What's Wrong With My Benchmark Results? Studying Bad Practices in JMH Benchmarks

Article *in* IEEE Transactions on Software Engineering · June 2019

DOI: 10.1109/TSE.2019.2925345

4 authors:

Diego Elias Costa
Concordia University Montreal
28 PUBLICATIONS **100** CITATIONS
SEE PROFILE

Cor-Paul Bezemer
University of Alberta
71 PUBLICATIONS **919** CITATIONS
SEE PROFILE

Philipp Leitner
Chalmers University of Technology
149 PUBLICATIONS **2,959** CITATIONS
SEE PROFILE

Artur Andrzejak
Universität Heidelberg
120 PUBLICATIONS **2,380** CITATIONS
SEE PROFILE

What's Wrong With My Benchmark Results? Studying Bad Practices in JMH Benchmarks

Diego Costa, Cor-Paul Bezemer, Philipp Leitner, Artur Andrzejak

Abstract—Microbenchmarking frameworks, such as Java's Microbenchmark Harness (JMH), allow developers to write fine-grained performance test suites at the method or statement level. However, due to the complexities of the Java Virtual Machine, developers often struggle with writing expressive JMH benchmarks which accurately represent the performance of such methods or statements. In this paper, we empirically study bad practices of JMH benchmarks. We present a tool that leverages static analysis to identify 5 bad JMH practices. Our empirical study of 123 open source Java-based systems shows that each of these 5 bad practices are prevalent in open source software. Further, we conduct several experiments to quantify the impact of each bad practice in multiple case studies, and find that bad practices often significantly impact the benchmark results. To validate our experimental results, we constructed seven patches that fix the identified bad practices for six of the studied open source projects, of which six were merged into the main branch of the project. In this paper, we show that developers struggle with accurate Java microbenchmarking, and provide several recommendations to developers of microbenchmarking frameworks on how to improve future versions of their framework.

Index Terms—Performance testing, microbenchmarking, JMH, bad practices, static analysis

1 Introduction

Performance characteristics, such as response time, latency or resource usage, are some of the most important non-functional properties of critical software systems. In many systems it is essential that performance issues are avoided, as they can have a devastating impact on the perceived quality of the software. For example, a one-second slowdown in the checkout processes would cost Amazon an estimated $1.6 billion per year [22].

To avoid such problems, the performance of a software system must be thoroughly tested. Similar to functional testing (e.g., through integration and unit tests), performance testing should be done at different granularities. The ultimate goal of performance testing is to achieve a good end-to-end performance. Even though the impact of the performance of each component on the end-to-end performance of a system is not necessarily linear, obvious performance issues at the component-level are likely to ripple through to the system-level. Hence, we must first ensure that the performance of each component of the system is adequate.

Microbenchmarking [1], [2] is an approach for precise performance evaluation of an isolated segment of code at the method, loop, or even statement level. Several frameworks have been proposed to facilitate specifying and executing microbenchmarks. For example, the Java Microbenchmark Harness (JMH) [1] is a popular framework for languages that target the Java Virtual Machine (JVM), such as Java and Scala. JMH makes it possible to obtain a statistical estimate of the execution time by providing a scaffold that runs the targeted segment of code a large number of times. In addition, JMH offers mechanisms to prevent a subset of JVM optimizations, which may lead to misleading performance results.

Unfortunately, despite the existence of microbenchmarking frameworks, writing microbenchmarks that yield correct insights on the performance of the benchmarked task turns out to be challenging. Prior work established that there exist several pitfalls to avoid [20] and that many developers struggle with writing appropriate microbenchmarks [37]. In addition, the execution results of real-life microbenchmarks of large open source projects can vary significantly across repeated executions in identical environments [30].

In this paper, we empirically study bad practices of writing microbenchmarks that can lead to misleading benchmark results. In particular, we study the occurrence of 5 bad practices, which were extracted from the JMH documentation, in the JMH microbenchmark suites of 123 Java-based open source projects. In addition, we quantify the impact of the identified bad practices on the benchmark results of six open source projects. Our study addresses the following research questions:

RQ1: *How frequently do bad JMH practices occur in real-life open source software?* To assist developers with avoiding bad practices, we implemented a plugin (`SpotJMHBugs` [16]) for the `SpotBugs` static analysis tool. `SpotJMHBugs` is able to automatically identify the bad JMH practices that are discussed in our study. Using the tool, we found that 35 of 123 studied projects had at least one instance of a bad JMH practice in their benchmark suites. In 12 of these projects, there were more than 10 instances of bad JMH practices. The most frequently occurring bad

- *Diego Costa and Artur Andrzejak are with the Institute of Computer Science, Heidelberg University, Germany. E-mail: {diego.costa|artur.andrzejak}@informatik.uni-heidelberg.de*
- *Cor-Paul Bezemer heads the Analytics of Software, Games and Repository Data (ASGAARD) lab at the University of Alberta, Canada. E-mail: bezemer@ualberta.ca*
- *Philipp Leitner is with the Software Engineering Division at Chalmers and the University of Gothenburg, Sweden. E-mail: philipp.leitner@chalmers.se*

JMH practice is using accumulation to consume computation in a loop, which could yield unreliable benchmark results.

