to the studied projects. We only focus on open-source projects from Github, and it is unclear whether our findings are equally valid in the context of industrial software or projects hosted on other platforms. Especially, the ten selected projects for our empirical evaluation (see § 5) might not be a representative sample for all JMH projects. Due to the long benchmark suite execution times, more projects would not have been feasible to study. We aimed for a diverse set of projects, spanning multiple domains (see Table 1), covering ~8% of the benchmarks from the pre-study (see § 3). Moreover, our focus has been on Java projects that use JMH as their benchmarking framework. Although the concepts from § 4 also translate to other frameworks and languages, the exact results might be different. We opted for Java/JMH because (1) it is a dynamically compiled language where warmup phases and multiple forks are essential, (2) JMH benchmark suites are long running [30] and can benefit greatly from dynamic reconfiguration, and (3) JMH is a mature framework with many features offering great opportunities for our approach. Finally, switching to different Java virtual machines, such as Eclipse OpenJ9 or Graal, might change the results due to different performance characteristics.

8 RELATED WORK

Performance testing is a form of measurement-based performance engineering [49], which comes in two main flavors: system/component-level tests and method/statement-level tests. Historically, research focussed on system-level tests [24, 35, 48], such as load and stress testing, with more recent advances targeting industrial applicability and practice [15, 38]. The other flavor, i.e., software microbenchmarks and performance unit tests, has only recently gained popularity in research. Studies on open-source projects [33, 46] found that adoption lags behind their functional counter-parts, i.e., unit tests. One problem is that handling performance tests is complex and requires in-depth knowledge from developers. To reduce this friction, Ding et al. [14] studied utilizing unit tests for assessing performance properties. Bulej et al. [8] proposed a framework that lets developers specify performance assertions and handles rigorous statistical evaluation. Horký et al. [21] compose performance unit test outcomes into code documentation to raise performance awareness, and Damasceno Costa et al. [10] uncover bad practices in microbenchmark code through static analyses. Generating tests removes the need to write tests by hand: AutoJMH helps avoiding pitfalls rooted in compiler optimization [41], Pradel et al. [39] generate performance regression tests for concurrent classes, and PerfSyn synthesizes inputs through mutation that expose worst-case performance behaviour [47]. Our work is orthogonal to the aforementioned works: it dynamically adapts software microbenchmark configurations to stop their execution once their result is stable.

Long execution times [15, 22, 30] and uncertain results [31, 34] are well-known to complicate the usage of performance tests in general, including software microbenchmarks. There are a few approaches that reduce the time spent in performance testing activities without considering result quality: (1) predicting commits that are likely to impact performance [22, 42], (2) prioritizing [36] and (3) selecting [2, 12] the tests in a suite that are more likely to expose performance changes. Our approach pursues the same goal of reducing benchmarking time, but with a focus on running all benchmarks (similar to prioritization) as long as necessary while maintain the same result quality. Result quality is impaired by not running enough measurements as well as measurement bias, which requires careful experiment planning and execution [5, 9, 13, 16, 17, 26, 37]. To mitigate measurement bias, Georges et al. [16] outlined a rigorous methodology how to asses performance of Java programs, which we base our measurement technique on. Using the correct statistical techniques to assess performance is paramount, with estimated confidence intervals using bootstrap being the state-of-the-art [7, 8, 25, 31]. One of our stopping criteria is based on and our result quality evaluation uses confidence intervals with bootstrap. To decide how many measurements are enough, approaches using statistical techniques have been proposed, employing CV [16, 34], confidence intervals [23, 34], and the Kullback-Leibler divergence (KLD) [18]. With these, performance experiments such as benchmark executions run until their results are accurate/stable enough and then abort execution, ideally reducing execution time. Our stoppage criteria use these three techniques and apply them in the context of software microbenchmarks after the warmup phase and after every fork. Closest to our approach are the ones by Maricq et al. [34] and He et al. [18]. Maricq et al. [34] estimate the number of trials and iterations using a bootstrap technique. However, while they perform this estimation before executing benchmarks, we evaluate result quality during execution. He et al. [18] stop system-level performance tests executed in cloud environments, once they reach a certain stability criteria. Different from the benchmarks used in their study, microbenchmarks are much shorter, with runtimes in the order of seconds instead of multiple hours. Our work builds on top of this statistics-based approach using KLD for system benchmarks, adapts it for microbenchmarks and extends it to other stoppage criteria.

