Empirical Software Engineering (2023) 28:13
https://doi.org/10.1007/s10664-022-10247-x

Towards effective assessment of steady state performance in Java software: are we there yet?

Luca Traini[1] · Vittorio Cortellessa[1] · Daniele Di Pompeo[1] · Michele Tucci[2]

Accepted: 30 September 2022
© The Author(s) 2022

Abstract

Microbenchmarking is a widely used form of performance testing in Java software. A microbenchmark repeatedly executes a small chunk of code while collecting measurements related to its performance. Due to Java Virtual Machine optimizations, microbenchmarks are usually subject to severe performance fluctuations in the first phase of their execution (also known as warmup). For this reason, software developers typically discard measurements of this phase and focus their analysis when benchmarks reach a steady state of performance. Developers estimate the end of the warmup phase based on their expertise, and configure their benchmarks accordingly. Unfortunately, this approach is based on two strong assumptions: (i) benchmarks always reach a steady state of performance and (ii) developers accurately estimate warmup. In this paper, we show that Java microbenchmarks do not always reach a steady state, and often developers fail to accurately estimate the end of the warmup phase. We found that a considerable portion of studied benchmarks do not hit the steady state, and warmup estimates provided by software developers are often inaccurate (with a large error). This has significant implications both in terms of results quality and time-effort. Furthermore, we found that dynamic reconfiguration significantly improves warmup estimation accuracy, but still it induces suboptimal warmup estimates and relevant side-effects. We envision this paper as a starting point for supporting the introduction of more sophisticated automated techniques that can ensure results quality in a timely fashion.

Communicated by: Philipp Leitner

✉ Luca Traini
luca.traini@univaq.it

Vittorio Cortellessa
vittorio.cortellessa@univaq.it

Daniele Di Pompeo
daniele.dipompeo@univaq.it

Michele Tucci
tucci@d3s.mff.cuni.cz

[1] Department of Information Engineering, Computer Science and Mathematics, University of L'Aquila, L'Aquila, Italy

[2] Department of Distributed and Dependable Systems, Faculty of Mathematics and Physics, Charles University, Prague, Czech Republic

Published online: 28 November 2022

Keywords Microbenchmarking · Performance testing · Performance evaluation · Java · JMH

1 Introduction

Microbenchmarking is a form of lightweight performance testing, widely used to assess the execution time of Java software (Leitner and Bezemer 2017). Although less demanding than other performance testing techniques (e.g., load tests (Jiang and Hassan 2015)), Java microbenchmarking requires careful design (Costa et al. 2021; Georges et al. 2007; Kalibera and Jones 2013) to enable a reliable performance assessment. A key challenge is the inherent non-linearity of Java performance: The Java Virtual Machine (JVM) uses just-in-time compilation to translate "hot" parts of the Java code into efficient machine code at runtime (Barrett et al. 2017), leading to (often severe) performance fluctuations and potentially unstable results.

To tackle this problem, practitioners rely on the *assumption* that microbenchmarking is characterized by *two distinct phases*. During an initial *warmup phase*, the JVM determines which parts of the software under test would most benefit from dynamic compilation, then, in a subsequent phase, the benchmark reaches a *steady state* of performance. Based on that, benchmarks are typically designed to discard measurements of the *warmup phase* and focus on *steady state* performance (Barrett et al. 2017; Georges et al. 2007; Kalibera and Jones 2013). Java Microbenchmark Harness (JMH), i.e., the most popular Java microbenchmarking framework (Leitner and Bezemer 2017), leverages this concept and enables developers to manually *configure* the expected *warmup time* of a benchmark. Once launched, a JMH benchmark continuously executes the software under test for the configured warmup time, and, only after that, starts to collect steady state performance measurements.

Although considered the cornerstone of most of the current Java microbenchmarking practice, the *two-phase assumption* is not yet confirmed by empirical studies. Quite the opposite, there are indications that such an assumption oversimplifies the actual microbenchmarking behavior. In a recent study, Barrett et al. (2017) studied a set of small and deterministic benchmarks (Bagley et al. 2004; Bolz and Tratt 2015) across different types of VMs (including the JVM), and found that a relevant portion of benchmarks never hit the steady state. It is worth to notice that these benchmarks significantly differ from "software testing oriented" benchmarks (e.g., JMH). Indeed, they are not aimed at assessing specific software, rather, they are typically used as optimization targets by VM authors. Even more, they are generally more effectively optimized by VMs than average software (Ratanaworabhan et al. 2009). Despite the peculiarity of these benchmarks, the finding of Barrett et al. (2017) raises concerns on the current Java microbenchmarking practice, and calls for further empirical investigation.

Nevertheless, even when benchmarks consistently reach a steady state, performance assessment remains far from trivial. A key challenge is to effectively estimate *warmup time*. An overestimated warmup time may waste too much time, thereby potentially hampering the adoption of benchmarks in the Continuous Integration (CI) pipeline (Laaber et al. 2020; Traini 2022). On the other hand, an underestimated warmup time may easily mislead steady state performance assessment (Georges et al. 2007; Kalibera and Jones 2013).

The current state-of-practice mostly relies on software developers' knowledge to determine warmup time. Software developers estimate warmup time based on their expertise, and statically *configure* benchmark execution according to this estimation. Unfortunately,

no empirical studies so far have directly investigated the effectiveness of this practice for steady state performance assessment.

Recently, an alternative approach to static benchmark configuration has been proposed by Laaber et al. (2020). This approach, called *dynamic reconfiguration*, leverages stability criteria (He et al. 2019; Kalibera and Jones 2013) to automatically determine the end of the warmup phase at run-time. According to their results, when compared to JMH default configurations, dynamic reconfiguration can significantly reduce execution time with low impact on results quality. Despite these promising results, there is still little knowledge on the effectiveness of dynamic reconfiguration for steady state performance assessment. And even more, it is yet unclear whether such techniques can improve the effectiveness of the current practice, i.e., developer static configurations.

In the past years, Java microbenchmarks have been widely studied in the literature. Leitner and Bezemer (2017) found that JMH was one of the predominant microbenchmarking framework in the Java community. Costa et al. (2021) empirically studied five JMH bad practices. Laaber et al. (2019) performed an exploratory study on software microbenchmarking in the cloud. Samoaa and Leitner (2021) studied the impact of parameterization in JMH microbenchmarks.

Despite these efforts, there is still a lack of knowledge on the effectiveness of modern Java microbenchmarking for steady state performance assessment. In this paper, we aim to fill this gap by presenting the first comprehensive study that investigates steady state performance assessment in Java microbenchmarking. After an extensive experimentation of 586 JMH benchmarks from 30 Java systems, involving ∼9.056 billion benchmark invocations for an overall execution time of ∼93 days, we determined whether and when each benchmark reaches a steady state using an automated statistical approach by Barrett et al. (2017) based on changepoint analysis (Killick et al. 2012). At the time of writing, Barrett et al.'s approach represents one of the most advanced automated technique to determine steady state execution in Java benchmarks. Besides investigating whether benchmarks ever reach a steady state or not, we also comprehensively evaluated the effectiveness of the current state-of-practice and state-of-the-art. In particular, we investigated to what extent statically-defined developer configurations (i.e., state-of-practice), and dynamic reconfiguration techniques (i.e., state-of-the-art) are effective in ensuring a reliable and time-efficient assessment of steady state performance in JMH benchmarks. Even more, we quantified the potential side effects due to inaccurate warmup estimation both in terms of execution time waste and misleading performance measurements.

Laaber et al. (2020) have already investigated the effectiveness of dynamic reconfiguration. However, their study was mainly concerned with a particular aspect of Java microbenchmarking, i.e., reducing execution time. In this paper, instead, we aim to provide a comprehensive investigation on the effectiveness of dynamic reconfiguration (and developer static configurations) for steady state performance assessment. Due to our goal, we do not use JMH defaults as baselines, as done by Laaber et al., since there is no guarantee that they can effectively capture steady state performance. Indeed, although JMH defaults are a reasonable baseline for comparison,[1] their use may have downsides when studying steady state performance. Several studies have shown that benchmarks often reach their steady

[1] JMH defaults are configurations defined by JMH developers, which are undoubtedly experts in the field of Java microbenchmarking.

states in different numbers of iterations (Georges et al. 2007; Kalibera and Jones 2013), thus it may be misleading to use a unique (though reasonable) configuration as baseline, i.e., JMH defaults may be effective for some benchmarks and suboptimal for other ones. In order to avoid this problem, we leverage a different (and more rigorous) approach to assess dynamic reconfiguration effectiveness. We first use a state-of-the-art steady state detection technique to determine if/when a benchmark reaches a steady state of performance, then we base on its outcome to assess dynamic reconfiguration effectiveness. Besides this evaluation of dynamic reconfiguration techniques, this paper presents the following novel investigations. We present the first study that investigates if/when JMH benchmarks reach a steady state of performance. Second, we perform the first evaluation on the effectiveness of statically defined developer configurations. Third, we introduce the first comparison between the effectiveness of developer configurations (i.e., the current state-of-practice) and dynamic reconfiguration techniques (i.e., the current state-of-the-art).

Our results show that JMH benchmarks do not always reach a steady state of performance, thereby demystifying the current cornerstone of Java microbenchmarking, i.e., the *two-phase assumption*. This finding implies that practitioners may rely on measurements that are not representative of "actual" steady state performance. In addition, our results suggest that developer static configurations are often ineffective for warmup estimation, and may cause either improperly long execution times or misleading performance assessment. On the other hand, dynamic reconfiguration techniques show significant improvement over the current state of practice, but they still produce inaccurate estimates of the warmup time, hence causing time-consuming benchmark executions and distorted results. This finding highlights room for improvement for dynamic reconfiguration, and it calls for further research on this topic.

The main contributions of this paper are:

- a statistically rigorous investigation of steady state performance in JMH microbenchmarks.
- an empirical evaluation of developer static configurations in JMH microbenchmarks.
- a comprehensive comparison among the effectiveness of developer static configurations and state-of-the-art dynamic reconfiguration techniques.
- a large dataset of labeled benchmark executions to facilitate future research on Java steady state performance assessment, and foster further innovations in dynamic reconfiguration.

The remainder of this paper is organized as follows. Section 2 introduces steady state performance assessment and JMH microbenchmarks. Section 3 describes our research questions and Section 4 explains the experimental design. Section 5 reports the results. Section 6 discusses some implications of our findings. Section 7 describes threats to validity. Section 8 presents related work, and Section 9 concludes this paper.

2 Background

2.1 Steady State Performance

At the beginning of its execution, typically, a Java microbenchmark is slowly executed by the JVM. In a subsequent phase, the JVM detects "hots" (i.e., frequently executed) loops or methods, and it dynamically compiles them into optimized machine code. As a consequence, subsequent executions of those loops or methods (usually) become faster. Once

dynamic compilation is completed, the JVM is said to have finished warming up, and the benchmark is said to be executing at a *steady state of performance*.

Java microbenchmarking aims at assessing steady state performance (e.g., execution time) of Java software. The typical approach to collect steady state measurements is straightforward: A microbenchmark is executed for a certain number of times, and the first n benchmark executions are discarded (i.e., those related to the *warmup phase*) to prevent potentially misleading results. Unfortunately, the fixed number of n executions does not guarantee that warmup has ended. In order to investigate this issue, researchers started to develop data-driven methodologies to identify the end of the warmup phase. Prominent works in this regard are the methodologies proposed by Georges et al. (2007), and Kalibera and Jones (2013). The former uses preset thresholds on the coefficient of variation to determine whether the warmup phase is finished or not, while the latter leverages data visualization techniques (i.e., auto-correlation function plots, lag plots and run-sequence plots). Unfortunately, each one of these methodologies has its own drawback. Kalibera and Jones showed that the Georges et al.'s heuristic (2007) often fails to accurately determine the end of the warmup phase (Kalibera and Jones 2013). On the other hand, the methodology proposed by Kalibera and Jones (2013) is mostly based on a manual process, which typically implies some major limitations: (i) humans are prone to error/disagreement, (ii) manual analysis doesn't enable automation, and therefore it is not scalable.

To overcome these limitations, Barrett et al. (2017) recently proposed a novel automated technique based on change point detection (Eckley et al. 2011). The main advantage of this technique is that it provides a more rigorous approach compared to the Georges et al.'s simple heuristic, while still enabling a fully automated process (unlike Kalibera and Jones' approach). The Barrett et al.'s technique leverages a standard change point detection algorithm, namely PELT (Killick et al. 2012), to determine shifts in benchmark execution time. The identified shifts (i.e., change points) are then post-processed (e.g., by removing negligible performance shifts) to determine *if/when* a benchmark reaches a steady state of performance. To the best of our knowledge, this technique currently represents the state-of-the-art for steady state detection.

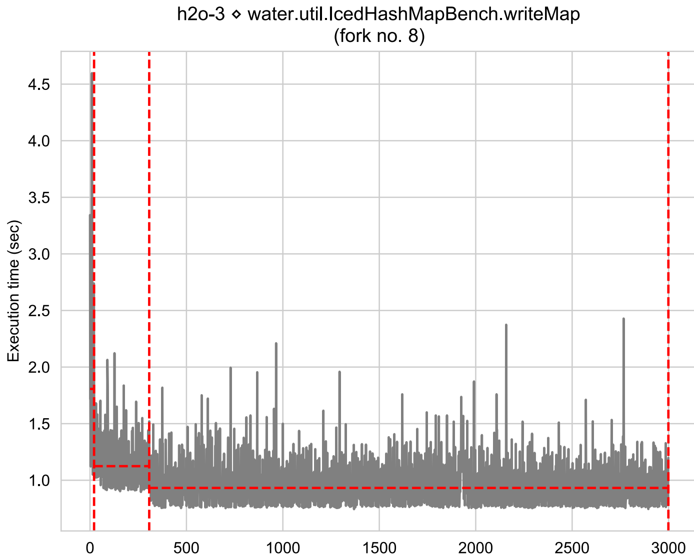
Figure 1a and b show two examples of benchmark executions along with the performance shifts identified by the PELT algorithm. The former consistently reaches a steady state of performance, while the latter doesn't.

2.2 Java Microbenchmark Harness (JMH)

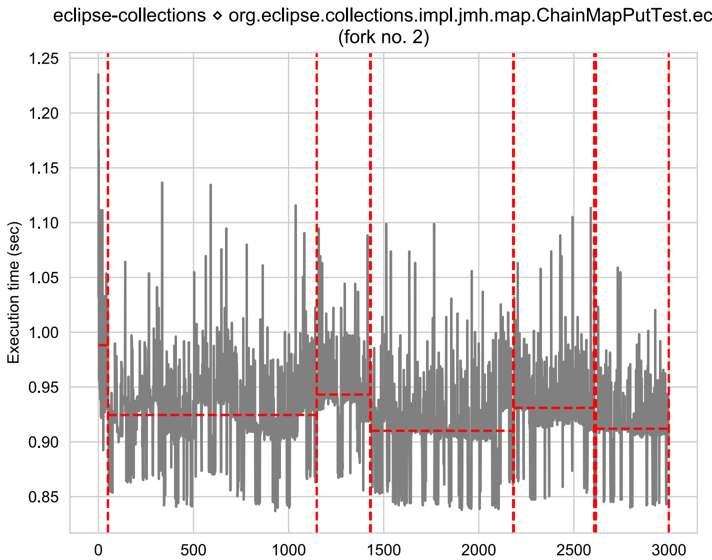
JMH is the de-facto standard framework for writing and executing microbenchmarks for Java software. It enables software developers to easily develop and execute microbenchmarks that measure fine-grained performance of specific units of Java code (e.g., methods). JMH supports steady state performance assessment by providing facilities that enable developers to statically *configure* the number of times each benchmark execution will be repeated (without compromising the reliability of results[2]).

Figure 2 depicts a typical JMH benchmark execution. JMH supports three different levels of repetitions: *forks*, *iterations* and *invocations*. *Invocations* (i.e., the lower level of repetition) are nominal benchmark executions that are continuously performed within a predefined amount of time, namely an *iteration*. In turn, a *fork* is constituted by a sequence of *iterations* performed on a fully clear instantiation of new JVM. Indeed, as suggested by best

[2] JMH code samples. Pitfalls of using loops in Java microbenchmarking. https://bit.ly/3lCqCZ4



(a) Example of benchmark execution that consistently reaches a steady state of performance.



(b) Example of benchmark execution that doesn't reach a steady state of performance.

Fig. 1 Two examples of benchmarks executions from our results. The grey line represents the execution times of each benchmark iteration visualized as a time-series. The x-axis represents the benchmark iteration number, while the y-axis represents the mean execution time within the iteration. Shifts in performance behaviour (i.e., changepoints) are indicated by dashed vertical red lines. The height of each dashed horizontal red line denotes the mean execution time within the changepoint segment. The plot titles show: the system that the benchmark belongs to (e.g., `h2o-3`), the benchmark name (e.g., `water.util.IcedHashMapBench.writeMap`), and the JMH fork number

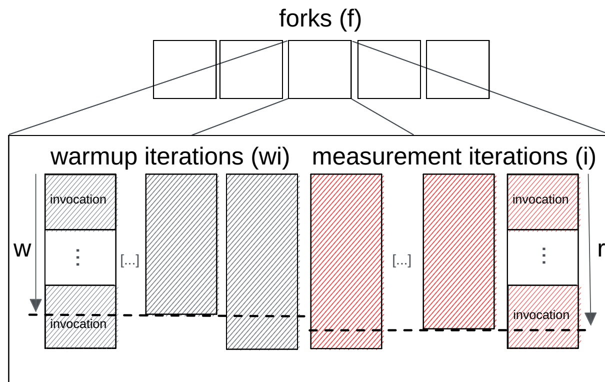


Fig. 2 The JMH microbenchmark life cycle

practices (Barrett et al. 2017; Costa et al. 2021; Georges et al. 2007; Kalibera and Jones 2013), iterations should be repeated multiple times on fresh JVM instantiations (i.e., forks) to mitigate the contextual effects of confounding factors.

Each fork is usually composed by two distinct types of iterations: *warmup* and *measurement iterations*. *Warmup iterations* are intended to bring the fork (i.e., the fresh JVM) into a steady state of performance, while *measurement iterations* are the ones where performance measurements are actually collected. Each measurement iteration typically returns a set of performance measurements (e.g., a sample of benchmark invocation execution times) or a performance statistic (e.g., average execution time or throughput).

JMH provides a set of configuration parameters to define the different levels of repetitions involved during microbenchmarking. These parameters include: warmup iteration time w, measurement iteration time r, warmup iterations wi, measurement iterations i, and forks f. Iteration time parameters (w and r) define the minimum time spent within an iteration. Given an iteration time w (resp. r), a warmup (resp. measurement) iteration will continuously perform benchmark *invocations* until the iteration time will expire. Warmup and measurement iterations (i.e., wi and i), instead, define the number of iterations performed within each fork. Finally, the fork parameter f defines the number of fresh JVM instantiation, i.e., the higher level of benchmark repetition.

Typically, Java developers directly set JMH configuration parameters on benchmark code through Java annotations. Nonetheless, when launching the benchmark, JMH allows to override developer configurations via Command Line Interface (CLI) arguments.

2.3 Dynamic Reconfiguration

JMH allows to statically define the expected length of the warmup phase using configuration parameters, such as warmup iteration time w and warmup iterations wi. Such estimation is typically performed on the basis of developer expertise and/or benchmark nature. Previous studies have shown that a static definition of the warmup time can be quite detrimental for steady state performance assessment (Barrett et al. 2017; Georges et al. 2007; Kalibera and Jones 2013). An alternative to JMH static configuration can be found in a recent approach called dynamic reconfiguration (Laaber et al. 2020). This approach is able to determine, during a JMH benchmark execution, whether the measurements appear to be stable, and more executions are unlikely to improve their accuracy. The rationale behind dynamic

reconfiguration is that, by using automated stability criteria to halt the benchmark execution, developers can save some of the time dedicated to performance testing while keeping an acceptable level of accuracy. To achieve this, the dynamic reconfiguration approach uses a sliding window to compare the last iterations to a stability criterion. Laaber et al. (2020) propose and evaluate three stability criteria to dynamically estimate the end of the warmup phase:

- *Coefficient of variation (CV)*[3]: CV is the ratio of the standard deviation to the mean, and it can be used to compare normally distributed data. Even when data is not normally distributed, as it is often the case for benchmark data, CV can still provide an estimate of measurement variability. A fork is considered stable when the difference between the largest and the smallest value of CV computed on the sliding window is within a fixed threshold.
- *Relative confidence interval width (RCIW)*: in this case, the variability in measurement data is estimated using a technique by Kalibera and Jones (2013, 2020) that employs hierarchical bootstrapping to compute the RCIW for the mean. The hierarchical levels are invocations, iterations, and forks.
- *Kullback-Leibler divergence (KLD)*: a technique described by He et al. (2019) to compute the probability that two distributions are similar based on the Kullback-Leibler divergence (KLD) (Kullback and Leibler 1951). In this case, the first distribution contains all the measurements in the sliding window excluding the last one, while the second distribution includes also the last measurement. As a consequence, stability is reached when the mean of the computed similarity probabilities is above some threshold.

In order to further reduce benchmark execution time, dynamic reconfiguration leverages the same stability criteria also to dynamically determine whether to execute the next fork or not.