RQ2: *What is the impact of the identified bad JMH practices on the benchmark results?*
We manually fixed 105 benchmarks across 6 projects and measured the impact of the bad practices by comparing the results before and after the fix. In 73 of the 105 fixed benchmarks, the bad JMH practice significantly impacted the benchmark results.

To evaluate whether developers in practice care about these bad practices, we submitted 7 pull requests containing fixes for 57 benchmarks from 6 projects. Six of these pull requests were accepted and merged into the main project repository, which demonstrates the significance of our work. In particular, our study shows that despite the JMH documentation warning explicitly about all five discussed bad JMH practices, they are still prevalent in open source software. Even worse, we show that these bad practices have a significant impact on the microbenchmarks, which often leads to unreliable benchmark results.

2 Background

In this section, we present the terminology that is used throughout this paper, and introduce the set of studied bad JMH practices.

2.1 JMH Microbenchmarks

Performance testing [43] is used to experimentally assess one or more non-functional quality attributes of software systems. In this paper, we focus on one specific performance testing approach, namely software microbenchmarking. In contrast to stress or load tests, which test the end-to-end performance of a system, microbenchmarks are relatively short-running and aim at measuring the fine-grained performance of specific units of program code. For instance, a microbenchmark may measure method-level execution times of a class, the performance of a specific data structure, or the implementation of an algorithm. Consequently, microbenchmarks are typically not used to evaluate system-level service level agreements, but rather to ensure the performance of critical low-level code components or to compare different implementation alternatives.

For Java, the Java Microbenchmark Harness (JMH) [1] is commonly used for microbenchmarking [39]. JMH is a tool developed under the OpenJDK umbrella that allows users to specify benchmarks through Java annotations, using a syntax that is similar to the well-known JUnit framework [40]. Every public method that is annotated with `@Benchmark` is executed as part of the microbenchmark suite. Listing 1 shows an example benchmark from the `RxJava` project, where the execution time and throughput of a latched observer are measured.

```
public class ComputationSchedulerPerf {

    @State(Scope.Thread)
    public static class Input
      extends InputWithIncrementingInteger {
        @Param({ "100" , "1000" })
        public int size;
    }

    @Benchmark
    public void observeOn(Input input) {
        LatchedObserver<Integer> o =
          input.newLatchedObserver();
        input.observable.observeOn(
          Schedulers.computation()
        ).subscribe(o);
        o.latch.await();
    }
}
```

Listing 1: JMH example from the `RxJava` project.

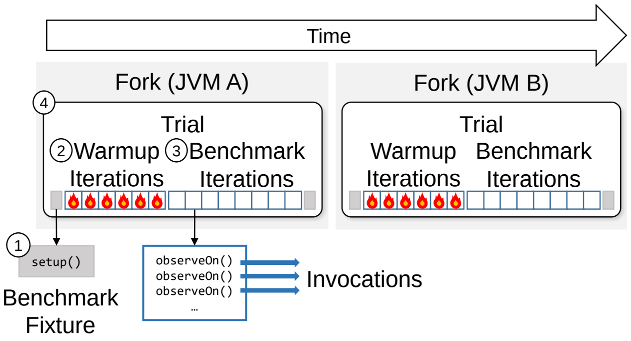

Fig. 1: JMH execution flow for a microbenchmark configured with 2 forks, 6 benchmark warmups and 8 benchmark iterations. In this example, the fixture methods are configured to be executed before and after each trial.

In Figure 1 we illustrate the execution flow that JMH uses to evaluate microbenchmarks in four major steps. (1) Initially, an optional *benchmark fixture* is invoked. The benchmark fixture is the code which initializes the benchmark environment (e.g., by filling a data structure with test data). (2) Afterwards, a defined number of *warmup iterations* are executed. These are identical to benchmark iterations (see below), but their results are discarded. Warmup iterations are intended to bring the JVM into a steady state (e.g., execute all applicable just-in-time compilations). (3) In the actual benchmark phase, a defined number of *benchmark iterations* is executed. Each iteration takes a defined amount of time (typically $1s$), during which the framework repeatedly calls the method annotated with `@Benchmark` (a single *invocation* in JMH parlor) and records all configured *performance counters* (e.g., throughput, execution time, latency). (4) One entire run of steps 1-3 (executing the benchmark fixture, zero to many warmup iterations, one to many benchmark iterations) is called a *trial*. By default, JMH executes each trial in a separate VM, and developers may specify how many *forks* each benchmark will sequentially execute (e.g., we present an example with two forks in Figure 1).

After the last iteration, JMH reports a summary of the results to the user and/or saves the results into an XML or JSON file. *Parameters* (e.g., `int size` in Listing 1) can be used to easily define different *benchmark instances* of the same method. In the example in Listing 1, JMH will effectively execute the `observeOn(Input)` benchmark twice; once with `size = 100` and once with `size = 1000`.

TABLE 1: Bad JMH practices collected from the JMH documentation.