9 CONCLUSIONS

This paper introduced a dynamic reconfiguration approach for software microbenchmarks, which reduces benchmark execution time and maintains the same result quality. In a pre-study based on real-world configurations of 13'387 microbenchmarks coming from 753 projects, we find that developers make extensive use of custom configurations to considerably reduce runtimes for 34% of the benchmarks. Still, about 15% of the projects have benchmark suite runtimes of more than 3 hours. Our dynamic reconfiguration approach implements data-driven decisions to stop microbenchmark executions, assisting developers with the intricate task of correctly configuring microbenchmarks. With overheads between 1% and 11%, it achieves a time reduction of 48.4% to 86.0%, with between 78.8% and 87.6% of the microbenchmarks preserving their result quality. These results show that dynamic reconfiguration is highly effective and efficient, and we envision it to enable regular performance microbenchmarking activities, such as part of CI.

ACKNOWLEDGMENTS

The research leading to these results has received funding from the Swiss National Science Foundation (SNF) under project MINCA – Models to Increase the Cost Awareness of Cloud Developers (no. 165546) and the Swedish Research Council VR under grant number 2018-04127 (Developer-Targeted Performance Engineering for Immersed Release and Software Engineers).

REFERENCES

[1] Hammam M. AlGhmadi, Mark D. Syer, Weiyi Shang, and Ahmed E. Hassan. 2016. An Automated Approach for Recommending When to Stop Performance Tests. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 279–289. https://doi.org/10.1109/ICSME.2016.46

[2] Deema Alshoaibi, Kevin Hannigan, Hiten Gupta, and Mohamed Wiem Mkaouer. 2019. PRICE: Detection of Performance Regression Introducing Code Changes Using Static and Dynamic Metrics. In *Search-Based Software Engineering*. Springer International Publishing, 75–88. https://doi.org/10.1007/978-3-030-27455-9_6

[3] Eytan Bakshy and Eitan Frachtenberg. 2015. Design and Analysis of Benchmarking Experiments for Distributed Internet Services. In *Proceedings of the 24th International Conference on World Wide Web* (Florence, Italy) *(WWW '15)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, Switzerland, 108–118. https://doi.org/10.1145/2736277.2741082 doi:10.1145/2736277.2741082.

[4] Cor-Paul Bezemer, Simon Eismann, Vincenzo Ferme, Johannes Grohmann, Robert Heinrich, Pooyan Jamshidi, Weiyi Shang, André van Hoorn, Monica Villavicencio, Jürgen Walter, and Felix Willnecker. 2019. How is Performance Addressed in DevOps?. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering* (Mumbai, India) *(ICPE '19)*. ACM, New York, NY, USA, 45–50. https://doi.org/10.1145/3297663.3309672 doi:10.1145/3297663.3309672.

[5] Stephen M. Blackburn, Amer Diwan, Matthias Hauswirth, Peter F. Sweeney, José Nelson Amaral, Tim Brecht, Lubomír Bulej, Cliff Click, Lieven Eeckhout, Sebastian Fischmeister, and et al. 2016. The Truth, The Whole Truth, and Nothing But the Truth: A Pragmatic Guide to Assessing Empirical Evaluations. *ACM Trans. Program. Lang. Syst.* 38, 4, Article Article 15 (Oct. 2016), 20 pages. https://doi.org/10.1145/2983574

[6] Lubomír Bulej, , Vojtěch Horký, and Petr Tůma. 2019. Initial Experiments with Duet Benchmarking: Performance Testing Interference in the Cloud. In *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. 249–255. https://doi.org/10.1109/MASCOTS.2019.00035

[7] Lubomír Bulej, , Vojtěch Horký, Petr Tůma, François Farquet, and Aleksandar Prokopec. 2020. Duet Benchmarking:Improving Measurement Accuracy in the Cloud. In *Proceedings of the 2020 ACM/SPEC International Conference on Performance Engineering* (Edmonton, Canada) *(ICPE 2020)*. ACM, New York, NY, USA.