3 Research Questions

In this work we aim to answer the following Research Questions (RQs):

RQ_1 *Do Java microbenchmarks reach a steady state of performance?*
RQ_2 *How does steady state impact microbenchmark performance?*
RQ_3 *How effective are developer configurations in assessing Java steady state performance?*
RQ_4 *How effective is dynamic reconfiguration in assessing Java steady state performance?*
RQ_5 *Does dynamic reconfiguration provide more effective warmup estimates than developers do?*

[3]It is worth to notice that, although both the approaches of Laaber et al. (2020) and Georges et al. (2007) are based on coefficient of variation (CV), they have some relevant differences. Indeed, Georges et al.'s heuristic uses a fixed threshold on CV, while Laaber et al.'s approach computes the difference between the minimum and the maximum CV in a sliding window of iterations, and it determines whether this difference exceeds a predefined threshold. Interestingly, Laaber et al. (2020) reported that the usage of the Georges et al.'s heuristic to dynamically estimate the end of the warmup is unrealistic for JMH microbenchmarks. (Indeed, in our experimental setup, it stops warmup iterations in only 46.7% of forks, when using a measurements window of 30, and a CV threshold of 0.02).

In the following subsections, we discuss in detail the motivation for each of the RQs and the methodology used to gather the answers. In Section 4, we describe the experimental setup along with the benchmarks used in our empirical study.

3.1 RQ_1—Steady State Assessment

With this research question, we aim to evaluate whether Java microbenchmarks reach a steady state of performance. We first use a state-of-the-art steady state detection technique (Barrett et al. 2017) to determine whether and when each fork reaches a steady state of performance (see Section 4.3 for details). Based on these results, we then classify each benchmark as either (i) *steady state*, if all the forks reach a steady state of performance, (ii) *no steady state*, if none of the forks reach a steady state or (iii) *inconsistent*, if the execution involves both *steady state* and *no steady state* forks.

We report the classification shares for both benchmarks and forks. Additionally, we report the percentages of *no steady state* forks for each benchmark.

3.2 RQ_2—Steady State Impact

With this research question, we want to investigate to what extent the attainment of a steady state impacts benchmark performance. To do so, we compare the measurements collected during steady state phases against those collected in non-steady phases of benchmark execution. We perform two different analyses: the first one investigates the difference between steady and non-steady performance within the same fork, the second one assesses the same aspect across different forks. Besides these two analyses, we also investigate two potential countermeasures to mitigate performance deviations in non-steady phases of benchmark execution.

All the aforementioned analyses involve a comparison between a set of steady measurements M^{stable} and a set of non-steady measurements $M^{unstable}$. In order to assess to what extent non-steady measurements differ from steady measurements, we use *relative performance deviation* (RPD).

In the following, we first explain the process we use to compute RPD. Then, we describe in the detail the four analyses we use to gather the answer.

3.2.1 Relative Performance Deviation

In order to quantify the relative performance deviation of $M^{unstable}$ compared to M^{stable}, we use the technique proposed by Kalibera and Jones (2013). The main benefit of this technique is that it provides a clear and rigorous account of the relative performance change and the uncertainty involved. For example, it can indicate that a set of execution time measurements is higher than another by $X\% \pm Y\%$ with 95% confidence. Following the guidelines of Kalibera and Jones (2013) and Kalibera and Jones (2020), we build each confidence interval using bootstrapping with random re-sampling and replacement (Davison and Hinkley 1997), with a confidence level of 95%. We run 10,000 bootstrap iterations. At each iteration, new realizations $\hat{M}^{unstable}$ and $\hat{M}^{stable}$ (respectively, of $M^{unstable}$ and M^{stable} measurements) are simulated and the relative performance change is computed. The simulation of the $\hat{M}^{unstable}$ new realizations randomly selects a subset of real data from $M^{unstable}$ with replacement. Similarly, $\hat{M}^{stable}$ is simulated by randomly sampling M^{stable}. The two means

($\mu^{unstable}$ and μ^{stable}) and the relative performance change (ρ) for simulated measurements are computed as follows:

$$\mu^{unstable} = \frac{\sum_{i=1}^{n} \hat{M}_i^{unstable}}{n}$$

$$\mu^{stable} = \frac{\sum_{i=1}^{m} \hat{M}_i^{stable}}{m}$$

$$\rho = \frac{\mu^{unstable} - \mu^{stable}}{\mu^{stable}}$$

where n is the number of measurements in $\hat{M}^{unstable}$, m the number of measurements in $\hat{M}^{stable}$. After the termination of all iterations, we collect a set of simulated realizations of the relative performance change $P = \{\rho_i \mid 1 \leq i \leq 10,000\}$ and estimate the 0.025 and 0.975 quantiles on it, for a 95% confidence interval. We consider a relative performance change as statistically significant if the confidence interval does not contain 0. For example, given a confidence interval of (0.05, 0.07), we can say that the mean execution time in $M^{unstable}$ is higher than the one in M^{stable} with a relative performance change that ranges between 5% and 7% with 95% confidence.

We leverage the confidence interval of the mean relative performance change (lb, ub) to compute the *relative performance deviation*:

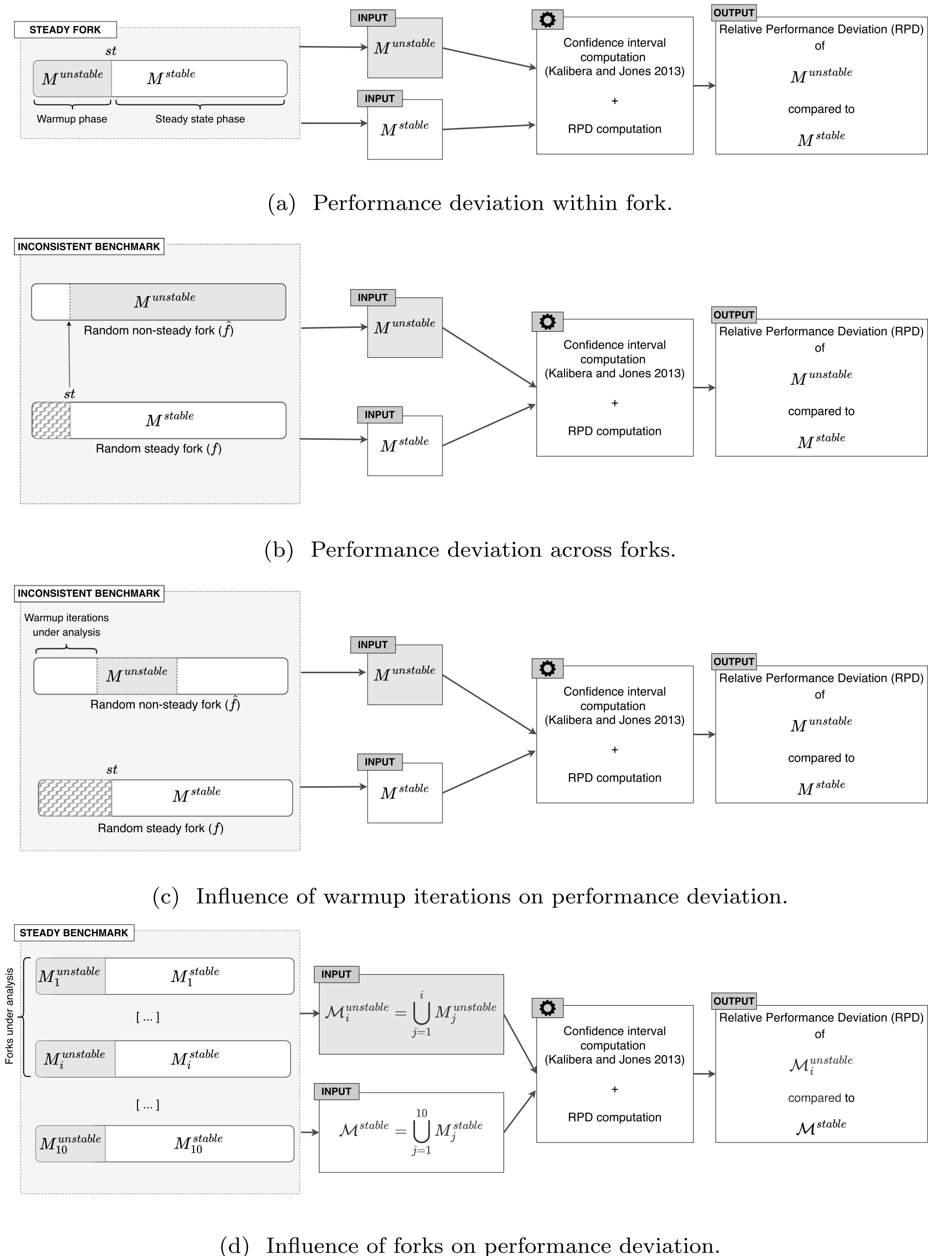
$$RPD = \begin{cases} 0 & \text{if } lb \leq 0 \leq ub \\ \left|\frac{lb+ub}{2}\right| & \text{otherwise} \end{cases}$$

In other words, we define RPD as the center of the 95% confidence interval of the mean relative performance change, if the interval doesn't contain zero. On the other hand, RPD evaluates to 0 if the confidence interval does not report a statistically significant performance change, i.e., the interval does contain 0. Higher values of RPD indicate that $M^{unstable}$ strongly deviates from M^{stable}.

3.2.2 Analyses

In order to analyze performance deviation within forks, we consider only forks that have reached a steady state of performance, and we analyze how performance changes when the steady state is reached. In particular, we partition the set of measurements of each fork in two distinct sets, namely M^{stable} and $M^{unstable}$, and we compare them to quantify the relative performance deviation (RPD). M^{stable} contains the measurements collected during steady state execution, i.e., those gathered after the *steady state starting time* st, while $M^{unstable}$ contains measurements collected before st. Figure 3a provides a graphical representation of this process for a given steady fork.

To investigate performance deviation across forks, instead, we assess how performance differs between steady and non-steady forks. To do that, we exclusively consider inconsistent benchmarks (i.e., the ones that contain both steady and non-steady forks), and we randomly pick from each benchmark a pair of forks $(f, \hat{f})$: one that has reached a steady state of performance (f) and one that does not ($\hat{f}$). We then use each pair to compare the measurements collected in the steady fork (M^{stable}) against those collected in the non-steady fork ($M^{unstable}$), and we quantify RPD. In order to enable a fair comparison, we



(a) Performance deviation within fork.

(b) Performance deviation across forks.

(c) Influence of warmup iterations on performance deviation.

(d) Influence of forks on performance deviation.

Fig. 3 RQ$_2$ analyses

use the same measurement window for both steady and non-steady forks. Namely, given a pair of forks $(f, \hat{f})$, we define $M^{unstable}$ (resp., M^{stable}) as the set of measurements collected in $\hat{f}$ (resp., f) after st, where st denotes the steady starting time of the steady fork f. Figure 3b depicts this process for one inconsistent benchmark.

Besides the aforementioned analyses, we investigate potential countermeasures to mitigate performance deviations of non-steady measurements. In particular, we analyze two specific aspects: number of warmup iterations and number of forks. With our analysis on warmup iterations, we aim to tackle performance deviations of non-steady forks, i.e., forks that never reach a steady state of performance. In particular, we want to assess to what extent an increase in the number of warmup iterations can mitigate performance deviations of non-steady forks. With our analysis on forks, instead, we aim to tackle deviations of non-steady measurements in forks that consistently reach a steady state of performance. That is, we investigate to what extent an increase in the number of forks mitigates performance deviations of measurements gathered during non-steady phases of benchmark execution.

In order to investigate the impact of warmup iterations, we first randomly sample one steady fork f and one non-steady fork $\hat{f}$ from each inconsistent benchmark. Then, for each pair $(f, \hat{f})$, we use a sliding window of 50 consecutive measurements ($M^{unstable}$) in $\hat{f}$, and we assess the deviation from the set of steady state measurements collected in f, namely M^{stable}. In particular, we partition the sequence of measurements gathered from $\hat{f}$ in 60 segments of equal size (i.e., 50 measurements), and we compute the relative performance deviation of each segment from M^{stable}. By doing so, we can assess whether increasing the number of warmup iterations can mitigate performance deviations of non-steady forks. Figure 3c shows the process used to compute the RPD for one specific segment ($M^{unstable}$) of $\hat{f}$, i.e., for a specific number of warmup iterations.

In order to investigate the influence of forks, instead, we assess how performance deviation of non-steady measurements changes when using different numbers of forks. Given the goal of this analysis, we consider only steady benchmarks, i.e., benchmarks that exclusively involve steady forks. For each benchmark, we progressively increment the number of considered forks and, at each increment i, we form a set $\mathcal{M}_i^{unstable}$ composed by all the non-steady measurements gathered from these forks. Then, we compute, for each set $\mathcal{M}_i^{unstable}$, the deviation from the entire set of steady measurements gathered from all the forks ($\mathcal{M}^{stable}$), as showed in Fig. 3d. In particular, we start by comparing the set of non-steady measurements collected from the first fork ($\mathcal{M}_1^{unstable}$) to the whole set of steady measurements $\mathcal{M}^{stable}$. Subsequently, we consider the set of non-steady measurements from the first two forks together ($\mathcal{M}_2^{unstable}$) and, again, we compare it to $\mathcal{M}^{stable}$. We proceed in this way for each set of non-steady measurements $\mathcal{M}_i^{unstable}$, with i ranging from 1 to 10. At the end of this process, we obtain one result (i.e., relative performance deviation) for each pair (b, i), where b denotes a steady benchmark and i denotes the number of considered forks. Through this analysis, we can assess if/how the increases in the number of forks mitigate the inherent deviations of non-steady measurements. It is worth to notice that we perform this analysis in an extreme situation, i.e., by considering exclusively non-steady measurements. We decided to do so because, if can we demonstrate that the increases in the number of forks can mitigate performance deviations in such an extreme case, then there are strong indications that they can effectively mitigate the impact of non-steady measurements.

3.3 RQ$_3$—Developer Configuration Assessment

With this research question, we aim to evaluate the effectiveness of developer configurations for steady state performance assessment. To do so, we assess how well software developers capture *steady state starting time* (*st*), i.e., the execution time required to reach a steady state of performance in a fork. Specifically, in each fork, we compare the *estimated warmup time* (*wt*) defined by software developers with the *steady state starting time st* detected by

the technique of Barrett et al. (2017). We measure the error of wt using *warmup estimation error* (WEE), i.e., how far wt is to the steady state starting time st:

$$WEE = |wt - st|$$

Additionally, we report the proportion of underestimated and overestimated forks (i.e., wt is respectively smaller or larger than st by at least 5 s), and we investigate potential side effects.

An overestimated warmup phase wastes execution time, since it leads to a surplus of warmup iterations, which, in turn, delay the beginning of measurement iterations. This inevitably increases benchmark execution time and causes potentially harmful practical implications. For example, the "time effort" of a performance test is often considered a critical factor when selecting the tests to execute before a software release (Traini 2022; Chen and Shang 2017). In that, an overestimated warmup phase can hamper the adoption of microbenchmarks for continuous performance assurance, especially when software releases happen frequently (Rubin and Rinard 2016), and tests are executed as part of a Continuous Integration (CI) pipeline (Fowler 2006). To investigate side effects of overestimated warmup time, we use *time waste*, i.e., the overestimation error induced by developer configurations. We quantify *time waste* through the difference (in terms of time) between wt and st. In other words, *time waste* represents the execution time that can be potentially saved in a fork.

On the other hand, an underestimated warmup phase may easily mislead steady state performance assessment. Indeed, unstable measurements (i.e., measurements gathered before st) may distort performance results, thereby leading to potentially wrong conclusions (Georges et al. 2007; Kalibera and Jones 2013). We assess the impact of underestimated warmup time using *relative performance deviation* (RPD), i.e., the magnitude of performance deviation compared to steady state measurements. That is, for each underestimated fork, we compare performance measurements in the steady state (M^{stable}) with those in the measurement time window defined by software developers (M^{conf}) using RPD. A high RPD indicates that performance measurements used by software developers strongly deviate from those collected in the steady state.

Besides investigating the effectiveness of developer configurations at fork level, we also analyze their effectiveness at benchmark level, i.e., across multiple forks. Through this analysis, we aim to evaluate developer configurations not only in terms of warmup and measurement iterations, but also based on the number of configured forks. To do that, we consider the entire set of measurements gathered through developer configuration (i.e., across the configured forks), and we compare it to the whole set of steady measurements gathered across all the steady forks of the benchmark. In particular, for each benchmark, we assess the extent to which the set $\mathcal{M}^{conf}$ of developer measurements deviate from the set $\mathcal{M}^{stable}$ of steady measurements. In addition, we report the time effort of benchmarks based on developer configurations by computing their overall execution time.

3.4 RQ$_4$—Dynamic Reconfiguration Assessment

With this research question, we aim to evaluate the effectiveness of dynamic reconfiguration approaches (Laaber et al. 2020) for steady state performance assessment. These approaches dynamically determine the warmup time during benchmark execution using stability criteria. In our evaluation, we consider three stability criteria proposed by Laaber et al. (2020): (i) Coefficient of variation (CV), (ii) Relative confidence interval width (RCIW), and (iii) Kullback-Leibler divergence (KLD).

In order to assess the effectiveness of each dynamic reconfiguration variant (i.e., stability criteria), we use the same methodology adopted in RQ_3. We first report the *warmup estimation error* (WEE) of forks, along with the proportion of underestimated and overestimated warmup time. Then, we consider the potential side effects of wrong warmup estimation using *time waste* and *relative performance deviation* (RPD). Finally, we investigate the effectiveness of dynamic reconfiguration at benchmark level (i.e., across forks) by assessing their RPDs and the overall execution time.

3.5 RQ_5—Dynamic vs Developer Configurations

To answer RQ_5, we compare dynamic reconfiguration techniques against developer configurations using three evaluation metrics: (i) *warmup estimation error*, (ii) *estimated warmup time*, and (iii) *relative performance deviation*.

Warmup estimation error (WEE) measures the accuracy of wt, i.e., how close wt is to the steady state starting time st. Lower values of WEE indicate better estimates of the warmup time.

Estimated warmup time (wt) measures the time spent to warmup a fork with respect to a specific configuration. Higher values of wt increase the time effort devoted to performance testing, and, therefore, can potentially hamper the adoption of benchmarks for continuous performance assessment.

Relative performance deviation (RPD), instead, measures in each fork the magnitude of performance deviation of M^{conf} compared to steady state performance measurements (M^{stable}). Higher values of RPD indicate that M^{conf} strongly deviates from M^{stable}, thus implying that performance measurements determined by the configuration may potentially mislead steady state performance assessment. As a complementary analysis, we further use RPD to measure performance deviations at benchmark level. That is, we quantify (for each benchmark) the magnitude of deviation of the entire set of measurements gathered through developers/dynamic configurations ($\mathcal{M}^{conf}$) when compared to the whole set of steady measurements collected through the entire benchmark execution ($\mathcal{M}^{stable}$).

The results of WEE, wt, and RPD are compared using the Wilcoxon Rank-Sum test (Cohen 2013), which is a non-parametric test that makes no assumption about underlying data distribution, hence, raises the bar for significance for both normally and non-normally distributed data. Additionally, a standardized non-parametric effect size measure, namely the Vargha Delaney's $\hat{A}_{12}$ statistic (Vargha and Delaney 2000), is used to assess the effect size. Given a dynamic reconfiguration technique D, $\hat{A}_{12}$ measures the probability of D performing better than developer configurations with reference to a specific evaluation metric. $\hat{A}_{12}$ is computed using (1), where R_1 is the rank sum of the first data group we are comparing, and m and n are the numbers of observations in the first and second data sample, respectively.

$$\hat{A}_{12} = \frac{\left(\frac{R_1}{m} - \frac{m+1}{2}\right)}{n} \tag{1}$$

We interpret $\hat{A}_{12}$ using the thresholds provided by Vargha and Delaney (2000). Based on (1), if dynamic configurations and developer configurations are equally good, $\hat{A}_{12} = 0.5$. Respectively, $\hat{A}_{12}$ higher than 0.5 means that dynamic reconfiguration is more likely to produce better results. The effect size is considered small for $0.56 \leq \hat{A}_{12} < 0.64$, medium for $0.64 \leq \hat{A}_{12} < 0.71$, and large for $\hat{A}_{12} \geq 0.71$. On the other hand, $\hat{A}_{12}$ smaller than 0.5 means that developer configurations provide better results. In this case, the effect size

is considered small for $0.34 > \hat{A}_{12} \leq 0.44$, medium for $0.29 > \hat{A}_{12} \leq 0.34$, and large for $\hat{A}_{12} \leq 0.29$.

In order to avoid misleading interpretations, we perform transformation on the $\hat{A}_{12}$ effect size (Neumann et al. 2015), since we consider values of WEE smaller than 5 s and RPD smaller than 5%, as negligible (Maricq et al. 2018), and, therefore, "equally good". In particular, we apply *Pre-Transforming Data* (Neumann et al. 2015) by replacing each value of $WEE < 5sec$ and $RPD < 0.05$ with zero. When it comes to *estimated warmup time* (wt), no transformation is performed on the $\hat{A}_{12}$ effect size, since we are interested in any improvement (Neumann et al. 2015; Sarro et al. 2016).

4 Experimental Design

In order to answer our research questions, we first collect performance measurements from the execution of 586 microbenchmarks across 30 systems. Then, we analyze collected measurements to determine whether and when each fork reaches a steady state of performance. Finally, we perform post-hoc analysis on the collected measurements to assess developers and dynamic configurations.