ID	Bad JMH Practice Description	Undesired Effect
RETU	Not using a returned computation	Dead code elimination
LOOP	Using accumulation to consume values inside a loop	Loop optimization
FINAL	Using a `final` primitive for benchmark input	Constant folding
INVO	Running fixture methods for each benchmark method invocation	JMH overhead
FORK	Configuring benchmarks with zero forks	Profile-guided optimization

2.2 Bad JMH Practices in Benchmark Creation

While JMH offers an infrastructure that generates the boilerplate code for a microbenchmark from user-specified code and annotations, the responsibility of creating a reliable and correct benchmark remains with the developer.

Unfortunately, creating microbenchmarks which accurately represent the performance of the benchmarked code is difficult. There exist several coding pitfalls and bad JMH practices that can affect the reliability and correctness of a microbenchmark, as illustrated by the 38 sample benchmarks [35] in the JMH documentation [1]. In the remainder of this section, we discuss the most important bad JMH practices described in the JMH documentation (see Table 1 for an overview). All code examples in this section were taken from the JMH samples.

Bad JMH practice 1: Not using a result that is returned by a method in the benchmark (RETU)

Description: A benchmark typically calls one or more methods from the main application code. If such a method returns a result that is not used in the benchmark, the JVM may consider part of the called method as "dead code" and eliminate that part.

Symptoms: Because the call to the benchmarked method was eliminated, the code will appear faster than in actual usage. Listing 2 shows an example of the RETU bad JMH practice and two possible solutions. In `measureWrong()`, `Math.log(x1)` is redundant and may be eliminated by the JVM.

Solution: Every object that is returned by a method called directly from the benchmark should be used in the benchmark method. In `measureRight1()`, `Math.log(x1)` is used as a return of the benchmark method, and therefore, not eliminated. Alternatively, the JMH infrastructure offers a `Blackhole` object which can be used to prevent dead-code elimination by consuming the result. In `measureRight2()`, `Math.log(x1)` is consumed by a `Blackhole` object.

Bad JMH practice 2: Using accumulation to consume computation inside a loop (LOOP)

Description: Developers often have to design benchmarks that measure a method call within a loop. If the method call returns a numeric variable, it is intuitive to accumulate the returned objects as a way of avoiding dead-code elimination. However, this leads to another set of JVM optimizations that optimizes the code beyond what would be expected in real usage.

Symptoms: The code appears faster than in actual usage, as the loop can be extensively optimized by the JVM. Listing 3 shows an example of the LOOP bad JMH practice. The `measureWrong()` method executes every `work()` method as intended, but the JVM is able to unroll the loop and merge operations between two distinct `work()` calls. Such optimizations are only performed because an accumulation is used instead of a proper consume method, and will not hold in a scenario where the application actually uses or stores the return of each method call.

```java
private double x1;
private double x2;

@Benchmark
public double measureWrong() {
    Math.log(x1);
    return Math.log(x2);
}

@Benchmark
public double measureRight1() {
    return Math.log(x1) + Math.log(x2);
}

@Benchmark
public double measureRight2(Blackhole bh) {
    bh.consume(Math.log(x1));
    bh.consume(Math.log(x2));
}
```

Listing 2: Example of the RETU bad JMH practice and two possible solutions.[1]

```java
private int[] xs;

@Benchmark
public int measureWrong() {
    int acc = 0;
    for (int x : xs) {
        acc += work(x);
    }
    return acc;
}

@Benchmark
public void measureRight(Blackhole bh) {
    for (int x : xs) {
        bh.consume(work(x));
    }
}
```

Listing 3: Example of the LOOP bad JMH practice and a possible solution.[2]

Solution: The user should avoid using accumulation as a method of consuming the numeric return and use the `Blackhole` facilities instead. The `measureRight()` method shows how a loop can be benchmarked more safely.

Bad JMH practice 3: Using a `final` primitive for benchmark input (FINAL)

Description: If the JVM realizes that the result of a computation is predictable, it will optimize the computation (i.e.,

[1] https://bit.ly/2uyb79b

[2] https://bit.ly/2UfpVI1

```
private double x = Math.PI;
private final double wrongX = Math.PI;

@Benchmark
public double measureWrong() {
    // Computation is predictable
    return Math.log(wrongX);
}

@Benchmark
public double measureRight() {
    // Computation is not predictable
    return Math.log(x);
}
```

Listing 4: Example of the FINAL bad JMH practice and a possible solution.[3]

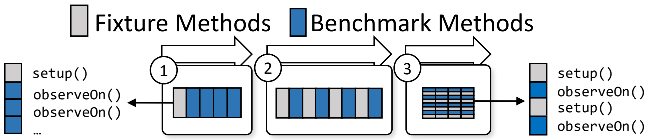

Fig. 2: JMH setup execution flow when configured to run (1) before a benchmark trial, (2) before each benchmark iteration and (3) before each benchmark invocation. The third option has several drawbacks and is classified as a bad practice (INVO) for short-running benchmarks.

using *constant folding* [41]), thereby affecting the runtime of the benchmark.