[8] Lubomír Bulej, Tomáš Bureš, Vojtěch Horký, Jaroslav Kotrč, Lukáš Marek, Tomáš Trojánek, and Petr Tůma. 2017. Unit testing performance with Stochastic Performance Logic. *Automated Software Engineering* 24, 1 (2017), 139–187. https://doi.org/10.1007/s10515-015-0188-0

[9] Charlie Curtsinger and Emery D. Berger. 2013. STABILIZER: Statistically Sound Performance Evaluation. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Houston, Texas, USA) *(ASPLOS '13)*. ACM, New York, NY, USA, 219–228. https://doi.org/10.1145/2451116.2451141

[10] Diego Elias Damasceno Costa, Cor-Paul Bezemer, Philipp Leitner, and Artur Andrzejak. 2019. What's Wrong With My Benchmark Results? Studying Bad Practices in JMH Benchmarks. *IEEE Transactions on Software Engineering* (2019), 1–1. https://doi.org/10.1109/TSE.2019.2925345

[11] A. C. Davison and D. V. Hinkley. 1997. Bootstrap Methods and Their Application. 94 (01 1997).

[12] Augusto Born de Oliveira, Sebastian Fischmeister, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. 2017. Perphecy: Performance Regression Test Selection Made Simple but Effective. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 103–113. https://doi.org/10.1109/ICST.2017.17

[13] Augusto Born de Oliveira, Jean-Christophe Petkovich, Thomas Reidemeister, and Sebastian Fischmeister. 2013. DataMill: Rigorous Performance Evaluation Made Easy. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering* (Prague, Czech Republic) *(ICPE '13)*. ACM, New York, NY, USA, 137–148. https://doi.org/10.1145/2479871.2479892

[14] Zishuo Ding, Jinfu Chen, and Weiyi Shang. 2020. Towards the Use of the Readily Available Tests from the Release Pipeline as Performance Tests. Are We There Yet?. In *Proceedings of the 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE 2020)*. ACM, New York, NY, USA.

[15] King Chun Foo, Zhen Ming (Jack) Jiang, Bram Adams, Ahmed E. Hassan, Ying Zou, and Parminder Flora. 2015. 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* (Florence, Italy) *(ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 159–168. http://dl.acm.org/citation.cfm?id=2819009.2819034

[16] Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically Rigorous Java Performance Evaluation. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications* (Montreal, Quebec, Canada) *(OOPSLA '07)*. ACM, New York, NY, USA, 57–76. https://doi.org/10.1145/1297027.1297033

[17] Joseph Yossi Gil, Keren Lenz, and Yuval Shimron. 2011. 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* (Portland, Oregon, USA) *(SPLASH '11 Workshops)*. ACM, New York, NY, USA, 297–308. https://doi.org/10.1145/2095050.2095100 doi:10.1145/2095050.2095100.

[18] Sen He, Glenna Manns, John Saunders, Wei Wang, Lori Pollock, and Mary Lou Soffa. 2019. 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* (Tallinn, Estonia) *(ESEC/FSE 2019)*. ACM, New York, NY, USA, 188–199. https://doi.org/10.1145/3338906.3338912

[19] Tim C. Hesterberg. 2015. What Teachers Should Know About the Bootstrap: Resampling in the Undergraduate Statistics Curriculum. *The American Statistician* 69, 4 (2015), 371–386. https://doi.org/10.1080/00031305.2015.1089789 arXiv:https://doi.org/10.1080/00031305.2015.1089789 PMID: 27019512.