In this section, we first describe the microbenchmarking setup we use to collect performance measurements and the benchmark subjects. Then, we describe in detail the steady state detection technique used in our empirical study. Finally, we present the process we use to extract both developer and dynamic configurations.

4.1 Microbenchmarking Setup

Following the methodology used in the study of Barrett et al. (2017), we execute each benchmark for a substantially longer time than "usual" JMH configurations (on average 171 times longer than developer configurations). We perform 10 JMH forks for each benchmark (as suggested by Barrett et al. (2017)), where each fork involves an overall execution time of at least 300 s and 3000 benchmark invocations. To do so, we configure the execution of each benchmark via JMH CLI arguments. Specifically, we configure 3000 measurement iterations (`-i 3000`) and 0 warmup iterations (`-wi 0`) to collect all the measurements along the fork. Each iteration continuously executes the benchmark method for 100ms (`-r 100ms`).[4] The number of fork is configured to 10 (`-f 10`). As benchmarking mode, we use sample (`-bm sample`), which returns nominal execution times for a sample of benchmark invocations within the measurement iteration.

The execution environment and external events occurring during the benchmark runs have a remarkable influence on the accuracy of results. This is especially true when executing microbenchmarks, as they tend to measure small portions of code that may last less than a microsecond and are, therefore, more prone to be affected by even small changes in the environment. Hence, we tried to control as many sources of variability as possible in order to obtain more reliable measurements.

We disabled Intel Turbo Boost, i.e., a feature that automatically raises the CPU operating frequency when demanding tasks are running (Suchanek et al. 2017). We also disabled

[4] We chose 100ms as iteration time because this value enables us to "replicate" every possible configuration considered in our study (see Section 4.4 for more details on how we obtain the set performance measurements for a particular JMH configuration).

hyper-threading, i.e., a feature in modern processors that executes two threads simultaneously on the same physical core (Suchanek et al. 2017). This is achieved by replicating the architectural state but sharing execution resources such as ALUs and caches. For this reason, hyper-threading may lead to contention patterns that continuously vary during the execution.

Another potential cause of variability among repeated runs is represented by Address Space Layout Randomization (ASLR), which is a security technique to randomly arrange the address space positions at each execution. We disabled ASLR as it may cause variability in the measurements from one fork to another.

The amount of available memory can also affect execution times. We fixed (through the `-Xmx` flag) the total amount of heap memory available to the JVM to 8GB, because this is the most important factor affecting garbage collection performance. In fact, the throughput of garbage collections is inversely proportional to the amount of memory available, since collections occur when memory fills up (Oaks 2014).

A large variety of operating system events may have a noticeable impact on execution times because they increase context switching in most cases. For this reason, we tried to keep the events that are not related to the benchmarks to a minimum. We disabled any Unix daemon that is not strictly necessary. We also disabled SSH logins for the entire duration of the experiments. To further reduce context switching, we used priority scheduling and increased the *niceness* of the JMH process running the benchmarks and all its children.

Finally, we ensured that the state of the system was consistent at each run by monitoring the *dmesg* log and the *systemd* journal for anomalies, as well as the shell environment of the process for changes in size (Mytkowicz et al. 2009a).

The benchmarks were executed on a bare metal server running Linux Ubuntu 18.04.2 LTS on a dual Intel Xeon E5-2650v3 CPU at 2.30GHz, with a total of 40 cores and 80GiB of RAM.

4.2 Subject Benchmarks

Table 1 reports the list of the 30 Java open source systems we use in this study. We selected such systems because they are relatively popular (i.e., they have more than 100 Github stars), have non-trivial JMH suites (i.e., have at least 20 benchmarks), and span different domains (e.g., application servers, logging libraries, databases). Given the large size of the benchmark suites, we randomly sample 20 benchmarks for each system. 14 out of the 600 sampled benchmarks failed in our experimental setup.[5]

Overall, in our empirical study, we assess the behavior of 586 randomly sampled benchmarks across 30 Java systems.

In order to investigate the correctness of the benchmarks selected for this study, we ran the SpotJMHBugs tool by Costa et al. (2021) on the systems. The tool was able to detect only one potential bad practice, of type LOOP, in the `netty` project. Therefore, we consider our selection of benchmarks suitable for the study.

4.3 Steady State Detection

Detecting the end of the warmup phase, and consequently the start of the steady state, is no trivial task, as the notion of "*steady*" resides on how much stability of results one wants to

[5]The 14 failed benchmarks are distributed as follows: 3 `cantaloupe`, 1 `jetty.project`, 4 `vert.x`, 3 `hazelcast` and 5 `jbdi`

Table 1 This table reports the name of each subject system, the Github organization, the number of Github stars, and the number of benchmarks in the performance testing suite

Organization	Name	Stars	No. Benchmarks
apache	arrow	8,065	46
raphw	byte-buddy	4,311	39
apache	camel	3,771	35
cantaloupe-project	cantaloupe	195	103
prometheus	client_java	1,545	33
crate	crate	3,109	39
eclipse	eclipse-collections	1,711	2,415
h2oai	h2o-3	5,400	73
hazelcast	hazelcast	4,406	144
HdrHistogram	HdrHistogram	1,871	75
apache	hive	3,781	1,402
imglib	imglib2	240	25
JCTools	JCTools	2,697	172
jdbi	jdbi	1,533	76
eclipse	jetty.project	3,147	212
jgrapht	jgrapht	1,927	91
apache	kafka	19,224	3,578
zalando	logbook	900	20
apache	logging-log4j2	1,207	572
netty	netty	26,984	1,746
prestodb	presto	12,153	1,534
protostuff	protostuff	1,649	31
r2dbc	r2dbc-h2	128	20
eclipse	rdf4j	251	132
RoaringBitmap	RoaringBitmap	2,281	1,620
ReactiveX	RxJava	44,802	1,302
yellowstonegames	SquidLib	364	334
apache	tinkerpop	1,351	57
eclipse-vertx	vert.x	12,177	41
openzipkin	zipkin	14,468	63

achieve during performance testing or, to put it in another way, how much variability one is willing to tolerate. Nonetheless, any analyst who wants to study steady state performance must establish the length of the warmup phase. In our study, we are interested in automatically detecting the length of such phase to determine whether and when a benchmark reached a steady state.

For this task, we build on the steady state detection approach proposed by Barrett et al. (2017), which we adapted to the purposes of our study. The approach is fully automated, and it is based on changepoint analysis (Eckley et al. 2011), which is a statistical technique to detect shifts in timeseries data. In our experimental setup, each datapoint of the timeseries represents the average execution time within a JMH iteration, and the whole timeseries represents a fork.

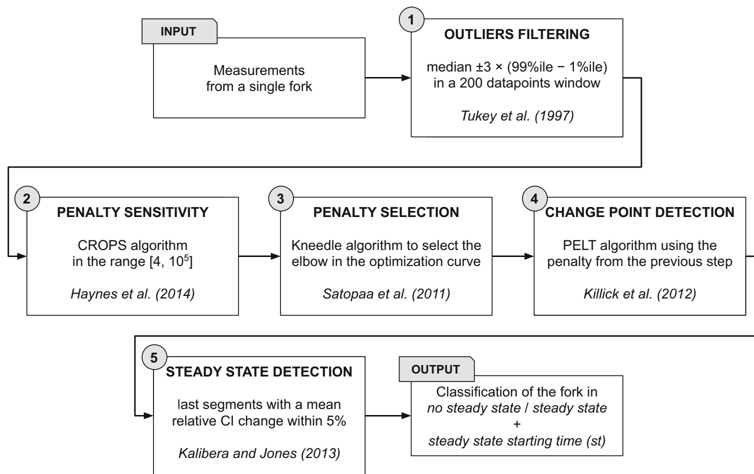


Fig. 4 Steady state detection process

An overview of the approach can be found in Fig. 4. As a first step of the approach, we identify and remove potential outliers as we are only interested in detecting shifts in execution time that appear to stay consistent for a period of time. As done by Barrett et al. (2017), we use the method by Tukey and et al (1977) to identify as outliers the datapoints that lie outside the median $\pm 3 \times (99\%ile - 1\%ile)$ in a 200 datapoints window ①. Out of the 1.8×10^7 datapoints in the study, 0.27% are classified as outliers, with the most of any fork being 1.3%.

After filtering outliers, we apply a changepoint algorithm to detect shifts in execution time. Changepoint algorithms are designed to divide the entire timeseries into segments, within which the behavior of the timeseries is considered to remain unchanged. On the basis of the segmentation of the timeseries we can detect if and when a benchmark execution reached a steady state.

The specific changepoint algorithm we used is called *PELT* (Killick et al. 2012). We applied the algorithm to timeseries data gathered from individual forks in order to detect changes in both the mean and the variance of execution time. An important parameter of the PELT algorithm is the *penalty*, namely an argument designed to avoid under/over-fitting and, therefore, directly impacting the number of changepoints the algorithm will detect. The higher the penalty value, the more difficult will be for the algorithm to detect change-points. Conversely, lower penalty values will result in more changepoints. Barrett et al. set this parameter to $15 \log(n)$, where n is the number of datapoints in the timeseries after discarding the outliers. We concluded that a single penalty value for all the timeseries (i.e., all the forks in the experiment) was not suitable for tuning the algorithm to the differences we found among benchmark execution data. As a consequence, we decided to employ a method to derive an appropriate penalty value for each timeseries: we used the *CROPS* algorithm (Haynes et al. 2014) to efficiently generate, for each timeseries, optimal change-point segmentations for all penalty values in a continuous range ($[4, 10^5]$, in our case) ②. The number of changepoints in the alternative segmentations and the corresponding penalty

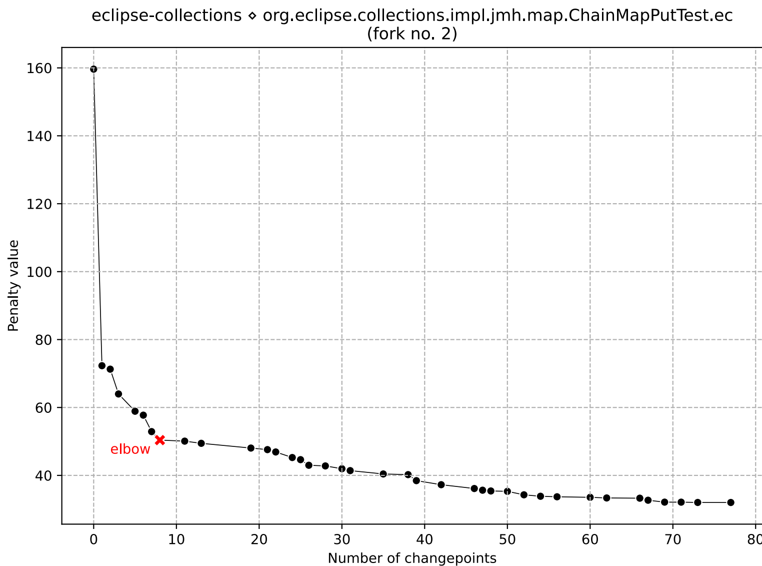


Fig. 5 Example of the selection procedure of a penalty value from an elbow diagram

values can be used to derive an optimization curve (sometimes referred to as *elbow diagram*). Figure 5 shows an example of such a curve computed on the timeseries in Fig. 1b. As suggested by Lavielle (2005), this diagram can be visually inspected to find suitably parsimonious penalty values in the area of the *elbow*. A penalty point in the elbow area can be automatically selected using the *Kneedle* algorithm (Satopaa et al. 2011), which is a method to find the point of maximum curvature in the continuous approximation of an optimization curve ③. A red x in Fig. 5 marks the point in the curve that was chosen by the *Kneedle* algorithm in that case. Using this procedure, we were able to automatically derive a different penalty value to guide the segmentation of each fork.

Once we obtain a segmentation of the timeseries ④, we can proceed to detect a possible steady state. A steady state should be detected when the execution time is reasonably stable after the end of the warmup phase. It is a matter of interpretation how much the execution time is allowed to vary but, following the approach from Barrett et al., we consider a fork to have reached a steady state if the last 500 measurements are contained in a single segment (i.e., no changepoint was detected within the last 500 datapoints). In general, to find when the steady state was first reached we could just take the start of the last segment. However, this would not consider practical cases in which the smallest variation in mean or variance would generate different segments, even if, from a performance evaluation point of view, the benchmark has completed its warmup phase. Therefore, we need to establish a suitable tolerance to allow the steady state period to span multiple segments whose variation is not meaningful to determine the execution time of the benchmark. In Barrett et al. the tolerance was provided by combining the maximum number of consecutive segments from the last one, such that a segment s_i is equivalent to the final segment s_f if mean(s_i) is within (mean(s_f) $\pm$ max(variance(s_f), 0.001s)). We did not intend to apply a fixed threshold in units of time (like 0.001 s) because our benchmarks vary from tens of nanoseconds to few seconds, nor we wanted to use the variance of the last segment as a threshold because it

leads to an extremely low tolerance for most of the benchmarks.[6] Hence, we preferred to compare the segments by applying a 5% tolerance on the confidence interval for the mean relative performance change, by using the approach of Kalibera and Jones (2013) ⑤ (see Section 3.3 for details).

On the basis of this information, we can classify individual forks as *steady state* if we were able to detect a steady state, or *no steady state* when the opposite occurred. Moreover, we classify a benchmark as *steady state* if all its forks reached a steady state, *no steady state* if all its forks did not reach a steady state, and *inconsistent* if at least one fork was classified as *steady state* while at least another one was classified as *no steady state*. The same information is used to derive the *steady state starting time* (*st*), which is the beginning of the first segment s_i that is considered to be steady. Consequently, for a given fork, the set of measurements in the range between *st* and the end of the timeseries is the set of the steady state performance measurements (M^{stable}) of our experiment. Based on that, given a specific benchmark, we can further define the entire set of steady measurements $\mathcal{M}^{stable}$ as the union of the steady measurements sets M^{stable} collected across all the steady forks of the benchmark.

4.4 Benchmark Configurations

Each benchmark configuration determines for each fork two relevant pieces of information: the *estimated warmup time* (*wt*) and a set of performance measurements M^{conf}.

We use this information to (i) compute warmup estimation error (WEE), (ii) determine whether a configuration underestimates or overestimates the *steady state starting time* (*st*), and (iii) assess potential side effects due to wrong estimation (i.e., *time waste* or *relative performance deviation*).

In the following, we describe the process we use to derive *wt* and M^{conf} for each benchmark fork, for both developer and dynamic configurations.

```
Data: warmup iteration time w, no. warmup iterations wi, fork measurements M
Result: Overall warmup time wt
wt ← 0 ;
sit ← 0 ;                                  ▷ initialize simulated iteration time
sic ← 0 ;                                  ▷ initialize simulated iterations count
for e in M do
    ni ← ⌈e/100ms⌉ ;                       ▷ compute no. benchmark invocations
    it ← ni · e ;                          ▷ estimate time spent in the iteration
    add it to sit ;
    if sit ≥ w then
        add sit to wt ;
        sit ← 0 ;
        increment sic by 1 ;
    if sic ≥ wi then
        break ;
return wt ;
```

Algorithm 1 Estimate overall warmup time.

[6] In 60% of benchmarks forks, the ratio between the variance and mean of last segment is smaller than 0.0000002. Using variance as a threshold in these forks, it would imply that any negligible performance shift larger than 0.00002% would be considered as a meaningful performance change.

```
Data: overall warmup time wt, measurement iteration time r, no. measurement iterations i, fork
      measurements M
Result: Selected measurements according to configuration M^conf
t ← 0 ;                                                   ▷ initialize simulated time
sit ← 0 ;                                                 ▷ initialize simulated iteration time
sic ← 0 ;                                                 ▷ initialize simulated iterations count
initialize empty list M^conf ;
for e in M do
    ni ← ⌈e/100ms⌉ ;                                      ▷ compute no. benchmark invocations
    it ← ni · e ;                                         ▷ estimate time spent in the iteration
    if t ≥ wt then
        add it to sit ;
        append e to M^conf ;
        if sit ≥ r then
            sit ← 0 ;
            increment sic by 1 ;
        if sic ≥ i then
            break ;
    add e to t ;
return M^conf ;
```

Algorithm 2 Select performance measurements.

4.4.1 Software Developer Configurations

Software developers define JMH configurations in benchmark code through Java annotations. When a benchmark is launched, JMH executes the benchmark according to developer configurations (e.g., no. measurements and warmup iterations). A trivial approach to obtain both wt and the set of performance measurements M^{conf} for each fork would be to simply run the benchmark, and extract this information from the execution logs and JSON result files produced by JMH. Unfortunately, this approach would be extremely expensive in terms of time, and would not fit our own needs. Instead, we use post-hoc analysis: We first obtain the JMH configurations as defined by developers, then we use this information to compute both wt and M^{conf} based on the performance measurements collected in our microbenchmarking setup (see Section 4.1).

In order to obtain developer configurations, we leverage a JMH feature that allows to overwrite configurations on-the-fly via CLI arguments.[7] We exploit this capability to reduce benchmark execution time and speed-up developer configurations retrieval. We first execute each benchmark twice while reducing execution time through JMH CLI arguments. Then, we retrieve developer configurations in the JSON result files of each individual execution. Specifically, we first obtain the number of measurement and warmup iterations (i.e., wi and i), and the number of forks f by executing each benchmark while setting the measurement and warmup time of each iteration to 1 nanoseconds (`-w 1ns-r 1ns`). Then, we retrieve the measurement and warmup time (i.e., w and r) of each iteration by running

[7] We use dynamic analysis (i.e., running benchmarks by overwriting CLI arguments) instead of static analysis because this methodology ensures a better coverage and lower margin of errors. Indeed, developers may rely on other mechanisms than JMH annotations to configure benchmarks, (e.g., see `OptionsBuilder` at https://bit.ly/3OkxzJS). Our approach allows to safely retrieve configurations also in these cases, while this would have been impractical through static analysis.

each benchmark while setting the number of forks, warmup and measurement iterations to 1 (`-f 1 -wi 1 -i 1`). At the end of this process, we obtain a tuple (w, wi, r, i, f), where w denotes the time of a warmup iteration, wi denotes the number of warmup iterations, r denotes the time of a measurement iteration, i the number of measurement iterations and f the number of forks. We exploit w, wi, r and i along with the fork measurements M (as collected in our microbenchmarking setup) to compute, for each fork, the *estimated warmup time* (wt) and the set of performance measurements M^{conf}. Specifically, we estimate the time spent in each warmup/measurement iteration using the average execution time of each iteration as observed in our microbenchmarking setup; then, we derive wt and M^{conf} based on the JMH configuration. We report the detailed process to obtain both wt and M^{conf} in Algorithm 1 and Algorithm 2, respectively.

Based on the above, we can exploit the number of configured forks f defined by software developers to obtain the whole set of performance measurements gathered from the entire benchmark execution, namely $\mathcal{M}^{conf}$. In other words, we derive $\mathcal{M}^{conf}$ by joining all the sets of measurements M^{conf} gathered from the first f forks of the benchmark.

4.4.2 Dynamic Configurations

In order to obtain (w, wi, r, i, f) for each dynamic reconfiguration variant, we leverage the replication package provided by Laaber et al. (2020). Specifically, we use the scripts provided for post-hoc analysis, which take as input JMH result JSON files, and return, for each fork, the number of warmup iterations (wi) according to stability criteria. The warmup iteration time w, the measurement time r, and the number of measurement iterations i are fixed to $w = 1s$, $r = 1s$ and $i = 10$, respectively, according to the experimental setup defined in Laaber et al. (2020).

We then obtain wt and M^{conf} using the same approach adopted for developer configurations.

5 Results

This section presents the results of the experiments and provides answers to the RQs formulated in Section 3.

5.1 RQ$_1$—Steady State Assessment

As described in Section 4.3, we classify the benchmarks in our study on the basis of their ability to eventually reach a steady state. Such classification is first performed at fork level, and then at benchmark level by combining results from the steady state detection on forks.

In order to provide an overview of how many forks reached a steady state, Fig. 6a reports the percentage of forks classified as *steady state* or otherwise, as grouped by systems. The percentage of forks that reached steady state varies between 69% (`JCTools`) and 98.9% (`cantaloupe`). Even if there is some variability among systems, 28 of them, out of the 30 in our study, show a percentage of steady state forks above 80%, with 18 of them above 90%. Globally, in most cases (89.1% in the last row of Fig. 6a), individual forks were able to reach a steady state according to our detection technique.