Symptoms: Because the constant computation was folded, the code will appear faster than in acual usage. Listing 4 shows an example of the FINAL bad JMH practice. The `measureWrong()` method does a computation using the `Math.PI` constant, making the result of the computation predictable and hence foldable.

Solution: Benchmark inputs should always be read from non-final primitive instance fields or reference instance fields. The `measureRight()` method in Listing 4 conducts the computation in a proper way, since its result is not predictable at compile time.

Bad JMH practice 4: Running fixture methods for each benchmark method invocation (INVO)

Description: Fixture methods are methods that are used to setup or tear down a benchmark. In Figure 2 we illustrate the three levels at which JMH allows fixture methods to run: (1) before/after a benchmark trial, (2) before/after a benchmark iteration and (3) before/after a benchmark method invocation. In most cases, it is a bad JMH practice to use the third option, as the overhead of the JMH infrastructure might be large compared to the actual benchmark runtime.

Symptoms: The JMH infrastructure must add a timestamp to each method invocation to calculate its execution time, as the time spent in the fixture methods is excluded from the performance measurement. On short-running benchmarks (ones that typically run for less than a millisecond), JMH saturates the system with timestamp requests, offsetting the measurements. According to the JMH documentation, this level might also omit problems stemming from time measurement, introducing unexpected and surprising results. Listing 5 shows an example of the INVO bad practice. As fixture methods are shared among benchmarks of the same class, the overhead caused by JMH will offset the measurements of all benchmarks, including the ones that do not access objects created in the setup/teardown methods.

Solution: Every invocation-level fixture method should be checked to make sure that it is necessary to be called at the invocation level. This necessity is rare and can be avoided in most situations by including the contents of the fixture method in the benchmark method (causing less overhead).

```
@TearDown(Level.Invocation)
public void check() {
    assert x > Math.PI : "Nothing changed?";
}
```

Listing 5: Example of the INVO bad JMH practice.[4]

Bad JMH practice 5: Configuring benchmarks with zero forks (FORK)

Description: The JVM is good at profile-guided optimization, i.e., optimization that is based on the usage profile of a method. However, such optimizations should be avoided in benchmarking, since a profile that was optimized for one benchmark may be reused across other benchmarks. In addition to the profile-guided optimization, running a non-forked benchmark may cause the JMH infrastructure to omit JVM options and compiler hints. These options and hints could be paramount to ensuring the correctness of the benchmark results. To avoid profile-guided optimization and risk affecting the execution correctness, each benchmark should execute in its own VM. Running a benchmark per VM is the JMH default behaviour, however, it is possible to override this behaviour using the `@Forks` annotation.

Symptoms: The code can appear faster or slower than in actual usage, depending on the optimized profile that was used by the JVM. JMH sample #12 contains a runnable example of a case where profile-guided optimization leads to unreliable benchmark results.[5]

Solution: Do not override the default JMH behaviour for running a benchmark trial per VM unless there is a very good reason to do so. In fact, since 2016, JMH issues a message when executed with zero forks to warn developers about potentially unreliable benchmarking results.

3 Methodology

In this paper, we present our study on bad JMH practices in benchmarks. Our study has three main goals:

1) Identify how frequently bad JMH practices occur in open source software projects (Section 4).
2) Study the performance impact of the used bad practices on the benchmark results of those projects (Section 5).
3) Validate our findings with the developers of those projects, by proposing patches to address the identified bad practices (Section 6).

[3] https://bit.ly/2TEJZzh

[4] https://bit.ly/2TZGkB6

[5] https://bit.ly/2PAHi3L

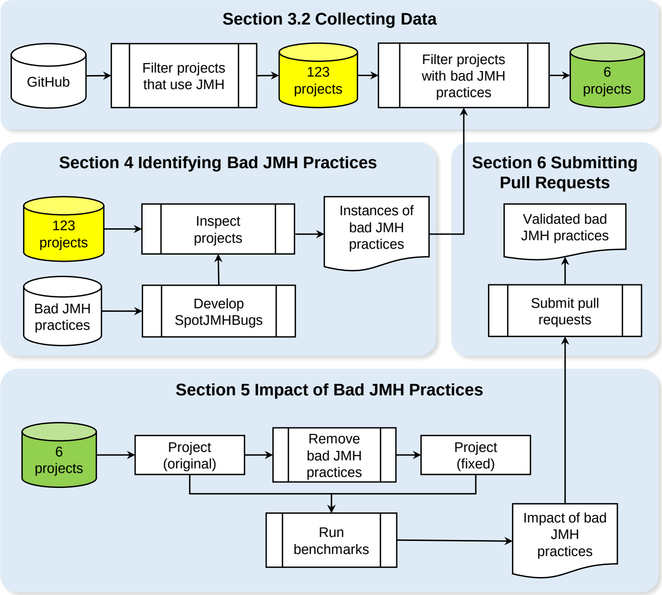
Section 3.2 Collecting Data
GitHub
Filter projects that use JMH
123 projects
Filter projects with bad JMH practices
6 projects
Section 4 Identifying Bad JMH Practices
123 projects
Inspect projects
Instances of bad JMH practices
Bad JMH practices
Develop SpotJMHBugs
Section 6 Submitting Pull Requests
Validated bad JMH practices
Submit pull requests
Section 5 Impact of Bad JMH Practices
6 projects
Project (original)
Remove bad JMH practices
Project (fixed)
Run benchmarks
Impact of bad JMH practices

Fig. 3: Overview of our study approach.