[20] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. 2016. Usage, Costs, and Benefits of Continuous Integration in Open-source Projects. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (Singapore, Singapore) *(ASE 2016)*. ACM, New York, NY, USA, 426–437. https://doi.org/10.1145/2970276.2970358

[21] Vojtěch Horký, Peter Libič, Lukáš Marek, Antonín Steinhauser, and Petr Tůma. 2015. Utilizing Performance Unit Tests To Increase Performance Awareness. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering* (Austin, Texas, USA) *(ICPE '15)*. ACM, New York, NY, USA, 289–300. https://doi.org/10.1145/2668930.2688051

[22] Peng Huang, Xiao Ma, Dongcai Shen, and Yuanyuan Zhou. 2014. Performance Regression Testing Target Prioritization via Performance Risk Analysis. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. ACM, New York, NY, USA, 60–71. https://doi.org/10.1145/2568225.2568232

[23] Raj Jain. 1991. *The Art of Computer Systems Performance Analysis.* Wiley.

[24] Zhen Ming Jiang and Ahmed E. Hassan. 2015. A Survey on Load Testing of Large-Scale Software Systems. *IEEE Transactions on Software Engineering* 41, 11 (Nov 2015), 1091–1118. https://doi.org/10.1109/TSE.2015.2445340

[25] Tomas Kalibera and Richard Jones. 2012. *Quantifying Performance Changes with Effect Size Confidence Intervals.* Technical Report 4–12. University of Kent. 55 pages. http://www.cs.kent.ac.uk/pubs/2012/3233

[26] Tomas Kalibera and Richard Jones. 2013. Rigorous Benchmarking in Reasonable Time. In *Proceedings of the 2013 International Symposium on Memory Management* (Seattle, Washington, USA) *(ISMM '13)*. ACM, New York, NY, USA, 63–74. https://doi.org/10.1145/2464157.2464160

[27] Solomon Kullback and Richard A. Leibler. 1951. On Information and Sufficiency. *Ann. Math. Statist.* 22, 1 (03 1951), 79–86. https://doi.org/10.1214/aoms/1177729694

[28] Christoph Laaber. 2020. bencher - JMH Benchmark Analysis and Prioritization. https://github.com/chrstphlbr/bencher

[29] Christoph Laaber. 2020. pa - Performance (Change) Analysis using Bootstrap. https://github.com/chrstphlbr/pa

[30] Christoph Laaber and Philipp Leitner. 2018. An Evaluation of Open-source Software Microbenchmark Suites for Continuous Performance Assessment. In *Proceedings of the 15th International Conference on Mining Software Repositories* (Gothenburg, Sweden) *(MSR '18)*. ACM, New York, NY, USA, 119–130. https://doi.org/10.1145/3196398.3196407

[31] Christoph Laaber, Joel Scheuner, and Philipp Leitner. 2019. Software microbenchmarking in the cloud. How bad is it really? *Empirical Software Engineering* (17 Apr 2019), 40. https://doi.org/10.1007/s10664-019-09681-1

[32] Christoph Laaber, Stefan Würsten, Harald C. Gall, and Philipp Leitner. [n.d.]. Replication Package "Dynamically Reconfiguring Software Microbenchmarks: Reducing Execution Time Without Sacrificing Result Quality". https://doi.org/10.6084/m9.figshare.11944875

[33] Philipp Leitner and Cor-Paul Bezemer. 2017. 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* (L'Aquila, Italy) *(ICPE '17)*. ACM, New York, NY, USA, 373–384. https://doi.org/10.1145/3030207.3030213

[34] Aleksander Maricq, Dmitry Duplyakin, Ivo Jimenez, Carlos Maltzahn, Ryan Stutsman, and Robert Ricci. 2018. Taming Performance Variability. In *Proceedings*

of the 13th USENIX Conference on Operating Systems Design and Implementation (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 409–425.