When we examine how the classified forks are distributed among the benchmarks, we get a less obvious outlook. In Fig. 6b, we report the percentage of benchmarks classified as *steady state*, or *inconsistent* in the respective systems (note we do not report percentages for

	no steady state	steady state
arrow	6.0%	94.0%
byte-buddy	7.5%	92.5%
camel	9.5%	90.5%
cantaloupe	1.1%	98.9%
client_java	11.0%	89.0%
crate	14.5%	85.5%
eclipse-collections	14.0%	86.0%
h2o-3	6.5%	93.5%
hazelcast	7.1%	92.9%
HdrHistogram	19.5%	80.5%
hive	2.0%	98.0%
imglib2	6.0%	94.0%
JCTools	31.0%	69.0%
jdbi	19.3%	80.7%
jetty.project	11.6%	88.4%
jgrapht	18.5%	81.5%
kafka	23.5%	76.5%
logbook	8.5%	91.5%
logging-log4j2	15.5%	84.5%
netty	8.0%	92.0%
presto	9.0%	91.0%
protostuff	8.0%	92.0%
r2dbc-h2	3.5%	96.5%
rdf4j	3.5%	96.5%
RoaringBitmap	5.0%	95.0%
RxJava	10.0%	90.0%
SquidLib	14.0%	86.0%
tinkerpop	8.0%	92.0%
vert.x	10.0%	90.0%
zipkin	17.5%	82.5%
Total	10.9%	89.1%

(a) Forks classification.

	inconsistent	steady state
arrow	30.0%	70.0%
byte-buddy	40.0%	60.0%
camel	45.0%	55.0%
cantaloupe	10.5%	89.5%
client_java	55.0%	45.0%
crate	60.0%	40.0%
eclipse-collections	50.0%	50.0%
h2o-3	40.0%	60.0%
hazelcast	41.2%	58.8%
HdrHistogram	65.0%	35.0%
hive	15.0%	85.0%
imglib2	20.0%	80.0%
JCTools	80.0%	20.0%
jdbi	60.0%	40.0%
jetty.project	52.6%	47.4%
jgrapht	45.0%	55.0%
kafka	75.0%	25.0%
logbook	45.0%	55.0%
logging-log4j2	60.0%	40.0%
netty	25.0%	75.0%
presto	25.0%	75.0%
protostuff	40.0%	60.0%
r2dbc-h2	30.0%	70.0%
rdf4j	5.0%	95.0%
RoaringBitmap	20.0%	80.0%
RxJava	40.0%	60.0%
SquidLib	70.0%	30.0%
tinkerpop	50.0%	50.0%
vert.x	56.2%	43.8%
zipkin	60.0%	40.0%
Total	43.5%	56.5%

(b) Benchmarks classification.

Fig. 6 RQ_1. Steady state classification

"no steady state" classification, since we didn't find any benchmark classified as such). The first clear result is that there are no cases in which all the forks of a benchmark did not reach a steady state, since the totality of benchmarks is always distributed among the *steady state* and *inconsistent* columns. On the one hand, this might encourage the assumption that, in the vast majority of cases, benchmarks reach and measure steady state performance. On the other hand, we can assess that the percentage of benchmarks in which all the forks reached a steady state is subject to large variability depending on the specific system. In fact, the percentage of *steady state* benchmarks varies between 20% (`JCTools`) and 95%(`rdf4j`). Only 5 systems overcome a 80% percentage and, as opposed to what one would expect, *no steady state* forks are unevenly distributed among benchmarks, thus causing most systems to have a low percentage of *steady state* benchmarks even though the percentage of *steady state* forks was higher. Since there are no benchmarks in which all the forks did not reach

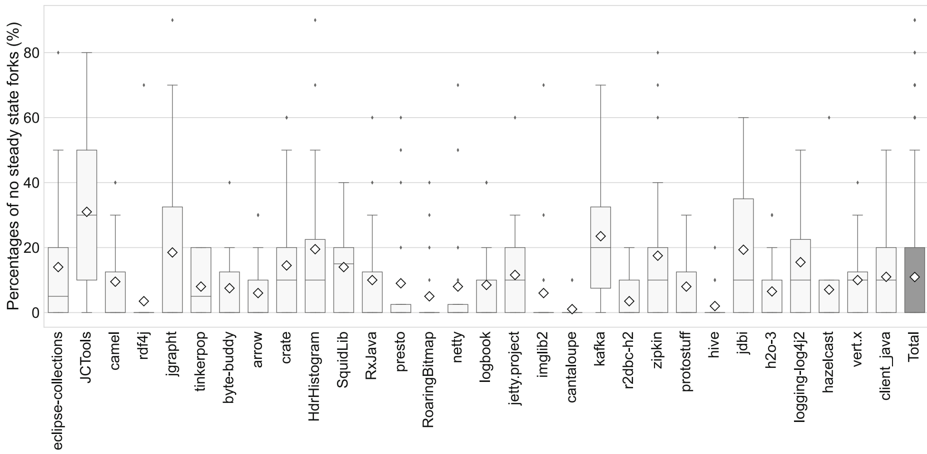


Fig. 7 RQ_1. Percentages of no steady state forks within each benchmark, grouped by subject system

a steady state, all the remaining benchmarks are classified as *inconsistent* (43.5%), which means that their forks showed mixed behavior.

Another viewpoint on the classification of benchmarks is provided in Fig. 7, where we report the distribution of the percentages of no steady state forks in benchmarks, as grouped by systems, which further clarifies what contributes to the percentages of inconsistent benchmarks. We can notice that, in most cases (with very few exceptions like `JCTools`), the systems tend to exhibit inconsistent benchmarks with small percentages of no steady state forks. It is worth recalling that a single fork (i.e., 10% in a benchmark) is enough to flip the classification from *steady state* to *inconsistent*. This is, in fact, the most common case, as we can see in the distribution computed over all the systems (i.e., *Total* in Fig. 7) that shows a mean around 10%.

From a practical perspective, it is also important to estimate how long it takes to reach a steady state, in the cases in which it is reached. This provides a better view on how the time budget could be spent when executing the benchmarks. Figure 8 shows the distributions of

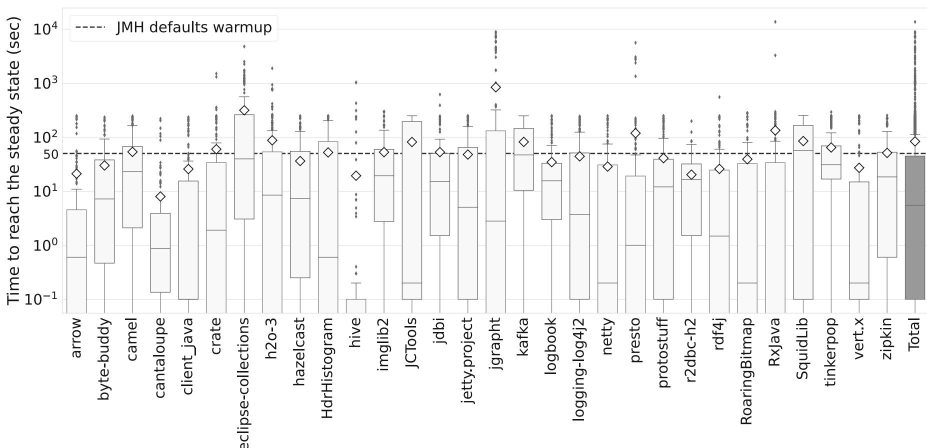


Fig. 8 RQ_1. Time to reach the steady states, in seconds. The y-axis uses a logarithmic scale

the time to reach a steady state, as grouped by system and in total. We can observe that the time spent considerably varies, even within a single system, therefore it can hardly be generalized. This result is not surprising, because the attainment of a steady state inherently depends on the nature of the benchmark. As most of the instability during the warmup is due to the JIT activity, we can imagine that, beside the size of the benchmark method itself, also the number of loaded classes plays a crucial role, since it will induce a different amount of compilation.

Figure 8 also shows how the time spent to reach a steady state compares to the JMH default setting of 50 s for the warmup phase (dashed horizontal line in the figure). We can observe that the difference between the detected end of the warmup phase and the JMH defaults largely differs from one system to the other, by extreme values for *eclipse-collections* and *jgrapht*. The clear picture that emerges from this is that, in most cases, by using the JMH defaults we would overestimate the time needed to warm the benchmark up, therefore wasting a considerable amount of time dedicated to performance testing. While the amount of time that would be wasted considerably varies from one system to another, the percentage of overestimated forks is quite consistent across all the systems. This leads to the conclusion that, more often than not, the JMH defaults for the warmup time should not be used, rather one should rely on techniques to assess the actual amount of time a specific benchmark requires to reach a steady state.

RQ_1 summary - When we only look at individual forks, measurements appear to reach a steady state in the majority of cases. However, when combining forks at the benchmark level, we obtain mixed results. These results provide evidence that benchmarks do not always reach a steady state of performance, thus showing, on a large corpus of JMH benchmarks, that the "*two-phase assumption*" does not always hold. Moreover, in most cases, the JMH defaults for the warmup time tend to overestimate the time needed to reach a steady state.

5.2 RQ_2—Steady State Impact

In this subsection, we present results of our analysis on the impact of steady state on performance.

Figure 9 and Table 2 report results of the analysis that investigates how performance changes (within each fork) when the steady state is reached. In particular, Fig. 9 depicts the distribution of the *relative performance deviation* (RPD) across all the steady forks of our study. The figure highlights strong performance deviations when the steady state is reached, with an average RPD of 123,937% and a median of 41% (IQR 14–195%).

By looking at the detailed results reported in Table 2, we can observe that the large mean is highly influenced by some specific projects (e.g., `camel`, `crate`, `h2o-3` and `presto`), which report extremely high RPD (up to 3.5 billion %). Nonetheless, even when considering projects with smaller RPD (e.g., `jgrapht`, `RoaringBitmap` and `SquidLib`), we can observe considerable performance deviations between steady and non-steady measurements (respectively, 36%, 40% and 57% on average). Besides the diversity across projects, Table 2 also highlights a substantial diversity within each project. Indeed, performance deviations substantially differ across benchmarks of the same projects, as they report extremely high standard deviations (maximum standard deviation of ∼16 billion (`presto`), and minimum of 56 (`RoaringBitmap`)).

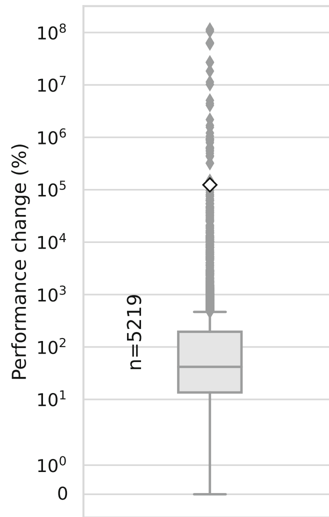


Fig. 9 RQ_2. Steady state impact within forks. The box plot reports the distribution of RPD between steady and non-steady phases of the execution across all the steady forks. The y-axis is in logarithmic scale

The above results suggest that performance substantially changes when forks reach a steady state of performance, and provide empirical evidence on the danger of using non-steady measurements during performance assessment. Indeed, given these large magnitudes, even a tiny portion of non-steady measurements can substantially distort performance indices, with significant implications on performance assessment.

Figure 10 and Table 3 report the results of our second analysis, which investigates performance deviations between steady and non-steady forks. By observing Fig. 10, we can notice that the reported performance deviations are significantly smaller than those within forks (see Fig. 9). The average RPD between steady and non-steady forks is 5%, while the median RPD is 2% (IQR 0-5%). Although these deviations may appear negligible at a first glance, they are still significant if placed in the context of Java microbenchmarking. Indeed, in these contexts, even relatively small performance regressions (e.g., 5%) may lead to rejections of code revisions,[8] as they can have significant impact at system level. Moreover, if we look at some specific projects, such as `byte-buddy`, `JCTools` and `RxJava`, the reported deviations are even more conspicuous (average RPDs of respectively 14%, 19% and 13%, see Table 3). Still, the deviations between steady and non-steady forks are substantially smaller than those within forks. In that, it is worth to remark that we compare steady and non-steady forks using on purpose the same measurement window (i.e., we discard all measurements collected before st, where st denotes the steady starting time of the paired steady fork). Indeed, this methodology may potentially discard measurements that are related to the most unstable phases of the (non-steady) fork execution, and it can consequently smooth deviations from steady measurements. Nonetheless, this fact may also suggest that performance deviations tend to improve over time during benchmark execution, even in forks that do not reach a steady state of performance.

[8] As an example, see `netty` pull request `8614` at https://bit.ly/33MqlMZ.

Table 2 RQ_2. Steady state impact within forks grouped by project. The first and second columns report, respectively, the name of the project and number of steady forks per project. The last three columns report RPD statistics (i.e., mean, standard deviation and median)

System	n	Mean	Std. Dev.	Median
arrow	188	599.6	1,814.9	72.1
byte-buddy	185	276.1	469.1	113.4
camel	181	10,277.3	51,821.1	49.7
cantaloupe	188	2,237.9	3,534.1	506.4
client_java	178	270.8	323.2	274.2
crate	171	20,919.6	125,119.1	26.5
eclipse-collections	172	70.4	185.0	17.4
h2o-3	187	42,129.6	342,798.8	18.9
hazelcast	158	279.1	966.1	30.0
HdrHistogram	161	74.3	143.7	25.9
hive	196	386.4	325.0	344.1
imglib2	188	103.1	187.1	68.2
JCTools	138	69.3	67.0	61.7
jdbi	121	2,305.6	14,856.2	40.2
jetty.project	168	387.0	649.1	106.8
jgrapht	163	35.9	65.0	16.3
kafka	153	196.9	2,063.9	16.0
logbook	183	169.0	304.6	42.4
logging-log4j2	169	181.6	356.6	72.8
netty	184	3,457.2	36,818.1	204.7
presto	182	3,439,550.1	16,220,992.1	27.6
protostuff	184	170.6	237.2	41.5
r2dbc-h2	193	3,460.6	10,268.1	303.7
rdf4j	193	97.7	211.6	35.2
RoaringBitmap	190	39.6	56.4	21.0
RxJava	180	113.9	278.0	35.9
SquidLib	172	57.1	118.8	8.2
tinkerpop	184	1,330.7	7,177.5	24.9
vert.x	144	654.1	951.8	207.0
zipkin	165	28,604.1	152,763.8	21.6
Total	5,219	123,972.4	3,087,067.7	41.8

To further investigate this aspect, we assess how warmup iterations impact performance deviation. In particular, we investigate to what extent warmup iterations mitigate performance deviations (RPD) of non-steady forks. Figure 11 reports the results of our analysis. As it can be seen from the figure, RPDs seem to be considerably influenced by the number of warmup iterations, especially at the early stages of benchmark execution. Indeed, we can see a substantial drop in the first 50 iterations, where the mean RPD decreases from 578% to 17% and the median from 30% to 12%. Moreover, we found a clear trend toward RPD

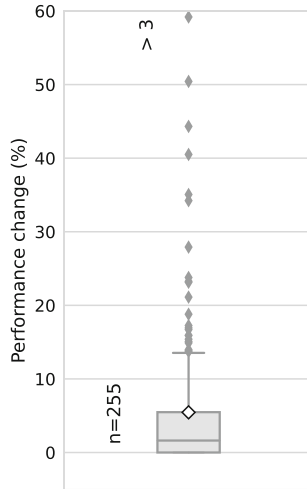


Fig. 10 RQ$_2$. Steady state impact across forks. The box plot reports the distribution of RPDs between steady and non-steady forks across all the inconsistent benchmarks

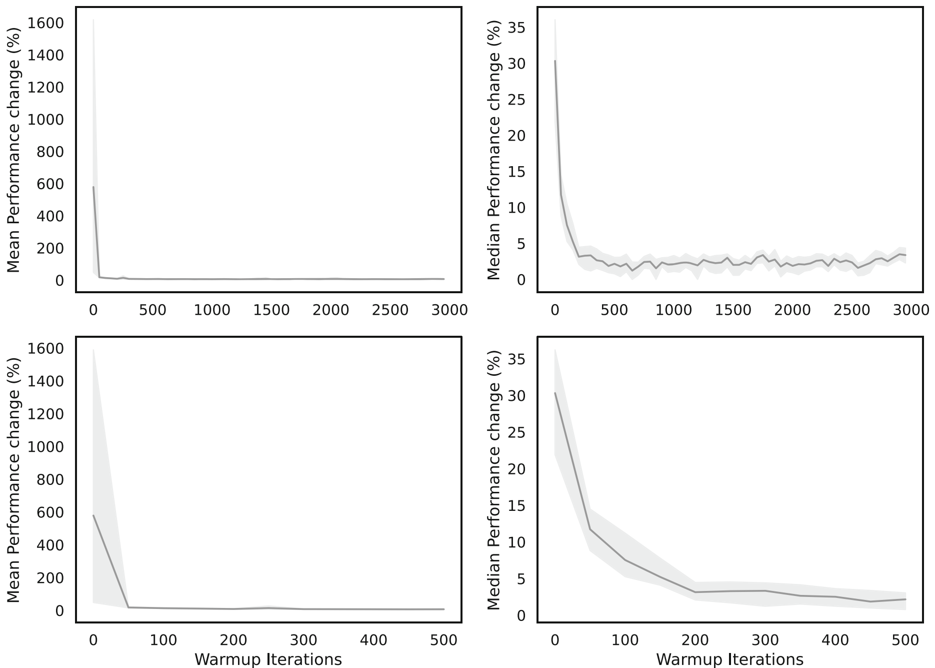


Fig. 11 RQ$_2$. The impact of warmup iterations on RPD. In each plot the x-axis denotes the number of warmup iterations, while the y-axis report RPD statistics. The dark shadow around the line represents the 95% confidence interval. The plots in the first column report the mean RPD, while those in the second column report the median RPD. Plots in the first row present the overall results, while those in the second row zoom on the first 500 iterations

Table 3 RQ$_2$. Steady state impact across forks grouped by project. The first and second columns report, respectively, the name of the project and the number of inconsistent benchmarks per project. The last three columns report RPD statistics for each project (i.e., mean, standard deviation and median)

System	n	Mean	Std. Dev.	Median
arrow	6	8.2	10.2	3.5
byte-buddy	8	14.5	27.0	0.0
camel	9	2.7	3.9	1.2
client_java	11	3.4	4.0	2.8
crate	12	3.3	3.9	2.4
eclipse-collections	10	2.5	4.3	0.2
h2o-3	8	3.7	4.9	1.7
hazelcast	7	6.3	15.1	0.0
HdrHistogram	13	7.3	19.8	0.0
JCTools	16	19.2	17.7	13.3
jdbi	9	7.4	7.1	6.0
jetty.project	10	6.3	5.8	5.4
jgrapht	9	6.6	5.7	4.2
kafka	15	1.8	4.2	0.0
logbook	9	3.2	4.8	0.7
logging-log4j2	12	4.4	3.8	3.9
protostuff	8	0.6	1.2	0.0
r2dbc-h2	6	1.7	1.4	2.1
RxJava	8	13.2	35.7	0.2
SquidLib	14	2.1	2.7	1.4
tinkerpop	10	1.6	1.4	1.6
vert.x	9	1.8	2.9	0.0
zipkin	12	5.5	5.5	3.7
others	24	3.3	5.0	1.0
Total	255	5.5	11.6	1.6

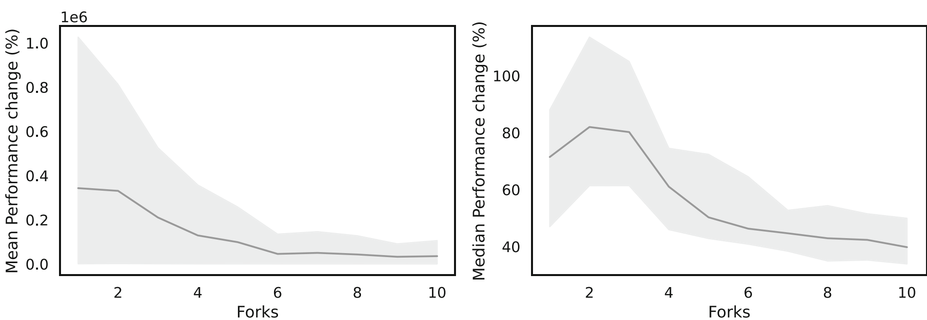


Fig. 12 RQ$_2$. The impact of forks on RPD. In each plot the x-axis denotes the number of non-steady forks considered in the comparison, while the y-axis report RPD statistics. The dark shadow around the line represents the 95% confidence interval. The left plot reports the mean RPD, while the right plot reports the median RPD

reduction in the first 300 iterations. After this point, the decreasing trend seems to disappear, and RPDs begin to show fluctuations with means ranging from a minimum of 6% to a maximum of 9%, and medians ranging from 1% to 3%.

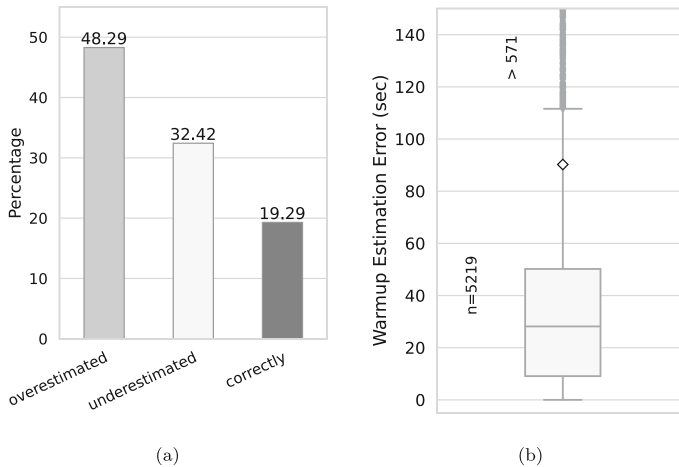
The above results suggest that performance deviations of non-steady forks may be substantially mitigated through a reasonable amount of warmup iterations. In particular, a significant reduction of performance deviations can be obtained (i.e., −561% on average and −18% on median) by using at least 50 warmup iterations, which correspond to 5 s of continuous benchmark execution and no less than 50 invocations. These deviations can be further reduced by using a higher number of warmup iterations up to 300 (i.e., 30 s of continuous execution and at least 300 invocations). After this point, increasing the number of warmup iterations barely affect RPDs.