In the following, we present the methodology of our study. An overview is also provided in Figure 3.

3.1 Identifying Instances of Bad JMH Practices

To identify the usage of bad JMH practices in the studied projects, we built a static code analyzer, SpotJMHBugs, for the JMH benchmarks. Our analysis tool was implemented as a plugin for the SpotBugs[6] tool, the successor of FindBugs. SpotJMHBugs is a rule-based tool that analyzes Java bytecode, identifies bad JMH practices, and reports them to the developer. We discuss the SpotJMHBugs tool in more detail in Section 4.1.

3.2 Collecting Data

Below we present our methodology for collecting the data for our study on (1) the frequency of bad JMH practices and (2) the performance impact of these bad JMH practices on the benchmark results of the projects. The full list of projects and data sets that we base our results on can be found in our online appendix [17].

3.2.1 Data Collection for Studying the Frequency of Bad JMH Practices

We queried the 2017 GitHub snapshot (the latest snapshot available at the time of our study) using Google Bigquery[7] to identify open source Java projects that contain at least one JMH benchmark. Concretely, we query for source files that import `org.openjdk.jmh.annotations.Benchmark` and have at least one method annotated with `@Benchmark`. This led to a full data set of 839 projects.

In a second step, we remove forked projects to avoid biasing our analysis towards popular programs' characteristics. Projects such as `RxJava` had 16 forked versions (aside from the original from ReactiveX) and would distort our results. The set without forked projects contained 506 projects. As our SpotJMHBugs tool analyzes JVM bytecode, it requires a compiled project to analyze. As manually compiling and executing tests for a large number of projects is extremely time-consuming, if at all possible [30], we selected a subset of 123 projects that could be built automatically or with minimal intervention using Gradle or Maven for our study. Figure 4 shows the descriptive statistics of the selected projects in terms of their number of stars and subscribers. Our selected projects cover a range of popular and less-popular projects.

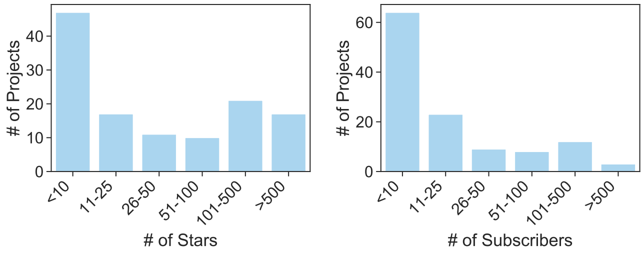

Fig. 4: Distribution of stars and subscribers of the 123 projects used to identify bad JMH practices.

3.2.2 Data Collection for Studying the Performance Impact of Bad JMH Practices

In the second part of our study, we study the performance impact of the bad JMH practices (Section 5). We limited our project selection further for this part of our study, as it requires manually addressing the instances of the identified bad JMH practices (and therefore requires knowledge about the project). To study the performance impact of bad JMH practices, we selected projects that matched the following criteria:

- The project is in the top-3 projects ranked by the number of stars on GitHub for a specific bad JMH practice.
- The project contains at least 2 instances of the identified bad JMH practice.

These selection criteria help us focus our efforts on (1) popular projects and (2) projects with multiple instances of a bad JMH practice, thereby reducing the effort that is necessary to address bad JMH practices.

Table 2 gives an overview of the projects for which we studied the performance impact of the followed bad JMH practices. Initially, our set of projects was composed of three projects per bad JMH practice. However, after careful inspection, we decide to remove two projects initially considered for the impact assessment of the FORK bad practice: `oopsla15-artifact` and `benchmark-arraycopy`. The first project was created as a paper artifact for the OOPSLA conference and the second is a series of benchmarks created to evaluate a specific library function. Hence, both are not good examples of production-quality open-source software, which is the primary subject of our study. Table 2 shows the list of projects ultimately selected for the impact

[6] https://spotbugs.github.io/
[7] https://cloud.google.com/bigquery/

TABLE 2: Selected projects per bad JMH practice. We select the top-3 most starred projects with at least two instances of each bad JMH practice. Column # BP shows the total number of bad practice instances that were identified in the three projects.

	Selected Projects			# BP
RETU	netty	gs-collections	logging-log4j2	54
LOOP	netty	druid	logging-log4j2	29
FINAL	netty	druid	logging-log4j2	9
INVO	netty	druid	h2o-3	18
FORK	pgjdbc			2
Total number of evaluated instances of bad JMH practices				112

assessment. Note that some projects (e.g., the `netty` project) were selected for multiple categories. For the performance impact study, we selected 6 projects which contain a total of 93 instances of the 5 studied bad JMH practices (see Section 2.1).