[35] Daniel A. Menascé. 2002. Load testing of Web sites. *IEEE Internet Computing* 6, 4 (July 2002), 70–74. https://doi.org/10.1109/MIC.2002.1020328

[36] Shaikh Mostafa, Xiaoyin Wang, and Tao Xie. 2017. PerfRanker: Prioritization of Performance Regression Tests for Collection-Intensive Software. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Santa Barbara, CA, USA) *(ISSTA '17)*. ACM, New York, NY, USA, 23–34. https://doi.org/10.1145/3092703.3092725

[37] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. 2009. Producing Wrong Data Without Doing Anything Obviously Wrong!. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems* (Washington, DC, USA) *(ASPLOS XIV)*. ACM, New York, NY, USA, 265–276. https://doi.org/10.1145/1508244.1508275

[38] Thanh H. D. Nguyen, Meiyappan Nagappan, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. 2014. An Industrial Case Study of Automatically Identifying Performance Regression-causes. In *Proceedings of the 11th Working Conference on Mining Software Repositories* (Hyderabad, India) *(MSR 2014)*. ACM, New York, NY, USA, 232–241. https://doi.org/10.1145/2597073.2597092

[39] Michael Pradel, Markus Huggler, and Thomas R. Gross. 2014. Performance Regression Testing of Concurrent Classes. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (San Jose, CA, USA) *(ISSTA '14)*. ACM, New York, NY, USA, 13–25. https://doi.org/10.1145/2610384.2610393

[40] Shiquan Ren, Hong Lai, Wenjing Tong, Mostafa Aminzadeh, Xuezhang Hou, and Shenghan Lai. 2010. Nonparametric bootstrapping for hierarchical data. *Journal of Applied Statistics* 37, 9 (2010), 1487–1498. https://doi.org/10.1080/02664760903046102 arXiv:https://doi.org/10.1080/02664760903046102

[41] Marcelino Rodriguez-Cancio, Benoit Combemale, and Benoit Baudry. 2016. Automatic Microbenchmark Generation to Prevent Dead Code Elimination and Constant Folding. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (Singapore, Singapore) *(ASE 2016)*. Association for Computing Machinery, New York, NY, USA, 132–143. https://doi.org/10.1145/2970276.2970346

[42] Juan Pablo Sandoval Alcocer, Alexandre Bergel, and Marco Tulio Valente. 2016. Learning from Source Code History to Identify Performance Failures. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering* (Delft, The Netherlands) *(ICPE '16)*. ACM, New York, NY, USA, 37–48. https://doi.org/10.1145/2851553.2851571

[43] Marija Selakovic and Michael Pradel. 2016. Performance Issues and Optimizations in JavaScript: An Empirical Study. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. ACM, New York, NY, USA, 61–72. https://doi.org/10.1145/2884781.2884829

[44] Aleksey Shipilev. 2018. *Reconsider defaults for fork count.* https://bugs.openjdk.java.net/browse/CODETOOLS-7902170

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

[46] Petr Stefan, Vojtěch Horký, Lubomír Bulej, and Petr Tůma. 2017. Unit Testing Performance in Java Projects: Are We There Yet?. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering* (L'Aquila, Italy) *(ICPE '17)*. ACM, New York, NY, USA, 401–412. https://doi.org/10.1145/3030207.3030226

[47] Luca Della Toffola, Michael Pradel, and Thomas R. Gross. 2018. Synthesizing Programs That Expose Performance Bottlenecks. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization* (Vienna, Austria) *(CGO 2018)*. Association for Computing Machinery, New York, NY, USA, 314–326. https://doi.org/10.1145/3168830

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

[49] Murray Woodside, Greg Franks, and Dorina C. Petriu. 2007. The Future of Software Performance Engineering. In *2007 Future of Software Engineering (FOSE '07)*. IEEE Computer Society, Washington, DC, USA, 171–187. https://doi.org/10.1109/FOSE.2007.32