Besides investigating the impact of warmup iterations, we also analyze the impact of forks. Specifically, we study whether using a higher number of forks reduces performance deviations of non-steady measurements. As it can be observed in Fig. 12, the analysis shows an overall trend toward RPD reduction when increasing the number of forks. The average RPD is reduced by each additional fork, while the median shows a swinging trend only in the first two forks, and then it constantly decreases. If we compare RPDs of individual forks to those of 5 forks, we observe a significant RPD reduction, i.e., −244,237% on the mean and −21% on the median. This trend leads to an RPD reduction of −307,555% (mean) and −31% (median) when using 10 forks. Still, the reported RPDs remain extremely high, i.e., mean of 35,797% and median of 40%. The latter results are not surprising if we consider that our analysis deliberately targets non-steady measurements, which are typically subject to severe performance deviations (as we have shown in the first analysis of this RQ, see Fig. 9). In fact, in this analysis we were not particularly interested on the absolute RPD, rather we wanted to assess how RPDs change with respect to the number of forks. In this regard, our results suggest that increasing the number of forks can effectively mitigate the impact of non-steady measurements.

RQ_2 summary - The attainment of steady state has relevant effects on software performance. Performance substantially changes within forks when their execution reaches steady state. This difference in performance is less pronounced when comparing forks that never reach steady state against those that consistently reach it. Nonetheless, the reported performance deviations are still considerable and potentially harmful for performance assessment. The use of an appropriate number of warmup iterations can significantly mitigate performance deviations induced by non-steady forks. In addition, the use of an adequate number of forks can alleviate deviations that are induced by unstable measurements, which are collected before steady state execution occurs.

5.3 RQ_3—Developer Configuration Assessment

In this subsection, we first present results of the assessment of developer static configurations, thereafter we provide answer to RQ_3.



(a) (b)

Fig. 13 RQ3. Developer configurations—Warmup estimation accuracy. The left plot reports the percentages of overestimated, underestimated, and correctly estimated forks. The right plot depicts the distribution of WEE across all benchmark forks. n is the total amount of data points, and the number on the top of the plot is the amount of outliers not drawn in the figure

5.3.1 Warmup Estimation Accuracy

In the first part of this subsection, we investigate to what extent developers accurately estimate *steady state starting time* (st), i.e., the end of the warmup phase.

Figure 13b depicts the distribution of *warmup estimation error* (WEE) across all benchmark forks. Developers configurations lead to a WEE that ranges between 9 and 50 s in half of the cases (i.e., the interquartile range (IQR)), and they lead to a median and mean WEE of 28 and 90 s, respectively. Interestingly, we found that the estimation error is approximately as large as the steady state starting time (or more) in half of the forks, i.e., the median of the ratios between WEE and st is 0.997 (IQR: 0.79–43.5).

It might be easier for some systems to assess the time required to reach a steady state than for other ones. With respect to the steady state we detected, most systems (27 out of 30) show an estimation error of less than 100 s. However, the developer configurations in `eclipse-collections`, `presto`, and `RxJava` seem to be less effective in estimating the warmup phase. The most evident case is represented by `eclipse-collections` with an error, on average, of around 5 minutes.

Overall, these results suggest that, in most of the cases, *software developers fail to accurately estimate the end of the warmup phase*, and often *with a non-trivial estimation error*.

Besides investigating the estimation accuracy of developers, we also check whether they provide more accurate estimates than JMH defaults[9] (i.e., the default configuration provided by JMH developers). To do so, we compare the WEEs provided by developer configurations against those provided by JMH defaults using the Wilcoxon Rank-Sum test (Cohen 2013), and the Vargha Delaney's effect size measure (Vargha and Delaney 2000). We

[9]In our evaluation, we use the default configuration defined for JMH versions $\geq$ 1.21, i.e., 5 warmup and measurements iterations ($wi = 5$ and $i = 5$), and iteration time of 10 s for both measurement and warmup ($w = 10s$ and $r = 10s$).

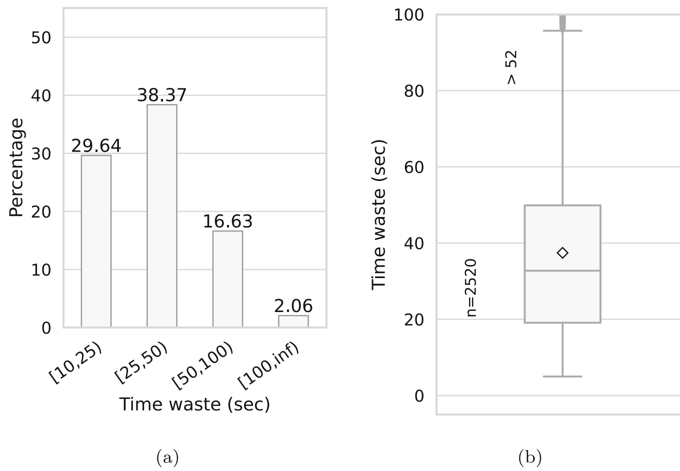


Fig. 14 RQ$_3$. Developer configurations—Overestimation side effects (*Time waste*). The left plot reports the percentages of overestimated forks where *Time waste* $(sec) \in \{[10, 25), [25, 50), [50, 100), [100, inf)\}$. The right plot depicts the distribution of *Time waste* across all overestimated forks. n is the total amount of data points, and the number on the top of the plot is the amount of outliers not drawn in the figure

found that developer configurations outperform JMH defaults with statistical significance ($p < 0.001$) and small effect size ($\hat{A}_{12} = 0.64$). The median and the mean WEE provided by JMH defaults are larger than those provided by developer configurations, i.e., mean of 97 s and median of 50 s (IQR: 36–50 s). These results suggest that the estimates provided by developers are more accurate than those provided by JMH defaults.

Figure 13a reports the percentages of overestimated, underestimated, and correctly estimated forks across all forks (i.e., the estimated warmup time wt is respectively smaller or larger than st by at least 5s). The bar chart shows that overestimation is more common than underestimation. Developers overestimate the end of the warmup phase in 48% of the forks (median WEE: 33 s, IQR: 19–50 s), whereas underestimation is reported in 32% of the cases (median WEE: 150 s, IQR: 36–240 s). Developers accurately estimate it in only 19% of the cases.

In the following subsections, we investigate side effects in both overestimated and underestimated forks.

5.3.2 Overestimation Side Effects

Figure 14 shows the *time waste* due to overestimation. 87% of the overestimated forks waste more than 10 s (i.e., 37% of all the forks). As can be observed by Fig. 14a, overestimation leads to a time waste between 10 and 25 s in 30% of the cases, between 25 and 50 s in 38% of the cases, and it leads to a time waste higher than 50 s in 19% of the cases. Figure 14b depicts the distribution of *time waste* across overestimated forks. The box plot shows that the average *time waste* is 37 s, and the median is 33 s (IQR: 19–49 s).

The amount of time wasted on warmup, after reaching a steady state, considerably varies from one system to another. In most systems, by using the developers configurations, each fork wastes, on average, more than 20 s. More extremes behaviors can be found, for instance, in `jdbi`, `netty`, and `rdf4j`, with an average wasted time of more than 50 s per fork.

On the contrary, `cantaloupe`, `JCTools`, `r2dbc-h2`, and `zipkin` might waste just a few s, and probably their configurations do not need any adjustment.

Such absolute *time wastes*, when contextualized within concrete performance assurance processes (that involve multiple benchmarks), can have substantial effects on the overall execution time and, as consequence, can hamper microbenchmarks adoption for continuous performance assessment. For example, a time waste of 33 s in a relatively small performance testing suite (e.g., `r2dbc-h2`), which involves a typical number of 5 forks,[10] could lead to an overall time waste of approximately one hour.[11] In larger testing suites, such as `RxJava`, the same *time waste* could lead to an overall waste of about 2 days and a half. Besides this, wt is mostly composed by time waste in a large number of cases. In fact, we have measured that the median of the ratios between the time waste and the estimated warmup time wt is approximately 0.97 (IQR: 0.77–0.99), i.e., in half of overestimated forks, at least 97% of the estimated warmup time wt consists of time waste.

The reported results highlight a substantial portion of time wasted during microbenchmarking, and stress the need for better microbenchmark configuration approaches that reduce their execution time. These findings further motivate prior efforts in reducing execution time through dynamic reconfiguration (Laaber et al. 2020), and highlight huge opportunities for execution time reduction in microbenchmarks.

5.3.3 Underestimation Side Effects

Although less frequent than overestimation, underestimation can have relevant side effects on microbenchmarking. Indeed, it can lead to consider performance measurements that significantly differ from steady state performance, as in practice they fall within the warmup phase.

Figure 15 reports the distribution of the relative performance deviation (RPD) across underestimated forks. Underestimation leads to an RPD of at least 5% in 57% of cases (i.e., 22% of all the forks). Figure 15a depicts that underestimation induces an RPD between 5% and 10% in 15% of the cases, between 10% and 25% in 20% of the cases, and it induces an RPD greater than 50% in 9% of the cases. The box plot (Fig. 15b) shows a mean RPD of 17%, and a median of 7% (IQR: 1–21%).

Some systems show a large performance deviation when underestimating the warmup time. In nine systems, developers will observe measurements that deviate at least by 20% from the performance reached in the steady state. Such large deviations prevent the benchmarks to spot smaller performance changes, in fact defeating their purpose in practical performance testing scenarios.

These results highlight significant performance deviations due to underestimation. In Java systems, even relatively small performance regressions (e.g., 5%) may lead to rejections of code revisions. In fact, a microbenchmark regression, for example due to software refactoring (Traini et al. 2021), can have huge impact at system level, as microbenchmarks measure performance at fine-grained level (Laaber and Leitner 2018; Leitner and Bezemer 2017). For this reason, the reported RPDs can have severe consequences on steady state performance assessment, as they can easily lead to faulty judgments of code revisions.
According to our results, developers may rely on measurements that significantly differ

[10]The default number of forks in JMH is 5 (see https://bit.ly/3mOBvHy)

[11]To compute the overall time waste, we multiplied the time waste (33 s) by the number of forks (5) and the number of benchmarks in the testing suite (20 for `r2dbc-h2` and 1,302 for `RxJava`).

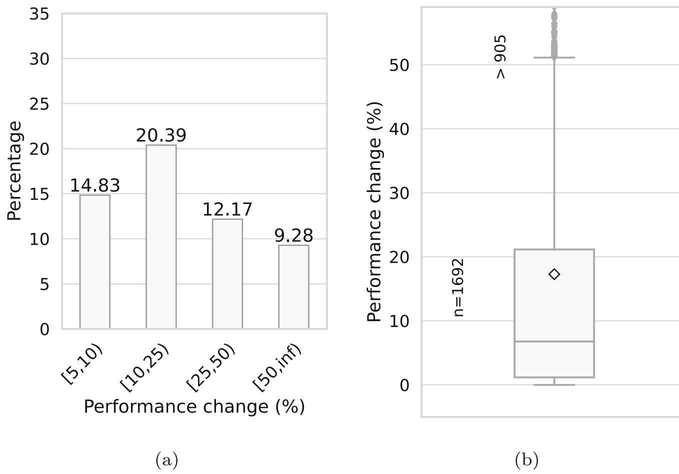


Fig. 15 RQ_3. Developer configurations—Underestimation side effects (*performance change*). The left plot reports the percentages of underestimated forks where *performance change* (%) $\in \{[5, 10), [10, 25), [25, 50), [50, inf)\}$. The right plot depicts the distribution of *performance change* across all underestimated forks. n is the total amount of data points, and the number on the top of the plot is the amount of outliers not drawn in the figure

from those collected during steady state execution. This finding sheds a light on the perils of underestimation, and on how such inaccuracy can disrupt performance assessment. Indeed, the reported results show that underestimation is not rare in the current developer practice (32% of forks), and it often leads to potentially misleading results (57% of underestimated forks lead to a performance deviation ≥5%).

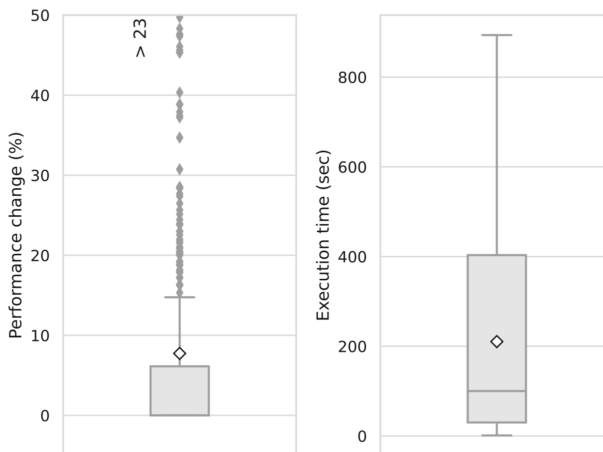


Fig. 16 RQ_3. Developer configurations: Benchmark level assessment. The left plot depicts the RPD distribution across all benchmarks. The right plot depicts the execution time distribution

5.3.4 Benchmark Level Assessment

Here, we discuss the results of the analysis of developer configurations at benchmark level. The left plot of Fig. 16 reports the distribution of performance deviations of measurements gathered by software developers when compared to steady state measurements. The first aspect that we can notice by observing the plot is that the impact of underestimation seems to be considerably mitigated when considering aggregated measurements. This finding is in line with our previous analysis on forks (see RQ_2), in which we have shown that the number of forks can substantially mitigate deviations of non-steady measurements. Performance deviations are not statistically significant in about half of the benchmarks (i.e., 53% of the cases), the median RPD is 0% and the IQR is 0–6%. Nonetheless, 47% of benchmarks report statistically significant performance deviations (i.e., the confidence interval does not contain zero), the mean RPD is 8%, and a quarter of benchmarks report a deviation higher 6%. As we have already discussed in our prior analysis on underestimation, these magnitudes of deviation can be harmful in Java microbenchmarking, as they can mislead performance assessment and lead to wrong judgements of software revisions.

The right plot of Fig. 16 reports the distribution of benchmark execution times based on developer configurations. Developer configurations lead to extremely different execution times, with durations ranging from a minimum of 1 second to a maximum of 893 s. The average execution time is 210 s, and the median is 100 s (IQR 30–403 s). Our previous analysis on overestimation has already highlighted large opportunities for execution time reduction. In the light of the above results, these opportunities appear even more significant. For example, our prior analysis has reported a median time waste of 33 s in overestimated forks. If we compare this result with the median execution time of a benchmark, i.e., 100 s, the time waste appears extremely relevant, i.e., approximately one third of the entire benchmark execution. In that, it is worth to remark that time wastes are measured on individual forks, while the execution times (reported in Fig. 16) measure the entire duration of a benchmark, which typically involves multiple forks (3 on average in the case of developer configurations). This finding further remarks on the need for better techniques to reduce benchmark execution time without affecting result quality.

RQ_3 summary - Developer configurations have limited effectiveness for steady state performance assessment. Developers fail to accurately estimate the end of the warmup phase, often with a non-trivial estimation error. In a large number of cases, this error leads to a substantial increase in the execution time (i.e., overestimation). Nevertheless, underestimation is not rare in the current developer practice, and when this happens it significantly distorts performance assessment.

5.4 RQ_4—Dynamic Reconfiguration Assessment

In this subsection, we first present results of the assessment of dynamic reconfiguration techniques, then we answer to RQ_4.

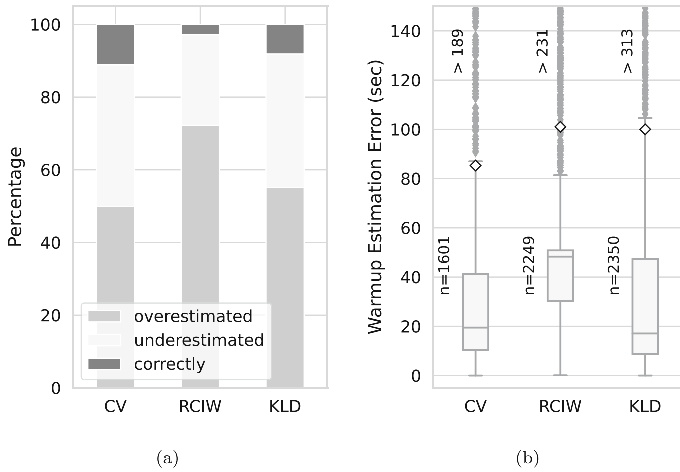


Fig. 17 RQ_4. Dynamic reconfiguration—Warmup estimation accuracy. The left plot reports the percentages of overestimated, underestimated, and correctly estimated forks per dynamic reconfiguration technique. The right plot depicts the distribution of WEE across all benchmark forks per dynamic reconfiguration technique. n is the total amount of data points, and the number on the top of the plot is the amount of outliers not drawn in the figure

5.4.1 Warmup Estimation Accuracy

Figure 17b depicts the distributions of warmup estimation errors (WEE) per dynamic reconfiguration technique across all benchmark forks. As it can be seen from the figure, *RCIW* leads to higher WEE (median: 48 s, IQR: 30–50 s) when compared to other techniques, whereas CV and KLD report similar distributions. *CV* leads to a median WEE of 19 s (IQR: 10–41 s), while *KLD* reports a median of 17 s (IQR: 9–47 s). Also, we have measured that, in half of the forks, all dynamic techniques report a WEE approximately as large as the steady state starting time st (or more), i.e., the medians of the ratios between WEE and st for *CV*, *RCIW* and *KLD* are respectively 0.97 (IQR: 0.79–45.7), 2.04 (IQR: 0.78–75) and 0.97 (IQR: 0.79–36.1). These results indicate that all dynamic reconfiguration techniques lead to a substantial error in the estimate of steady state starting time. Nevertheless, *CV* and *KLD* clearly provide more accurate estimates than *RCIW*.

The bar chart in Fig. 17a confirms the specificity of *RCIW*. As it can be observed from the bar chart, *RCIW* reports a strong tendency toward overestimation, while *CV* and *KLD* show similar frequencies both in terms of underestimation and overestimation. *RCIW* overestimates 72% of forks (median WEE: 46 s, IQR: 31–50 s), and it underestimates only 25% forks (median WEE: 181 s, IQR: 86–236 s). On the other hand, *CV* reports 50% of overestimations (median WEE: 17 s, IQR: 12–24 s) and 39% of underestimations (median WEE: 131 s, IQR: 29–249 s). Similarly, *KLD* reports 55% overestimated forks (median WEE: 13 s, IQR: 9–21 s) and 37% underestimated forks (median WEE: 145 s, IQR: 29–255 s).

5.4.2 Overestimation Side Effects

Besides the large number of overestimated forks, *RCIW* also causes higher *time wastes* when compared to other techniques. As it can be observed in Fig. 18, *RCIW* reports a mean *time waste* of 45 s and a median of 46 s (IQR: 31–55 s). 95% of overestimated forks lead to

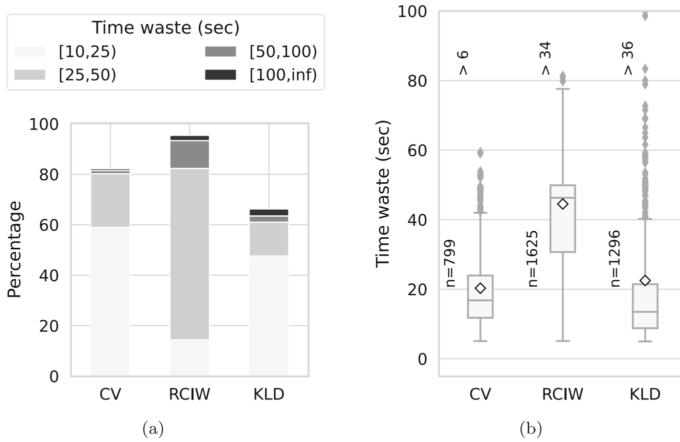


Fig. 18 RQ$_4$. Dynamic reconfiguration—Overestimation side effects (*time waste*). The left plot reports the percentages of overestimated forks where *time waste* $(sec) \in \{[10, 25), [25, 50), [50, 100), [100, inf)\}$. The right plot depicts the distribution of *time waste* across all overestimated forks per dynamic reconfiguration technique. *n* is the total amount of data points, and the number on the top of the plot is the amount of outliers not drawn in the figure)

a *time waste* of at least 10 s, 81% lead to a time waste of at least 25 s, and 13% to a time waste of at least 50 s. On the other hand, *CV* and *KLD* report a median time waste of 17 s (IQR: 12–24 s) and 14 s (IQR: 9–21 s), respectively (see Fig. 18b). Nonetheless, we have measured that in half of the overestimated forks, all techniques lead to an estimated warmup time that mostly consists of time waste, i.e., the medians of the ratios between the *time waste* and *wt* for *CV*, *RCIW*, and *KLD* are respectively 0.98 (IQR: 0.88–0.99), 0.96 (IQR: 0.7–0.99), and 0.97 (IQR: 0.88–0.99). Overall, these results indicate that different techniques

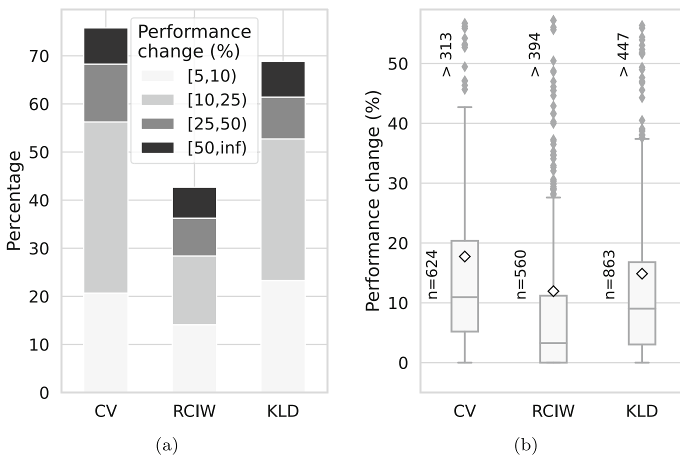


Fig. 19 RQ$_4$. Dynamic reconfiguration—Underestimation side effects (*performance change*). The left plot reports the percentages of underestimated forks where *performance change* $(\%) \in \{[5, 10), [10, 25), [25, 50), [50, inf)\}$. The right plot depicts the distribution of *performance change* across all underestimated forks per dynamic reconfiguration technique. *n* is the total amount of data points, and the number on the top of the plot is the amount of outliers not drawn in the figure)

lead to diverse outcomes in terms of overestimation. For example, RCIW induces more frequent overestimations and higher time wastes when compared to other techniques (see Fig. 18a). Despite this diversity, overestimation is frequent across all dynamic reconfiguration techniques, and it often leads to a non-trivial time waste, which can hamper continuous performance assessment.