3.3 Assessing the Performance Impact of Bad JMH Practices

To assess the performance impact of bad JMH practices, we manually analyze the instances of the bad JMH practices that we identified in the 6 projects in Table 2. We then generate an alternative, "fixed" version of the benchmarks, by removing the bad practices according to the solutions proposed by the JMH documentation as stated in Section 2.2. It is of paramount importance that in our fix we do not introduce artificial latency or modify what has been measured in a benchmark. Most fixes are *non-intrusive*, and require simple code refactoring, such as consuming variables, removing the final modifier from a primitive field, or some level of benchmark reconfiguration.

However, one solution for removing the INVO bad practice required us to introduce the code from fixture methods inside the benchmark. We classify this particular solution as *intrusive* and benchmarks fixed with this solution are evaluated using a separate methodology.

Assessing the Impact of Non-Intrusive Fixes: At the end of our experiment we collect the resulting performance counters (in particular, we focus on the execution time of the benchmarks). If a bad JMH practice is irrelevant to the benchmark result, both the original and fixed version should lead to similar performance counter distributions. If the distributions differ significantly, we conclude that the bad practice impacts the benchmark result.

Assessing the Impact of Intrusive INVO Fixes: In the case of the INVO fix, we also collect and compare both performance counter distributions. If inserting the setup/teardown code inside a benchmark method makes the benchmark faster to execute, the JMH overhead takes longer than the time spent on fixture methods. In this case, we conclude that the benchmark was impacted by the INVO bad practice.

To test whether the distributions of performance counters for the original version and fixed version are statistically significantly different, we used the Wilcoxon non-parametric test [44] with a significance level of $\alpha = 0.01$. The Wilcoxon test only indicates whether the distributions have a statistically significant difference. However, the test does not indicate whether the difference is large enough to be noticeable in practice. To quantify the difference, we use Cliff's Delta effect size [8]. We use the following common thresholds [38] for interpreting the effect size:

$$\text{Effect size } d = \begin{cases} negligible(N), & \text{if } |d| \leq 0.147 \\ small(S), & \text{if } 0.147 < |d| \leq 0.33 \\ medium(M), & \text{if } 0.33 < |d| \leq 0.474 \\ large(L), & \text{if } 0.474 < |d| \leq 1 \end{cases}$$

It is important to keep in mind that a large effect size does not necessarily imply a large absolute difference in mean benchmark times, but rather in the mean benchmark times *as well as* the variation in these times. For example, if a benchmark consistently takes 100ms to execute before the fix, and 102ms after the fix, the difference could have a large effect size, even though the absolute difference between the mean benchmark times is fairly small. In the remainder of the paper, we clarify whether 'large' refers to the effect size of the difference, or the absolute mean benchmark times.

We consider the benchmark as *impacted* if it has at least one benchmark instance where the performance counters of the fixed version differ significantly from the original, with a non-negligible effect size. In the particular case of the INVO intrusive fix, we consider the benchmark as *impacted* if all benchmark instances yield performance counters comparable or faster than the original version.

We further define the *benchmark effect size*, as the highest absolute effect size observed in its instances. The reasoning behind this definition is that a benchmark should yield consistent results on all defined input parameters. A single set of parameters in a benchmark that is impacted by a bad JMH practice, is sufficient to mislead an analysis and affect the benchmark quality.

3.4 Evaluating Fixed Versions and Results with Developers

To evaluate the identified instances of bad JMH practices, and their assessed impact, we manually submitted pull requests to six open source projects (selected based on where we found the largest impact on the benchmark results). These pull requests contained the fixed versions that we constructed as part of our study. The goal of this step was two-fold. Firstly, we wanted to see whether the developers agree with our assessment that the original benchmarks produced misleading results. Secondly, we wanted to validate whether the developers agree with our fixes.

4 Identifying Bad JMH Practices

In this section, we present the results of our first RQ: *How frequently do bad JMH practices occur in real-life open source software?*

4.1 The `SpotJMHBugs` Tool

To investigate the occurrences of bad JMH practices in Java projects, we first derive a set of rules that can be used to identify such practices via static code analysis. We present

TABLE 3: Static rules used to identify bad JMH practices.

	Static Rule
RETU	Variable not consumed in the benchmark Ignored return of static method calls
LOOP	Numerical variable is accumulated in a loop
FINAL	Final and non-static primitives in @State classes
INVO	@Setup/@Teardown with invocation level
FORK	@Fork with a value of zero

TABLE 4: Distribution of identified instances of bad JMH practices in *all* 123 projects and on a subset of 49 projects containing at least 10 benchmarks.

	# of Identified Bad JMH Practices							
Dataset	0	1	2	3	4	5	6-10	+10
All projects	88	9	6	2	1	1	4	12
Projects with 10+ benchs	24	3	4	1	1	0	4	12

TABLE 5: The number of identified instances of bad JMH practices for all 123 studied projects. The 'Total' column shows the number of instances found in all projects per bad JMH practice.

Bad JMH Practice	Total	# of Projects	% of Projects
RETU	89	15	12.2
LOOP	128	16	13.0
FINAL	25	9	7.3
INVO	82	10	8.1
FORK	7	3	2.4

a brief description of each derived rule in Table 3. Bad JMH practices that relate to the benchmark configuration, such as INVO and FORK are easily verifiable and have unique static rules. For instance, FORK requires a simple check on the occurrence of a `@Fork` annotation with a value of 0.