5.4.3 Underestimation Side Effects

RCIW is less prone to underestimation than other techniques, and it also has less marked side effects due to underestimation. As it can be observed in Fig. 19, *CV* and *KLD* lead to higher performance changes when compared to *RCIW*. *RCIW* reports a median RPD of 3% (IQR: 0-11%), whereas *CV* and *KLD* report medians of 10% (IQR: 5-20%) and 9% (IQR: 3%-17%), respectively (see Fig. 19b). Additionally, *CV* and *KLD* cause performance changes of at least 5%, respectively, in 76% and 69% of the cases, while *RCIW* achieves a similar deviation in only 43% of forks (see Fig. 19a). These results suggest that underestimation side effect varies depending on the dynamic reconfiguration technique. Some dynamic reconfiguration techniques (i.e., CV and KLD) are more prone to induce underestimation, and they often lead to a non-trivial performance deviation which can potentially mislead steady state performance assessment. Other techniques, such as *RCIW*, are instead "safer" in terms of underestimation, and ensure higher results quality in terms of performance assessment.

5.4.4 Benchmark Level Assessment

The analysis at benchmark level confirms the trend observed in our prior analysis. Figure 20 reports the deviations (RPD) of measurements gathered through dynamic reconfiguration techniques when compared to steady measurements (left plot), along with the distributions of benchmark execution times (right plot). As it can be observed, *RCIW* is by far the most reliable technique in terms of performance deviation. It reports an average RPD of 2%, a

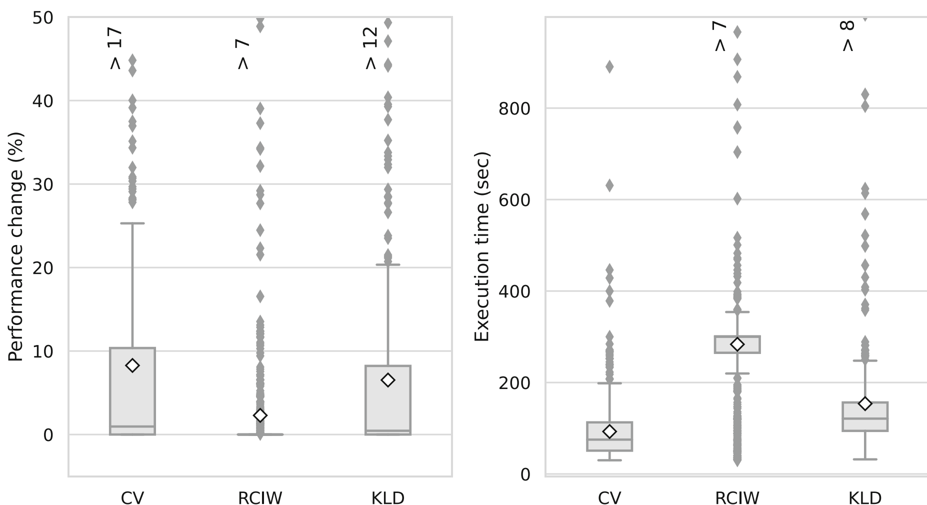


Fig. 20 RQ$_4$. Dynamic reconfiguration: Benchmark level assessment. The left plot depicts the RPD distribution across all benchmarks. The right plot depicts the execution time distribution

median of 0% and an IQR of 0-0%. About 91% of the benchmarks report an RPD smaller than 5%. On the other hand, *RCIW* provides also the most time consuming process with a mean execution time of 283 s and a median of 300 s (IQR 265–300 s). This is not surprising, as our prior analysis has highlighted a large overestimation rate (72%) and relevant time wastes (45 s on average) in *RCIW*.

Contrariwise, *CV* and *KLD* are less reliable in terms of results quality, by reporting a mean RPD of respectively 8% (median: 1%, IQR: 0–10%) and 7% (median: 0.5%, IQR 0–8%), but they are also less demanding in terms of execution time. *CV* reports a mean execution time of 93 s (median: 75 s, IQR: 51–113 s), while *KLD* reports a mean of 154 s (median: 121 s, IQR: 94–156 s). *CV* and *KLD* show similar behaviors in terms of execution times and performance deviations, however, as it emerges by the box plots in Fig. 20, they show two opposite tendencies. In particular, *CV* shows a slight tendency towards faster execution times and less reliable results, while *KLD* shows the opposite behavior, i.e., better results quality and more time-consuming executions.

RQ$_4$ summary - Dynamic reconfiguration techniques provide far from optimal estimates of the warmup phase, often with a non-trivial error. The side effects vary depending on the technique. *RCIW* is more prone to overestimation than other techniques, and it induces more time-consuming benchmark executions (i.e., higher time waste). On the other hand, *CV* and *KLD* often lead to performance measurements that differ from those collected during the steady state, while *RCIW* provides a “safer” assessment of steady state performance.

5.5 RQ$_5$—Dynamic Reconfiguration *vs* Developer Configuration

In this subsection, we first present results of the comparison between dynamic reconfiguration techniques and developer static configurations for each considered metric: (i) *warmup estimation error* (WEE), (ii) *estimated warmup time* (wt), and (iii) *relative performance deviation* (RPD). Then, we provide answer to RQ$_5$.

5.5.1 Warmup Estimation Error

We performed the Wilcoxon test to check the significance of the difference between WEE of dynamic reconfiguration techniques and developer static configurations. The detailed results of the comparison within and across systems are reported in Table 4. As it can be observed by the last row of the table, the differences are statistically significant for all dynamic reconfiguration techniques ($p \leq 0.05$), with two techniques reporting extremely small p-values ($p < 0.001$). The comparison leads to a medium effect size in $RCIW$ ($\hat{A}_{12} \geq 0.64$), and to a large effect size in CV and KLD ($\hat{A}_{12} \geq 0.71$). These results indicate that, when compared to developer static configurations, dynamic reconfigurations techniques overall provide more accurate estimates of the end of the warmup phase.

Nevertheless, Table 4 also shows that the difference between WEE of dynamic reconfiguration techniques and developer configurations varies across systems. Figure 21 shows a summary of these results, where each cell reports the number of projects whose comparison leads to a statically significant difference ($p \leq 0.05$) within a specific $\hat{A}_{12}$ effect size range.

As it can be observed by the figure, CV provides better warmup estimates than developers in 15 of the 30 systems (13 with large effect sizes, and 2 with medium effect sizes).

Table 4 RQ$_5$. WEE comparison. Results of the Wilcoxon test (with $\hat{A}_{12}$ effect sizes in brackets) that compare WEE of dynamic reconfiguration techniques to the ones obtained using developer configurations. Systems where dynamic configurations perform better than developer configurations ($p \leq 0.05$ and $\hat{A}_{12} > 0.5$) are highlighted in bold. Asterisks denote interpretation of the $\hat{A}_{12}$ effect size: small (*), medium (**), large (***)

System	CV	RCIW	KLD
arrow	<**0.001(0.95)***	**0.003(0.86)***	<**0.001(0.92)***
byte-buddy	0.321(−)	<**0.001(0.64)*	**0.01(0.8)***
camel	0.25(−)	0.264(−)	<**0.001(0.86)***
cantaloupe	<0.001(0.46)	<0.001(0.36)*	<0.001(0.36)*
client_java	**0.019(0.88)***	**0.01(0.66)**	0.154(−)
crate	0.405(−)	0.097(−)	0.063(−)
eclipse-collections	**0.007(0.85)***	0.163(−)	0.43(−)
h2o-3	**0.025(0.86)***	0.321(−)	**0.003(0.79)***
hazelcast	**0.014(0.73)***	0.147(−)	0.527(−)
HdrHistogram	**0.023(0.65)**	0.282(−)	0.346(−)
hive	<0.001(0.38)*	<0.001(0.36)*	<0.001(0.33)**
imglib2	0.094(−)	0.279(−)	0.225(−)
JCTools	<**0.001(0.7)**	0.719(−)	<**0.001(0.65)**
jdbi	0.091(−)	0.136(−)	0.052(−)
jetty.project	0.867(−)	0.265(−)	0.357(−)
jgrapht	0.261(−)	0.492(−)	<**0.001(0.77)***
kafka	**0.019(0.82)***	0.655(−)	0.225(−)
logbook	**0.002(0.89)***	<**0.001(0.67)**	**0.001(0.84)***
logging-log4j2	<**0.001(0.89)***	0.304(−)	<**0.001(0.86)***
netty	<**0.001(0.86)***	<**0.001(0.77)***	<**0.001(0.84)***
presto	0.181(−)	<**0.001(0.59)*	<**0.001(0.56)**
protostuff	0.14(−)	0.645(−)	0.294(−)
r2dbc-h2	<**0.001(0.72)***	<**0.001(0.63)*	0.988(−)
rdf4j	0.361(−)	0.076(−)	0.534(−)
RoaringBitmap	<**0.001(0.93)***	0.204(−)	<**0.001(0.88)***
RxJava	**0.034(0.91)***	**0.002(0.87)***	<**0.001(0.86)***
SquidLib	**0.001(0.79)***	**0.009(0.74)***	**0.01(0.77)***
tinkerpop	0.215(−)	0.131(−)	0.803(−)
vert.x	0.7(−)	<**0.001(0.57)*	0.355(−)
zipkin	0.53(−)	0.064(−)	0.372(−)
Total	**0.047(0.79)***	<**0.001(0.68)**	<**0.001(0.74)***

Conversely, developer configurations provide lowers estimation errors than CV in only 2 systems (respectively, negligible and small effect size).

KLD shows a similar trend to that observed for CV. KLD provides better warmup estimates than developers in 13 out of the 30 systems (11 with large effect sizes, 1 with medium effect size, and 1 with negligible effect size), whereas developers outperform KLD in only 2 systems (respectively, small and medium effect size).

	L	M	S	N	N	S	M	L
CV	0	0	1	1	0	0	2	13
RCIW	0	0	2	0	0	4	2	4
KLD	0	1	1	0	1	0	1	11

0.29 0.34 0.44 0.5 0.56 0.64 0.71

Vargha-Delanay's $\hat{A}_{12}$

Fig. 21 RQ5. *WEE* comparison summary. Each cell reports the number of systems whose comparison leads to a statistically significant change ($p \leq 0.05$) within a specific $\hat{A}_{12}$ effect size range: negligible (N), small (S), medium (M) and large (L). $\hat{A}_{12} > 0.5$ indicates that dynamic configurations perform better than developer configurations

RCIW also shows improvement over static configurations in a considerable number of systems (10 out of 30), though with lower effect sizes (4 large, 2 medium, and 4 small). Again, developer configurations provide better estimations in only 2 systems.[12]

Overall, we can observe that dynamic reconfiguration techniques provide more accurate warmup estimates than software developer ones. In particular, *CV* and *KLD* outperform developer configurations in terms of *WEE* on a considerable number of projects with high effect sizes.

5.5.2 Estimated Warmup Time

In this subsection, we investigate the difference between the estimated warmup time (wt) provided by dynamic reconfiguration techniques and developer static configurations. Table 5 reports results of Wilcoxon tests for each system, and across all systems.

As it can be observed by the last row of the table, *CV* and *KLD* report a statistically significant difference (p < 0.001) with tendency toward improvement (i.e., smaller wt values) but with negligible effect size. *RCIW* also reports statistically significant difference (p $<$ 0.001), but with the opposite tendency, i.e., larger wt ($\hat{A}_{12} < 0.5$), and a small effect size.

If we look at project-level results, we can observe a remarkable diversity among projects for *CV* and *KLD*. As it can be seen in Fig. 22, *CV* leads to higher wt than those defined by developers ($\hat{A}_{12} < 0.5$) in 10 of the 30 systems, and it reports lower wt ($\hat{A}_{12} > 0.5$) in 17 systems, thus not showing a clear trend. A similar behavior can be observed for *KLD*: 10 of the 28 systems that report a statically significant difference (p ≤ 0.05) have $\hat{A}_{12} < 0.5$, while 17 of them have $\hat{A}_{12} > 0.5$.

On the other hand, *RCIW* performs worse than developer configurations in most of the systems. *RCIW* leads to higher wt than developers in 22 of the 30 systems: 12 with large effect sizes, 4 with medium effect sizes, and 4 with small effect sizes, 2 with negligible effect sizes. It leads to shorter wt in only 5 projects (1 negligible, 3 small, and 1 medium effect sizes).

Furthermore, if we compare overestimation frequency/side effect of *RCIW* (see Figs. 17a and 18) and developer configurations (see Figs. 13 and 14), we can observe that

[12] Interestingly, `cantaloupe` and `hive` are the same two systems where developers provide better warmup estimations than whatever dynamic reconfiguration technique.

Table 5 RQ$_5$. Estimated warmup time (wt) comparison. Results of the Wilcoxon test (with $\hat{A}_{12}$ effect sizes in brackets) that compare the wt of dynamic reconfiguration techniques to the ones obtained using developers configurations. Systems where dynamic configurations report shorter wt than developer configurations ($p \leq 0.05$ and $\hat{A}_{12} > 0.5$) are highlighted in bold. Asterisks denote interpretation of the $\hat{A}_{12}$ effect size: small (*), medium (**), large (***)

System	CV	RCIW	KLD
arrow	<**0.001(0.87)*****	<**0.001(0.63)***	<**0.001(0.87)*****
byte-buddy	**0.002(0.56)***	<0.001(0.21)***	<**0.001(0.59)***
camel	<**0.001(0.75)*****	0.002(0.49)	<**0.001(0.77)*****
cantaloupe	<0.001(0.22)***	<0.001(0.05)***	<0.001(0.06)***
client_java	<**0.001(0.66)****	<0.001(0.21)***	<**0.001(0.66)****
crate	<**0.001(0.62)***	<0.001(0.41)*	<**0.001(0.65)****
eclipse-collections	<**0.001(0.59)***	<0.001(0.42)*	<**0.001(0.58)***
h2o-3	<**0.001(0.62)***	<0.001(0.44)*	**0.005(0.57)***
hazelcast	<0.001(0.39)*	<0.001(0.32)**	0.078(−)
HdrHistogram	<0.001(0.34)*	<0.001(0.25)***	<0.001(0.33)**
hive	<0.001(0.06)***	<0.001(0.06)***	<0.001(0.06)***
imglib2	<0.001(0.4)*	<0.001(0.34)**	<0.001(0.39)*
JCTools	<0.001(0.36)*	<0.001(0.27)***	0.002(0.42)*
jdbi	<**0.001(0.65)****	0.004(0.47)	**0.002(0.63)***
jetty.project	<0.001(0.37)*	<0.001(0.28)***	<0.001(0.42)*
jgrapht	<**0.001(0.67)****	<0.001(0.37)*	0.363(−)
kafka	<**0.001(0.65)****	0.18(−)	<**0.001(0.64)***
logbook	<**0.001(0.76)*****	<0.001(0.33)**	<**0.001(0.75)*****
logging-log4j2	<**0.001(0.75)*****	0.525(−)	<**0.001(0.75)*****
netty	<**0.001(0.69)****	<**0.001(0.58)***	<**0.001(0.74)*****
presto	0.075(−)	<0.001(0.15)***	<0.001(0.3)**
protostuff	<0.001(0.34)**	<0.001(0.23)***	<0.001(0.35)*
r2dbc-h2	<0.001(0.2)***	<0.001(0.05)***	<0.001(0.35)*
rdf4j	<**0.001(0.7)****	0.383(−)	0.25(−)
RoaringBitmap	<**0.001(0.83)*****	<**0.001(0.71)****	<**0.001(0.8)*****
RxJava	<**0.001(0.72)*****	**0.021(0.6)***	<**0.001(0.72)*****
SquidLib	<**0.001(0.67)****	<**0.001(0.53)**	<**0.001(0.64)****
tinkerpop	0.33(−)	<0.001(0.32)***	**0.008(0.52)**
vert.x	0.958(−)	<0.001(0.13)***	**0.03(0.62)***
zipkin	<0.001(0.31)**	<0.001(0.2)***	<0.001(0.41)*
Total	<**0.001(0.55)**	<0.001(0.36)*	<**0.001(0.55)**

$RCIW$ reports more frequent overestimations, namely 72% *vs* 48%, and higher *time wastes*, namely median of 46 s (IQR: 31–55 s) *vs* median of 33 s (IQR: 19–49 s).

Overall, our results indicate that CV and KLD can lead to different behaviors based on the context, that is they can provide either higher or lower wt than developers depending on the system. On the other hand, $RCIW$ induces higher estimates of the warmup time wt when compared to developer configurations, thus increasing microbenchmark execution time.

	L	M	S	N	N	S	M	L
CV	3	2	5	0	0	4	7	6
RCIW	12	4	4	2	1	3	1	0
KLD	2	2	6	0	1	6	3	7

0.29 0.34 0.44 0.5 0.56 0.64 0.71

Vargha-Delanay's $\hat{A}_{12}$

Fig. 22 RQ_5. Summary wt comparison. Each cell reports the number of systems whose comparison leads to a statistically significant change ($p \leq 0.05$) within a specific $\hat{A}_{12}$ effect size range: negligible (N), small (S), medium (M) and large (L). $\hat{A}_{12} > 0.5$ indicates that dynamic configurations lead to shorter wt than developer configurations

5.5.3 Relative Performance Deviation

In this subsection, we assess the difference between RPD of dynamic reconfiguration techniques and developer static configurations. Overall, we can observe that dynamic reconfiguration techniques slightly improve developer configurations, as shown in the last row of Table 6. The comparisons report statistically significant differences for all techniques ($p < 0.001$), respectively with small (CV and $RCIW$) and negligible (KLD) effect sizes.

If we look at project-level results (see Table 6 and Fig. 23), we can observe that $RCIW$ leads to statistically significant improvements over developer configurations in a large number of projects. In particular, $RCIW$ improves developer configurations in 18 of the 30 systems (5 with large, 3 with medium, 5 with small, and 5 with negligible effect sizes), while it degrades RPD in only 2 systems (both with negligible effect sizes). By comparing the RPD distributions of $RCIW$ (Fig. 19) and developer configurations (Fig. 15), we can observe that the former produces lower deviations with respect to steady state measurements. For example, using developer configurations, about 57% of the forks lead to an RPD of at least 5%, while the same performance deviation is achieved in 43% of forks when using $RCIW$. Even more, $RCIW$ provides a median RPD of 3% (IQR: 0–11%), whereas developer configurations lead to a deviation of 7% (IQR: 1–21%).

These results demonstrate that performance measurements gathered through $RCIW$ deviate less from steady state measurements than those collected through developer static configurations, thereby ensuring better results quality.

CV and KLD also report statistically significant improvements over static configurations in 16 and 17 systems, respectively. Nonetheless, developer configurations perform better than CV and KLD in, respectively, 10 and 12 systems. Despite this, by looking at Fig. 23, we can observe that systems where developer configurations perform better than CV and KLD, tend to have lower effect sizes than those where they provide improvement. For example, if we exclude negligible effect sizes, we can observe that CV still leads to improvement ($\hat{A}_{12} \geq 0.56$) in 7 systems (5 large and 2 medium), while it performs worse than developer configurations ($\hat{A}_{12} \leq 0.44$) in only one case with small effect size. Similarly, KLD leads to an effect size $\hat{A}_{12} \geq 0.56$ in 10 systems (5 large, 2 medium, 3 small), while it reports a small effect size $\hat{A}_{12} \leq 0.44$ in 5 systems.

These results suggest that the comparison between RPD of CV/KLD and developer configurations lead to different outcomes depending on the system, though with a slight overall tendency towards improvement. The results across benchmarks of all systems

Table 6 RQ$_5$. RPD fork level comparison. Results of the Wilcoxon test (with $\hat{A}_{12}$ effect sizes in brackets) that compare the RPD of dynamic reconfiguration techniques to the ones obtained using developer configurations. Projects where dynamic configurations perform better than developer configurations ($p \leq 0.05$ and $\hat{A}_{12} > 0.5$) are highlighted in bold. Asterisks denote interpretation of the $\hat{A}_{12}$ effect size: small (*), medium (**), large (***)

System	CV	RCIW	KLD
arrow	<0.001(0.48)	0.145(−)	<0.001(0.46)
byte-buddy	<**0.001(0.53)**	**0.005(0.6)***	<**0.001(0.53)**
camel	<0.001(0.44)	<0.001(0.5)	<0.001(0.4)*
cantaloupe	<**0.001(0.66)****	<**0.001(0.67)****	<**0.001(0.67)****
client_java	**0.001(0.54)**	**0.048(0.55)**	<**0.001(0.53)**
crate	<0.001(0.48)	0.346(−)	<0.001(0.46)
eclipse-collections	<**0.001(0.53)**	0.295(−)	<**0.001(0.5)**
h2o-3	<0.001(0.5)	**0.019(0.56)**	<0.001(0.5)
hazelcast	<**0.001(0.76)*****	<**0.001(0.77)*****	**0.008(0.76)*****
HdrHistogram	0.296(−)	**0.007(0.61)***	**0.012(0.6)***
hive	<**0.001(0.52)**	<**0.001(0.52)**	**0.002(0.52)**
imglib2	<**0.001(0.8)*****	<**0.001(0.78)*****	<**0.001(0.76)*****
JCTools	0.739(−)	**0.031(0.59)***	**0.042(0.57)***
jdbi	<**0.001(0.54)**	0.282(−)	<**0.001(0.54)**
jetty.project	**0.006(0.69)****	<**0.001(0.69)****	**0.019(0.66)****
jgrapht	**0.002(0.52)**	0.114(−)	0.179(−)
kafka	<0.001(0.44)	0.076(−)	<0.001(0.41)*
logbook	<0.001(0.44)	**0.004(0.51)**	<0.001(0.39)*
logging-log4j2	0.079(−)	0.334(−)	**0.007(0.51)**
netty	<0.001(0.48)	0.059(−)	<0.001(0.44)*
presto	0.753(−)	<**0.001(0.66)****	**0.021(0.63)***
protostuff	<**0.001(0.79)*****	<**0.001(0.82)*****	<**0.001(0.78)*****
r2dbc-h2	<**0.001(0.84)*****	<**0.001(0.87)*****	<**0.001(0.82)*****
rdf4j	<0.001(0.45)	0.827(−)	0.007(0.47)
RoaringBitmap	<**0.001(0.51)**	<**0.001(0.53)**	<0.001(0.48)
RxJava	0.003(0.49)	0.35(−)	0.003(0.5)
SquidLib	<0.001(0.41)*	0.003(0.45)	<0.001(0.42)*
tinkerpop	<**0.001(0.51)**	**0.044(0.6)***	<0.001(0.47)
vert.x	**0.039(0.54)**	**0.001(0.57)***	<**0.001(0.5)**
zipkin	<**0.001(0.78)*****	<**0.001(0.8)*****	<**0.001(0.75)*****
Total	<**0.001(0.57)***	<**0.001(0.6)***	<**0.001(0.55)**

confirm this tendency. As shown in the last row of Table 6, CV and KLD report statistically significant improvements (p $<$ 0.001), respectively with small and negligible effect sizes. Given these results, we can safely state that CV and KLD only provide marginal improvements over developer static configurations in terms of performance deviation.

	L	M	S	N	N	S	M	L
CV	0	0	1	9	9	0	2	5
RCIW	0	0	0	2	5	5	3	5
KLD	0	0	5	7	7	3	2	5

0.29 0.34 0.44 0.5 0.56 0.64 0.71

Vargha-Delanay's $\hat{A}_{12}$

Fig. 23 RQ$_5$. Summary *RPD* fork level comparison. Each cell reports the number of systems whose comparison leads to a statistically significant change ($p \leq 0.05$) within a specific $\hat{A}_{12}$ effect size range: negligible (N), small (S), medium (M) and large (L). $\hat{A}_{12} > 0.5$ indicates that dynamic configurations perform better than developer configurations

5.5.4 Benchmark Level Assessment

Interestingly, when we look at benchmark level results, we can observe significant differences (see Fig. 24 and Table 7). *CV* notably shifts from a tendency towards improvement to a tendency towards regression ($p < 0.001$ and $\hat{A}_{12} = 0.45$). *KLD*, which reported statistically significant differences and tendency towards improvement at fork level, reports neither improvement nor regression ($p > 0.05$) at benchmark level. By analyzing project level results, we can further appreciate this shift. At fork level, *CV* reports worse RPDs than developers in only one project (with non-negligible effect size). At benchmark level instead, if we exclude negligible effect sizes, it reports worse performance deviations than developer configurations in 11 projects (3 with small effect sizes, 2 with medium, and 6 with large). Likewise, *KLD* reports worse RPDs in 9 projects at benchmark level (2 small, 2 medium and 5 large effect sizes), while, at fork level, it reports worse RPDs in only 5 project with small effect sizes. The only technique that seems to provide improvement over developer configurations both at benchmark and fork level is *RCIW*, which reports statistically significant improvement ($p < 0.001$) and small effect size ($\hat{A}_{12} = 0.64$). *RCIW* outperforms developer configurations in 17 projects (10 with large effect sizes, 5 with medium and 2 with small), and the effect sizes provided at benchmark level are even better than those provided at fork level (i.e., higher $\hat{A}_{12}$) in 15 out of the 30 projects. These results may suggest that the capability of *RCIW* to dynamically stop forks at run-time may further improve performance deviations when compared to those of developers. To further investigate this aspect, we analyzed the results of these 15 projects, and we found that, in 80% of the benchmarks,

	L	M	S	N	N	S	M	L
CV	6	2	3	1	0	1	0	5
RCIW	0	0	1	0	0	2	5	10
KLD	5	2	2	0	0	3	2	2

0.29 0.34 0.44 0.5 0.56 0.64 0.71

Vargha-Delanay's $\hat{A}_{12}$

Fig. 24 RQ$_5$. Summary *RPD* benchmark level comparison. Each cell reports the number of systems whose comparison leads to a statistically significant change ($p \leq 0.05$) within a specific $\hat{A}_{12}$ effect size range: negligible (N), small (S), medium (M) and large (L). $\hat{A}_{12} > 0.5$ indicates that dynamic configurations perform better than developer configurations

Table 7 RQ$_5$. RPD benchmark level comparison. Results of the Wilcoxon test (with $\hat{A}_{12}$ effect sizes in brackets) that compare RPD of dynamic reconfiguration techniques to the ones obtained using developer configurations. Systems where dynamic configurations perform better than developer configurations ($p \leq 0.05$ and $\hat{A}_{12} > 0.5$) are highlighted in bold. Asterisks denote interpretation of the $\hat{A}_{12}$ effect size: small (*), medium (**), large (***)

System	CV	RCIW	KLD
arrow	0.015(0.29)**	0.285(−)	0.046(0.37)*
byte-buddy	0.084(−)	**0.015(0.73)*****	0.196(−)
camel	0.001(0.19)***	0.18(−)	<0.001(0.11)***
cantaloupe	**0.001(0.79)*****	**0.001(0.83)*****	**0.002(0.81)*****
client_java	0.008(0.4)*	**0.043(0.57)***	<0.001(0.29)**
crate	0.131(−)	0.735(−)	0.114(−)
eclipse-collections	0.002(0.2)***	0.347(−)	0.005(0.29)***
h2o-3	0.002(0.25)***	0.069(−)	0.05(0.39)*
hazelcast	**0.004(0.76)*****	**0.004(0.8)*****	**0.009(0.71)****
HdrHistogram	0.776(−)	0.074(−)	**0.028(0.61)***
hive	**0.03(0.72)*****	**0.031(0.73)*****	0.056(−)
imglib2	**0.035(0.59)***	0.068(−)	**0.015(0.59)***
JCTools	0.144(−)	0.655(−)	0.317(−)
jdbi	0.055(−)	**0.025(0.71)*****	0.087(−)
jetty.project	0.064(−)	**0.003(0.76)*****	0.099(−)
jgrapht	0.756(−)	0.311(−)	0.51(−)
kafka	<0.001(0.15)***	**0.018(0.68)****	0.002(0.24)***
logbook	0.046(0.36)*	**0.037(0.66)****	0.019(0.26)***
logging-log4j2	0.463(−)	0.18(−)	0.31(−)
netty	0.091(−)	**0.028(0.65)****	0.055(−)
presto	0.861(−)	**0.026(0.67)****	**0.041(0.61)***
protostuff	<**0.001(0.73)*****	<**0.001(0.88)*****	**0.001(0.74)*****
r2dbc-h2	**0.006(0.76)*****	<**0.001(0.88)*****	**0.013(0.68)****
rdf4j	0.015(0.38)*	0.48(−)	0.551(−)
RoaringBitmap	0.001(0.24)***	0.009(0.4)*	0.028(0.34)**
RxJava	0.463(−)	**0.005(0.71)****	0.959(−)
SquidLib	0.011(0.31)**	0.861(−)	0.079(−)
tinkerpop	<0.001(0.12)***	0.155(−)	<0.001(0.14)***
vert.x	0.114(−)	**0.005(0.79)*****	0.286(−)
zipkin	0.148(−)	**0.005(0.73)*****	0.14(−)
Total	<0.001(0.45)	<**0.001(0.64)***	0.186(−)

RCIW involves 5 fork executions, i.e., the maximum number of forks for dynamic reconfiguration techniques (based on the original parameterization provided in Laaber et al. (2020)). That is, forks are not halted by stability criteria, rather they are stopped because the technique has reached the maximum number of allowed forks. This is somehow equivalent to statically fix the number of forks to 5.

Our analysis at fork level has shown that dynamic reconfiguration outperforms developer configurations in terms of performance deviation due to its capability to dynamically stop

warmup iterations. The same cannot be said when stability criteria are applied to halt forks. Indeed, the analysis at benchmark level has shown that this capability of dynamic reconfiguration has only neutral or negative effects on performance deviations, and it is hard to perceive any improvement brought by this specific feature.

RQ_5 summary - When compared to developer configurations, dynamic reconfiguration techniques provide more accurate estimates of the warmup time and better results quality (in most of the cases). CV and KLD show the largest improvement in terms of estimation accuracy, but only provide slight improvement in terms of results quality. On the other hand, $RCIW$ outperforms developer configurations both in terms of estimation accuracy and results quality, but this improvement often comes at the expense of an increased warmup time. When time doesn't not represent a key concern, $RCIW$ should be the primary choice.

6 Discussion

Overall, we can observe that Java microbenchmarking is still subject to some flaws. The results for RQ_1 provide evidence that benchmarks do not always reach a steady state of performance. About 11% of benchmark forks never reach a steady state of performance, and 43% of benchmark executions involve at least one fork that doesn't hit the steady state. These results are consistent with the seminal study on VM microbenchmarking of Barrett et al. (2017), thus showing, on a larger corpus of benchmarks and in the more defined scope of "testing-oriented" Java microbenchmarks, that the "*two-phase assumption*" does not always hold. With this finding, we aim to raise awareness among developers (and researchers) that deal with Java microbenchmarking. An important lesson here is that some benchmark forks (mean ∼10%) may not be representative of "actual" steady state performance, since their performance may continuously fluctuate over time, with a non-negligible deviation from steady state performance. Unfortunately, the only way to avoid this issue is to execute each fork for a large number of iterations, and then run the Barrett et al. (2017) technique to determine if the steady state of performance is reached or not. While this methodology may be appropriate in a research context (like ours), it may be impractical in real-word performance assurance processes, where benchmarks are repeatedly executed against software evolution, and time/resources are subject to constraints (Traini 2022). Nonetheless, there are certain measures that can be put in place to (partially) mitigate this problem. For example, the analyses performed for RQ_2 showed that performance deviations of non-steady forks can be significantly reduced by using a minimum of 50 warmup iterations that, based on our microbenchmarking setup, correspond to 5 s of continuous benchmark execution and no less than 50 invocations. The performance deviations can be further mitigated by increasing the number of warmup iterations up to 300 (i.e., 30 s of continuous benchmark execution and no less than 300 invocations). It is worth to notice that these values are considerably different than those provided by JMH defaults, which define 50 s of continuous benchmark execution and no less than 5 invocations for warmup. On the basis of our results, our practical suggestion is to never execute a benchmark for less than 5 s (and less than 50 invocations) before starting to collect measurements. When time does not represent a major concern, warmup should last for at least 30 s of continuous benchmark execution, and no less than 300 invocations.

Microbenchmarking is far from trivial even when benchmarks consistently reach a steady state of performance. The results of RQ_2 sheds a light on the potential pitfalls of using non-steady measurements. Performance measurements gathered in non-steady phases of benchmark execution substantially deviate from those collected during steady phases ($\sim$124k% on average). Hence, relying on them can significantly mislead performance assessment. To deal with this problem, the current practice mostly rely on developers' *guesses* to estimate the end of the warmup phase and discard measurements subject to performance fluctuations. Based on the results for RQ_3, this approach seems to drastically mitigate these large deviations, i.e., developer configurations lead to an average deviation from steady measurements of 8%. Nonetheless, warmup estimation remains challenging and subject to (large) errors. The results for RQ_3 show that developer static configurations fail to accurately estimate the end of the warmup phase, often with a non-trivial estimation error (median: 28 s). Developers tend to overestimate warmup time more frequently than underestimating it (48% *vs* 32%). Nonetheless, both of these kinds of estimation errors produce relevant (though diverse) side effects. For example, we showed that overestimation produces severe time wastes (median: 33 s), thereby hampering the adoption of benchmarks for continuous performance assessment. On the other hand, underestimation often leads to performance measurements that significantly deviate from those collected in the steady state (median 7%), thus leading to poor results quality and potentially wrong judgements. The latter side effect can be partially mitigated by running an adequate number of forks (e.g., 5). Indeed, as we have shown in RQ_2, forks play a significant role in reducing performance deviations of non-steady measurements. Unfortunately, they also largely increase benchmark execution time, and this may be impractical in real-word contexts. Another option is to leverage automated techniques that can effectively estimate the end of the warmup time at run-time. Prior work tried to address this challenge through dynamic reconfiguration (Laaber et al. 2020).

Based on the results for RQ_5, dynamic reconfiguration techniques significantly improve the effectiveness of the state-of-practice. The achieved results show that dynamic reconfiguration techniques outperform developer static configurations in terms of *warmup estimation error* with statical significance ($p \leq 0.05$) and large/medium effect sizes (see Section 5.5.1). Nevertheless, this improvement may come at the expense of an increased microbenchmark execution time. For example, $RCIW$ produces higher estimates of the warmup time with non-negligible effect size in 20 out of the 30 systems (see Section 5.5.2). On the other hand, CV and KLD have more heterogeneous behaviors depending on the system, but they still report higher warmup estimates than those of developers in 10 systems. Further empirical studies are needed to assess whether such time increase is acceptable for practitioners.

The results for RQ_4 also highlight a substantial diversity among different dynamic reconfiguration techniques. One peculiar example is $RCIW$ that, on one hand, induces the highest increase in microbenchmark execution time, but on the other hand, it provides the most reliable set of performance measurements. Microbenchmark practitioners that do not have specific concerns on time (e.g., small benchmarks suites) should adopt $RCIW$ for a reliable steady state performance assessment. In the other cases, KLD and CV represent the best alternatives.

Despite the promising results highlighted in RQ_5, our findings suggest room for improvement for dynamic reconfiguration. As shown for RQ_4, all dynamic reconfiguration techniques lead to a substantial estimation error, with $RCIW$ providing by far the largest error (median: 48 s), and, CV and KLD producing smaller, but still relevant, estimation errors (median of 19 and 17 s, respectively). These errors induce significant, though diverse, side effects depending on the technique. For example, $RCIW$ induces substantial side

effects in terms of time waste (median: 46 s), while KLD and CV induce more frequent and impactful side effects on the reliability of performance measurements (median performance deviation of 10% and 9%, respectively). Nonetheless, half of the warmup estimates of dynamic reconfiguration techniques can be reduced by at least 96%, when only considering overestimated forks. These results highlight a large space for improvement in dynamic reconfiguration techniques, and call for further research on designing and developing more effective dynamic reconfiguration techniques. For example, future research may explore the use of other stability metrics (e.g., autocorrelation metrics, other confidence interval metrics (Fieller 1954)), or combinations of them to more effectively determine the end of the warmup phase. Another suggestion is to focus more on improving warmup estimation accuracy, rather than finding stability criteria that are suitable to both stop forks and warmup iterations. Indeed, based on our results, dynamically stopping forks does not produce any tangible improvement over developer static configurations. In this regard, our suggestion is to allow practitioners manually configuring forks based on their own needs and time constraints. Nonetheless, we always recommend to run at least 5 forks (i.e., the default in JMH) to mitigate the impact of non-steady measurements. Practitioners may decide to run less forks when time represents a major concern, however they should be aware of potential implications on results quality.

7 Threats to Validity

Our study may be affected by different threats that span from how we collected performance data to the subject project domains, and we describe them in the following.

Construct Validity We assume that the benchmarks that reach a steady state will be able to do it within the execution time we defined. There may be benchmarks that need more time to show some stability. However, we chose the execution time to be considerably longer than the time we found in developer configurations (171 times longer on average). Also, as done in other studies (Laaber et al. 2020), measurement time, the number of iterations, and the number of forks are fixed for every benchmark. The consequence is that the number of invocations varies from one benchmark to another. Nonetheless, no benchmark is invoked less than 3000 times per fork, that is 1000 times more than in Barrett et al. (2017). The number of forks is fixed at 10, as recommended in Barrett et al. (2017).

Our experiment was performed in an environment where we tried our best to reduce the measurement noise and external influencing factors (Papadopoulos et al. 2021; Mytkowicz et al. 2009b). Such settings are effective in improving the accuracy of results, but may not represent the more common environment in which developers execute the benchmarks. However, general reference environments can hardly exist for benchmarks. In fact, different developers can potentially execute the same microbenchmark on a wide range of different machines/environments, given the inherently distributed nature of open-source software development. On top of that, there is an increasing interest in promoting the adoption of microbenchmarks in CI (Laaber and Leitner 2018; Laaber et al. 2019, 2020, 2021), which are most likely uncontrolled/noisy environments. All these aspects make it impractical to identify a reference environment for each benchmark/system. For this reason, we have deliberately chosen to execute benchmarks on the same bare-metal server, using precautionary measures to mitigate measurement noise and external influencing factors (see Section 4.1 for details). In this respect, we preferred to control the confounding variables rather than losing accuracy in more noisy and unreproducible settings. This consideration was especially

motivated by the first goal of our study (RQ_1), i.e., checking whether benchmarks reach a steady state of performance. Indeed, we prefer to run benchmarks on an environment that is considerably less subject to noise than those of developers, rather than possibly causing non-steady executions due to the noise that is specific to our environment. Nonetheless, we do believe that future research should further strengthen our investigation beyond controlled environments, i.e., by studying steady state performance even in uncontrolled and noisy environments. For example, future studies may replicate our experiments in cloud contexts to assess how these virtualized environments affect steady state performance. We envision that this latter step is crucial to foster the adoption of Java microbenchmarking in CI.

Developer configurations for warmup iterations wi and measurement iterations i are derived from the JSON reports generated by JMH (details in Section 4.4). We do not consider other ways in which benchmarks might be executed by developers, by using, for example, different command line arguments at launch time or by configuring additional parameters in build automation pipelines. Nonetheless, to the best of our knowledge, no study so far showed evidence of microbenchmarks being used as part of build automation pipelines (Beller et al. 2017; Rausch et al. 2017; Vassallo et al. 2017).

Following the methodology of Laaber et al. (2020), we performed post-hoc analysis to derive measurements for developers and dynamic configurations. Indeed, a comprehensive execution of all benchmarks across all developer static configurations and dynamic reconfiguration alternatives would have made our experimental evaluation impractical due to extremely long execution times (the execution of all 586 benchmarks using our single setup took about 93 days). In order to mitigate the impact of post-hoc analysis on the validity of our study, we employed a carefully defined process to derive measurements. We first estimated the time spent in each warmup/measurement iteration using the average execution time of each iteration as observed in our microbenchmarking setup. Then, we derived the *estimated warmup time* wt and the set of measurements M^{conf} based on the JMH configurations provided by software developers or dynamic reconfiguration techniques. The detailed process to estimate wt and derive M^{conf} can be found in Section 4.4. Nevertheless, our results may be marginally affected by approximations in converting measurement samples from our configuration to the others.

Internal Validity The steady state is detected on the basis of execution time measurements. We do not consider other event-based stability criteria, such as JIT activity, as these are not part of any JMH report and, therefore, are not something we can compare against when examining developer configurations. However, considering such additional criteria may lead to a different steady state classification.

Before performing the change point detection, we filtered the outliers using Tukey's fences on a sliding window, as described in Section 4.3. The parametrization of this procedure might affect the detection of the steady state. However, we tried to select parameters that would result in a very conservative outliers filtering. In fact, we only filter 0.27% of the datapoints. Nonetheless, the outliers filtering is a necessary procedure when employing change point detection algorithms, as the vast majority of such algorithms cannot distinguish between actual changes and outliers (Fearnhead and Rigaill 2019), thus leading to an overestimation of changes.

External Validity We only focused on GitHub repositories. Therefore, it is unclear if the findings are valid for other open-source hosting platforms or industrial software. Nonetheless, we conducted our experiment on 30 systems, a number larger than most recent

empirical studies on performance (e.g., see Laaber and Leitner 2018; Laaber et al. 2020; Ding et al. 2020; Reichelt et al., 2019).

We chose to limit the scope of the experiment to JMH microbenchmarks, because JMH is a mature and widely adopted Java microbenchmark harness. We ran the benchmarks on JVM 8 (JDK 1.8.0 update 241) or 11 (JDK 11.0.6), depending on the requirements of the specific system. Using other JVM versions or JVMs from other vendors may change the results. All the benchmarks were executed on the same bare-metal server running Linux. Nothing can be said about other hardware characteristics or other operating systems.