The RETU, FINAL and LOOP bad JMH practices are related to the source-code and manifest themselves in different ways. In such cases, we use heuristics to identify scenarios in which the undesired JVM optimizations could happen. For instance, the RETU bad practice may occur when a variable is not properly consumed in the benchmark or when developers ignore the return of a static method call. Our rule for identifying unconsumed variables in a benchmark is based on the following principle. A local variable V is considered consumed if at least one of the following criteria are met:

1) V is stored into a class field.
2) V is returned at the end of the benchmark method.
3) V is consumed by a JMH Blackhole object.
4) There exists another variable V' that has a data dependency on V and V' is a consumed variable. To check if such a dependency exists, we build a data-dependency graph [42], and verify the existence of a path between V and V' in the graph.

Our rule reports every local variable that does not fulfill the above mentioned criteria and is therefore prone to dead-code elimination. For an explanation of the other rules, we refer the reader to the source-code documentation of `SpotJMHBugs` [16].

Our tool can be executed through a batch command using Maven, Gradle or Ant, or integrated with the Eclipse IDE. If used in conjunction with Eclipse, the warnings about bad JMH practices are shown directly in the editor view. `SpotJMHBugs` restricts its analysis to classes that contain at least one method annotated with `@Benchmark`, as bad JMH practices are only potentially harmful in the context of JMH benchmarks. Calls to methods outside of benchmark classes are skipped, keeping the analysis time short and primarily dependent on the JMH benchmarks, which tend to be a very small fraction of the overall application code base [31], [39].

4.2 Results

35 out of 123 projects (28%) contained at least one instance of a bad JMH practice. Table 4 shows the number of identified instances of bad JMH practices per project together with the number of benchmarks potentially affected by such instances. If we limit our consideration to the 49 projects with more than 10 benchmarks, the share of projects with at least one bad JMH practice increases to 51%. The number of benchmarks potentially affected by bad JMH practices varies considerably per project. In 23 projects we identified at most 10 instances, while we identified more than 10 instances in 12 other projects. In total, `SpotJMHBugs` identified 331 instances of bad practices in our dataset.

LOOP was the most commonly identified bad JMH practice, with a presence in 13% of the studied projects. Table 5 summarizes the number of identified instances for all bad practices. The second most commonly identified bad JMH practice was the RETU bad practice, occurring in 12% of the studied projects. FINAL and INVO occurred in respectively 9 (7%) and 10 (8%) of the studied projects. The FORK bad practice was identified in only 3 projects.

Table 6 gives a detailed overview of the distribution of the bad practices across the top 25 projects with the largest number of benchmarks in their benchmark suite in our dataset. Overall, the distribution of identified bad JMH practices appears to be very particular to each project. For example, although `RxJava` has 215 benchmarks, `SpotJMHBugs` did not identify any bad JMH practice in the project. On the other hand, we identified instances of four different bad JMH practices in the benchmarks of `netty`. In total, we found 22 bad practice instances in this project alone. Aside from FORK (which was found in only three projects) every bad JMH practice was found in at least six projects.

> 28% of the studied projects contained at least one instance of a bad JMH practice. Our results show that the studied bad JMH practices occur frequently in open source projects: LOOP was the most frequently ocurring bad JMH practice, but aside from FORK, all bad practices appeared in at least 6 projects.

TABLE 6: The number of instances of bad JMH practices that were found in the 25 studied projects with the largest benchmark suites. The projects in bold were selected for an experimental evaluation of the impact of bad JMH practices instances in the next step of the study. The 'Benchs' column denotes the number of `@Benchmark`-annotated methods, and 'Affected Benchs' column shows the amount of benchmarks containing at least one bad JMH practice. We also included the `h2o-3` project in the impact assessment which was not in the listed top-25 projects.

Project	Stars	Benchs	Affected Benchs		Bad JMH Practices				
			#	%	RETU	LOOP	FINAL	INVO	FORK
gs-collections	1652	451	48	10.6%	47				
logging-log4j2	256	346	17	4.9%	5	7	5		
RxJava	23,558	215							
oopsla15-artifact	16	213	25	11.7%	4	1	2	12	3
netty	9,746	159	29	18.2%	2	14	2	4	
reactive-streams-commons	106	157	2	1.2%	2				
druid	4,743	148	15	10.1%		8	2	2	
JCTools	1,053	92	3	3.2%		1		2	
golo-jmh-benchmarks	4	92							
zipkin	5,627	74							
microbenchmarks	7	67	17	25.3%		17	5		
xodus	248	66	52	78.7%		7		18	
lab-java8streamperformancebenchmark	4	64	6	9.3%		6			
mini2Dx	137	55							
fast-select	3	48							
jenetics	183	47	16	34.0%		8	1		
rtree	482	42							
byte-buddy	1,495	39							
caffeine	2,414	38	1	2.6%	1				
pgjdbc	322	35	5	14.2%					2
java-final-benchmark	10	34							
streamalg	15	34	4	11.7%		4			
pinot	1,475	31	15	48.3%		16			
cache2k-benchmark	12	28	9	32.1%			1		
template-compiler	12	27							
h2o-3	1,943	18	12	66.6%		6		12	

5 Impact of Bad JMH Practices

To answer our second RQ (*What is the impact of the identified bad JMH practices on the benchmark results?*), we fix the identified bad JMH practices in selected projects (Section 5.1) and compare the benchmark times of the fixed and original benchmarks (Section 5.2).