Conclusion Validity The changepoint detection method we used assumes independence in the time series. Even when the data contain some dependence, changepoint methods can still be used, provided that larger penalty values are used (Antoch et al. 1997; Barrett et al. 2017). The penalty values we dynamically compute for each fork, as explained in Section 4.3, tend to be larger (the average penalty is 504.54) than the value used, for example, in Barrett et al. (2017). Also, we manually inspected some of the time series to ensure that the segmentation was reasonable given the goal of the experiment.

Wherever possible, we used appropriate statistical procedures with p-value and effect size measures to test the significance of the differences and their magnitude.

8 Related Work

There are different perspectives of tackling performance analysis of software systems, through models at runtime (Cortellessa et al. 2022; Giese et al. 2020), or by means of benchmarking. Recently, benchmarking technique has played a key role to discover potential performance flaws (Stefan et al. 2017).

One of the main problems with benchmarking results is the reliability of the data. Recently, different approaches have defined rigorous processes to interpret those data. For example, some approaches rely on statistical inference for identifying and measuring the reliability of benchmarking results (Kalibera and Jones 2013, 2020). Other approaches, instead, have presented performance analysis methodologies to extract data in a more reliable way (Georges et al. 2007). Barrett et al. (2017) introduced a fully automated statistical approach based on changepoint analysis. In their work, Barrett et al. (2017) studied a set of small and deterministic VM benchmarks across different types of VMs, including the JVM. They found that VM microbenchmarks may not always reach a steady state of performance.

On the other hand, performance benchmarking is a time-demanding process. Recently, some approaches investigated solutions to reduce the time for performance analysis while preserving reliable results (AlGhamdi et al. 2020; Mostafa et al. 2017; He et al. 2019). He et al. (2019) have studied the reduction of performance testing in the cloud. They introduced a statistical tool, namely PT4Cloud, that provides stop conditions in order to obtain reliable performance indices. Another way to reduce testing time is, for example, by reusing the "functional" unit tests, which are likely available and maintained. For example, Bulej et al. (2017) extended "functional" unit tests with performance knowledge by equipping them with stochastic performance logic.

Java Microbenchmark Harness (JMH) is a popular benchmarking framework for Java software. JMH allows defining performance testing to reduce variability in the measurements as well as external factor contributions during the microbenchmark testing phase. Hence, different studies spanned over different JMH aspects (Costa et al. 2021; Samoaa and Leitner 2021; Laaber and Leitner 2018; Laaber et al. 2019). A JMH microbenchmark

might be affected by bad practices that could degrade performance results. Costa et al. (2021) studied those bad practices by analyzing a corpus of 123 OSS, and they extracted those bad practices that more likely lead to bad performance indicators. Laaber et al. (2020) focused their study on reducing the required execution time of microbenchmarking tests through dynamic reconfiguration. They have defined three stability criteria to dynamically estimate the end of the warmup phase and halt warmup iterations accordingly. Samoaa and Leitner (2021) studied, instead, the impact of benchmark parameters and how they affect performance results.

In this work, we studied the effectiveness of modern Java microbenchmarking for steady state performance assessment. Similarly to Barrett et al. (2017), we investigated whether microbenchmarks reach a steady state of performance. However, unlike them, we studied this aspect in the more defined scope of "testing-oriented" Java benchmarks, i.e., JMH benchmarks specifically designed to assess performance of a particular software. Our results are consistent with those gathered by Barrett et al. (2017), thus confirming that, even in a different context, Java benchmarks may not always reach a steady state of performance.

Costa et al. (2021) broadly studied bad practices in JMH benchmarks, instead we specifically investigated the effectiveness of developer configurations for steady state performance assessment. Laaber et al. (2020) presented dynamic reconfiguration as a viable alternative to developer static configurations. In their study, they compared dynamic reconfiguration to JMH default configurations, and they observed a significant reduction in execution time with a negligible loss of result quality. In our study, instead, we evaluated the effectiveness of dynamic reconfiguration for steady state performance assessment. Furthermore, we showed, through a rigorous comparison, that dynamic reconfiguration is significantly more effective than developer configurations and, as such, it produces less pronounced side effects.

9 Conclusion

This paper presents a comprehensive investigation on Java steady state performance assessment. Through a rigorous assessment, we showed that Java microbenchmarks do not always reach a steady state of performance, thus confirming the finding of Barrett et al. (2017) in the more defined scope of "testing-oriented" Java microbenchmarks.

Even when microbenchmarks consistently reach a steady state of performance, a reliable assessment remains far from trivial. According to our results, the current state-of-practice, which mostly relies on developer static configurations, show poor effectiveness for steady state performance assessment. Developers often fail to accurately estimate the end of the warmup phase, thereby causing either large time wastes or poor results quality. Dynamic reconfiguration provides a significant leap forward over developer static configurations by providing more accurate warmup estimates and less pronounced side effects. Still, the achieved results highlight non-trivial estimation errors, large time wastes, and distorted performance measurements.

The findings of our work have implications for both practitioners and researchers.

For the former, it is important to be aware that benchmark forks may not always reach a steady state of performance. The recommendation here is to perform an adequate number of forks (e.g., 10) to mitigate the noise introduced by "non-steady" forks. Another important lesson for practitioners is to favor dynamic reconfiguration over static configuration when possible. Indeed, when compared to developer configurations, dynamic reconfiguration techniques provide more accurate estimates of warmup time, though this improvement

may (sometimes) come at the expense of a more time-consuming performance assessment process. Further empirical studies are needed to assess whether this "cost" is acceptable for practitioners. Nonetheless, the achieved results are also helpful for suggesting which technique to use depending on the practitioner's need.

On the researchers' side, given the promising results of dynamic reconfiguration and the large room for improvement suggested by our investigation, we envision research aimed at designing novel and more effective dynamic reconfiguration techniques to (i) reduce the time effort devoted to performance assessment and (ii) strengthen the reliability of performance measurements. This is a direction we aim to investigate in future work.

We have made the code and the data used in our study publicly available to encourage further research on this topic.

Acknowledgements We thank the anonymous reviewers for their constructive comments, which guided us in improving the paper. Luca Traini is grateful for the financial support by "Fondo Territori Lavoro e Conoscenza CGIL, CSIL and UIL" through the project "Territori Aperti". Daniele Di Pompeo is supported by the EMERGE project at Centre of EXcellence on Connected, Geo-Localized and Cybersecure Vehicle (EX-Emerge), funded by Italian Government under CIPE resolution n. 70/2017. Michele Tucci is supported by the OP RDE project No. CZ.02.2.69/0.0/0.0/18_053/0016976 "International mobility of research, technical and administrative staff at Charles University".

Data Availability To aid reproducibility we provide the data and scripts needed to replicate our findings. The complete replication package is available at DOI:10.5281/zenodo.7058361

References

AlGhamdi HM, Bezemer CP, Shang W, Hassan AE, Flora P (2020) Towards reducing the time needed for load testing. J Softw: Evol Process e2276. https://doi.org/10.1002/smr.2276. https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.2276, smr.2276

Antoch J, Hušková M, Prášková Z (1997) Effect of dependence on statistics for determination of change. J Stat Plan Inference 60(2):291–310. https://doi.org/10.1016/S0378-3758(96)00138-3. https://www.sciencedirect.com/science/article/pii/S0378375896001383

Bagley D, Fulgham B, Gouy I (2004) The computer language benchmarks game. https://benchmarksgame-team.pages.debian.net/benchmarksgame. Accessed: 2021-10-12

Barrett E, Bolz-Tereick CF, Killick R, Mount S, Tratt L (2017) Virtual machine warmup blows hot and cold. Proc ACM Program Lang 1(OOPSLA). https://doi.org/10.1145/3133876

Beller M, Gousios G, Zaidman A (2017) Oops, my tests broke the build: an explorative analysis of travis ci with github. In: 2017 IEEE/ACM 14th international conference on mining software repositories (MSR), pp 356–367. https://doi.org/10.1109/MSR.2017.62

Bolz CF, Tratt L (2015) The impact of meta-tracing on vm design and implementation. Sci Comput Program 98(P3):408–421. https://doi.org/10.1016/j.scico.2013.02.001

Bulej L, Bures T, Horký V, Kotrc J, Marek L, Trojánek T, Tuma P (2017) Unit testing performance with stochastic performance logic. Autom Softw Eng 24(1):139–187. https://doi.org/10.1007/s10515-015-0188-0

Chen J, Shang W (2017) An exploratory study of performance regression introducing code changes. In: 2017 IEEE International conference on software maintenance and evolution, ICSME 2017, Shanghai, China, September 17–22, 2017. IEEE Computer Society, pp 341–352. https://doi.org/10.1109/ICSME.2017.13

Cohen J (2013) Statistical power analysis for the behavioral sciences. Taylor & Francis

Cortellessa V, Di Pompeo D, Eramo R, Tucci M (2022) A model-driven approach for continuous performance engineering in microservice-based systems. J Syst Softw 183:111084. https://doi.org/10.1016/j.jss.2021.111084. https://www.sciencedirect.com/science/article/pii/S0164121221001813

Costa D, Bezemer CP, Leitner P, Andrzejak A (2021) What's wrong with my benchmark results? Studying bad practices in jmh benchmarks. IEEE Trans Softw Eng 47(7):1452–1467. https://doi.org/10.1109/TSE.2019.2925345

Davison AC, Hinkley DV (1997) Bootstrap methods and their application. Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge University Press, Cambridge. https://doi.org/10.1017/CBO9780511802843

Ding Z, Chen J, Shang W (2020) Towards the use of the readily available tests from the release pipeline as performance tests: are we there yet? In: Rothermel G, Bae D (eds) ICSE '20: 42nd international conference on software engineering, Seoul, South Korea, 27 June–19 July, 2020. ACM, pp 1435–1446. https://doi.org/10.1145/3377811.3380351

Eckley IA, Fearnhead P, Killick R (2011) Analysis of changepoint models. Cambridge University Press, Cambridge, pp 205–224. https://doi.org/10.1017/CBO9780511984679.011

Fearnhead P, Rigaill G (2019) Changepoint detection in the presence of outliers. J Am Stat Assoc 114(525):169–183

Fieller EC (1954) Some problems in interval estimation. J R Stat Soc B: Stat (Methodol) 16(2):175–185. http://www.jstor.org/stable/2984043

Fowler M (2006) Continuous integration. https://www.martinfowler.com/articles/continuousIntegration.html. Accessed: 25 Jan 2022

Georges A, Buytaert D, Eeckhout L (2007) Statistically rigorous java performance evaluation. In: Proceedings of the 22nd annual ACM SIGPLAN conference on object-oriented programming systems, languages and applications, OOPSLA '07. Association for Computing Machinery, New York, pp 57–76. https://doi.org/10.1145/1297027.1297033

Giese H, Lambers L, Zöllner C (2020) From classic to agile: experiences from more than a decade of project-based modeling education. In: Guerra E, Iovino L (eds) MODELS '20: ACM/IEEE 23rd international conference on model driven engineering languages and systems, virtual event, Canada, 18–23 October, 2020, companion proceedings. ACM, pp 22:1–22:10. https://doi.org/10.1145/3417990.3418743

Haynes K, Eckley IA, Fearnhead P (2014) Efficient penalty search for multiple changepoint problems. 1412.3617

He S, Manns G, Saunders J, Wang W, Pollock L, Soffa ML (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, ESEC/FSE 2019. Association for Computing Machinery, New York, pp 188–199. https://doi.org/10.1145/3338906.3338912

Jiang ZM, Hassan AE (2015) A survey on load testing of large-scale software systems. IEEE Trans Softw Eng 41(11):1091–1118. https://doi.org/10.1109/TSE.2015.2445340

Kalibera T, Jones R (2013) Rigorous benchmarking in reasonable time. In: Proceedings of the 2013 international symposium on memory management, ISMM '13, pp 63–74. Association for Computing Machinery, New York. https://doi.org/10.1145/2491894.2464160

Kalibera T, Jones R (2020) Quantifying performance changes with effect size confidence intervals. 2007.10899

Killick R, Fearnhead P, Eckley IA (2012) Optimal detection of changepoints with a linear computational cost. J Am Stat Assoc 107(500):1590–1598. https://doi.org/10.1080/01621459.2012.737745

Kullback S, Leibler RA (1951) On information and sufficiency. Ann Math Stat 22(1):79–86

Laaber C, Leitner P (2018) 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. Association for Computing Machinery, New York, pp 119–130. https://doi.org/10.1145/3196398.3196407

Laaber C, Scheuner J, Leitner P (2019) Software microbenchmarking in the cloud. how bad is it really? Empir Softw Eng 24(4):2469–2508. https://doi.org/10.1007/s10664-019-09681-1

Laaber C, Würsten S, Gall HC, Leitner P (2020) Dynamically reconfiguring software microbenchmarks: reducing execution time without sacrificing result quality. In: Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering, ESEC/FSE 2020. Association for Computing Machinery, New York, pp 989–1001. https://doi.org/10.1145/3368089.3409683

Laaber C, Gall HC, Leitner P (2021) Applying test case prioritization to software microbenchmarks. Empir Softw Eng 26(6):133. https://doi.org/10.1007/s10664-021-10037-x

Lavielle M (2005) Using penalized contrasts for the change-point problem. Signal Process 85(8):1501–1510. https://doi.org/10.1016/j.sigpro.2005.01.012

Leitner P, Bezemer CP (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, ICPE '17. Association for Computing Machinery, New York, pp 373–384. https://doi.org/10.1145/3030207.3030213

Maricq A, Duplyakin D, Jimenez I, Maltzahn C, Stutsman R, Ricci R (2018) Taming performance variability. In: 13th USENIX symposium on operating systems design and implementation (OSDI 18). USENIX Association, Carlsbad, pp 409–425. https://www.usenix.org/conference/osdi18/presentation/maricq

Mostafa S, Wang X, Xie T (2017) Perfranker: prioritization of performance regression tests for collection-intensive software. In: Bultan T, Sen K (eds) Proceedings of the 26th ACM SIGSOFT international symposium on software testing and analysis, Santa Barbara, CA, USA, July 10–14, 2017. ACM, pp 23–34. https://doi.org/10.1145/3092703.3092725

Mytkowicz T, Diwan A, Hauswirth M, Sweeney PF (2009a) Producing wrong data without doing anything obviously wrong! In: Soffa ML, Irwin MJ (eds) Proceedings of the 14th international conference on architectural support for programming languages and operating systems, ASPLOS 2009, Washington, DC, USA, March 7–11, 2009. ACM, pp 265–276. https://doi.org/10.1145/1508244.1508275

Mytkowicz T, Diwan A, Hauswirth M, Sweeney PF (2009b) Producing wrong data without doing anything obviously wrong!. In: Soffa ML, Irwin MJ (eds) Proceedings of the 14th international conference on architectural support for programming languages and operating systems, ASPLOS 2009, Washington, DC, USA, March 7–11, 2009. ACM, pp 265–276. https://doi.org/10.1145/1508244.1508275

Neumann G, Harman M, Poulding SBarros M, Labiche Y (eds) (2015) Transformed vargha-delaney effect size. Springer International Publishing, Cham

Oaks S (2014) Java performance—the definitive guide: getting the most out of your code. O'Reilly. http://shop.oreilly.com/product/0636920028499.do

Papadopoulos AV, Versluis L, Bauer A, Herbst N, von Kistowski J, Ali-Eldin A, Abad CL, Amaral JN, Tuma P, Iosup A (2021) Methodological principles for reproducible performance evaluation in cloud computing. IEEE Trans Softw Eng 47(8):1528–1543. https://doi.org/10.1109/TSE.2019.2927908

Ratanaworabhan P, Livshits B, Simmons D, Ba Zorn (2009) Jsmeter: characterizing real-world behavior of javascript programs. Tech. Rep. MSR-TR-2009-173. https://www.microsoft.com/en-us/research/publication/jsmeter-characterizing-real-world-behavior-of-javascript-programs/

Rausch T, Hummer W, Leitner P, Schulte S (2017) An empirical analysis of build failures in the continuous integration workflows of java-based open-source software. In: 2017 IEEE/ACM 14th international conference on mining software repositories (MSR), pp 345–355. https://doi.org/10.1109/MSR.2017.54

Reichelt DG, Kühne S, Hasselbring W (2019) Peass: a tool for identifying performance changes at code level. In: 34th IEEE/ACM international conference on automated software engineering, ASE 2019, San Diego, CA, USA, November 11–15, 2019. IEEE, pp 1146–1149. https://doi.org/10.1109/ASE.2019.00123

Rubin J, Rinard M (2016) The challenges of staying together while moving fast: an exploratory study. In: Proceedings of the 38th international conference on software engineering, ICSE '16. Association for Computing Machinery, New York, pp 982–993. https://doi.org/10.1145/2884781.2884871

Samoaa H, Leitner P (2021) An exploratory study of the impact of parameterization on jmh measurement results in open-source projects. In: Proceedings of the ACM/SPEC international conference on performance engineering, ICPE '21. Association for Computing Machinery, New York, pp 213–224. https://doi.org/10.1145/3427921.3450243

Sarro F, Petrozziello A, Harman M (2016) Multi-objective software effort estimation. In: Proceedings of the 38th international conference on software engineering, ICSE '16. Association for Computing Machinery, New York, pp 619–630. https://doi.org/10.1145/2884781.2884830

Satopaa V, Albrecht JR, Irwin DE, Raghavan B (2011) Finding a "kneedle" in a haystack: detecting knee points in system behavior. In: 31st IEEE international conference on distributed computing systems workshops (ICDCS 2011 workshops), 20–24 June 2011, Minneapolis, Minnesota, USA. IEEE Computer Society, pp 166–171. https://doi.org/10.1109/ICDCSW.2011.20

Stefan P, Horký V, Bulej L, Tuma P (2017) Unit testing performance in java projects: are we there yet? In: Binder W, Cortellessa V, Koziolek A, Smirni E, Poess M (eds) Proceedings of the 8th ACM/SPEC on international conference on performance engineering, ICPE 2017, L'Aquila, Italy, April 22–26, 2017. ACM, pp 401–412. https://doi.org/10.1145/3030207.3030226

Suchanek M, Navratil M, Bailey L, Boyle C (2017) Performance tuning guide (red hat enterprise Linux 7). https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/performance_tuning_guide/, (Online; accessed 28 June 2021)

Traini L (2022) Exploring performance assurance practices and challenges in agile software development: an ethnographic study. Empir Softw Eng 27(3):74. https://doi.org/10.1007/s10664-021-10069-3

Traini L, Di Pompeo D, Tucci M, Lin B, Scalabrino S, Bavota G, Lanza M, Oliveto R, Cortellessa V (2021) How software refactoring impacts execution time. ACM Trans Softw Eng Methodol 31(2). https://doi.org/10.1145/3485136

Tukey JW et al (1977) Exploratory data analysis, vol 2. Reading

Vargha A, Delaney HD (2000) A critique and improvement of the “cl” common language effect size statistics of Mcgraw and Wong. J Educ Behav Stat 25(2):101–132. http://www.jstor.org/stable/1165329
Vassallo C, Schermann G, Zampetti F, Romano D, Leitner P, Zaidman A, Di Penta M, Panichella S (2017) A tale of ci build failures: an open source and a financial organization perspective. In: 2017 IEEE International conference on software maintenance and evolution (ICSME), pp 183–193. https://doi.org/10.1109/ICSME.2017.67

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Luca Traini is a postdoctoral researcher in the Department of Computer Science and Engineering, and Mathematics at the University of L'Aquila, Italy. His research interests centre around software performance engineering, encompassing both human and technical aspects, with the goal of improving techniques and methodologies for software performance assurance. His current research is focused on performance assurance processes, performance testing and debugging, and the interplay between software maintenance and performance. More information at: https://lucatraini.me.

Vittorio Cortellessa is Full Professor at the Department of Computer Science and Engineering, and Mathematics of University of L'Aquila. He had received his Ph.D. in Computer Science at University of Roma Tor Vergata in 1995. Between 1996 and 1999 he held postdoc positions at the same institution and at European Space Agency. In 2000 and 2001 he has been Research Assistant Professor at the Computer Science and Electrical Engineering Department of West Virginia University. Since 2022 he is at University of L'Aquila. His main research interests are in the areas of Software Performance, Software Reliability, and Model-Driven Engineering. He has published more than 120 papers on international conferences and journals in these areas, and he has co-authored a monographic book on Software Performance. He has served and serves in program committees and editorial boards of conference and journals in the Software Engineering domain. He currently is Co-Chair of the Steering Committee of ASM/SPEC International Conference on Performance Engineering (ICPE). More information at: http://people.disim.univaq.it/cortelle/.

Daniele Di Pompeo is a postdoctoral researcher at the Centre of EXcellence on Connected, Geo-localized and Cyber-secure vehicles (ExEmerge) with a focus on performance analysis. He received his Ph.D. in Information and Communication Technologies from the University of L'Aquila in 2019. His research interests include software performance analysis, architecture optimization and refactoring of software systems.

Michele Tucci is a postdoctoral researcher in the Department of Distributed and Dependable Systems of the Faculty of Mathematics and Physics at Charles University in Prague, Czech Republic. He received his Ph.D. in computer science from the University of L'Aquila, Italy, in 2021, where he was advised by Romina Eramo and Vittorio Cortellessa. His current research interests include performance regression testing, software refactoring, and optimization of software architectures towards quality aspects.