5.1 Generating Fixed Versions

We aim to evaluate the impact on performance of each bad JMH practice separately. Thus, in projects that had multiple identified bad practices, we generated multiple fixed versions, each containing fix-patches for a single bad practice. For instance, we identified three bad JMH practices in `druid`, thus we generated three fixed versions. In total, we fixed 93 instances of bad JMH practices, generating 13 fixed versions for 6 selected projects. Our concrete process for generating fixes differed per bad JMH practice. We detail how fixed versions were generated when discussing the results for each bad practice.

5.2 Running Benchmarks

There are a number of challenges in evaluating steady-state performance in a managed runtime environment, such as the JVM [20]. Various uncontrolled factors can impact the performance, such as the garbage collector, Just-in-Time compilation (JIT) and method sampling optimizations. As mentioned in Section 2.1, JMH helps to mitigate the uncontrolled factors by allowing developers to configure the number of warmup iterations, benchmark iterations and forks. This configuration is highly important to achieve reliable and repeatable results in a benchmark, e.g., too few warmup or benchmark iterations may yield a large variance in the performance counters of some benchmarks [30].

To avoid building our analysis on unreliable benchmark configurations, we repeat each experiment 5 times, while keeping the original benchmark configuration (i.e., the number of warmup and benchmark iterations, and forks). Our experiments generate a median of 780 performance counters (min=100, max=14,400) for each benchmark instance. Finally, we also alternate runs between the original and fixed versions to reduce the chances of circumstantial external influence impacting only one version of the program.

We conducted our experiments on a computational server with an E5-1660-3.3GHz CPU, with 6 physical cores and 64 GB RAM using Linux 3.16.0-53. The benchmarks use the JVM HotSpot 64 bits and jdk1.8.0_65 as the Java version. Aside from the basic operating system functionality, no other process was running during the execution of our experiments.

5.3 False Positives

We describe in this section, for each bad JMH practice, the criteria for filtering false-positives. False positives are bad JMH practices that were wrongly reported by our tool, due

TABLE 7: Identified bad JMH practice instances instances characterized as false-positives (FP) by further manual inspection. The 'TP' column shows the correctly identified cases by SpotJMHBugs and '# Benchs' shows the number of benchmarks that are potentially affected by the bad JMH practices.

Bad JMH Practice	Identified	FP	TP	# Benchs
RETU	54	11	43	43
LOOP	29	4	25	25
FINAL	9	2	7	7
INVO	18	2	16	25
FORK	2	0	2	5
Total	**112**	**19**	**93**	**105**

to (1) limitations in our rules or (2) needing to understand the intent behind the benchmark creation.

Table 7 shows the number of false positives found (the 'FP' column) per bad JMH practice. Upon manual analysis of the 112 bad practice instances that were initially identified by SpotJMHBugs, we found that 19 (17%) were false positives reported by our tool. Note that the remaining 93 bad JMH practice instances could affect 105 benchmarks, as a single instance of INVO and FORK may affect multiple benchmarks. We further detail the criteria used to filter the encountered false positives.

RETU: 11 of 54 instances of RETU were considered false positives after manual inspection. In those cases, developers inserted the variable that was identified as non-consumed inside a conditional check, throwing an exception in case of an unexpected value. This can be done by a call to an assert method, or through an explicit `if` clause followed by throwing an exception. This is a path-sensitive strategy for checking the value of a variable and preventing dead code elimination, and is currently not considered by our SpotJMHBugs tool.

LOOP: In 5 of 29 cases, the accumulation of a numeric variable in a benchmark loop was considered a false positive after manual inspection. In these cases, the accumulation was an integral part of the benchmark, and not only used to consume the return of a method call.

FINAL: From the initial set of 9 FINAL cases, 2 were considered a false positive. The final primitives were used in a method unrelated to benchmarks.

INVO: From the initially reported 18 INVO bad practices, 2 could not be removed or reduced with our methodology. Both scenarios are comprised of fixture methods that are required to execute on every invocation, and run for too long (longer than 1 ms) to be included in the benchmark without offsetting the measurements. These constitute correct configuration of invocation level fixture methods, and are hence false positives.

FORK: We found no false positives in the two instances of FORK reported by SpotJMHBugs.

5.4 Results

In this section, we describe the methodology used to generate the fixed version and the results of the impact analysis for each of the studied bad JMH practices. For each bad practice, we also detail a specific case where removing the bad JMH practice had a significant impact on the performance counters of the benchmarks.

TABLE 8: The number of benchmarks that were impacted by the bad JMH practices which were removed through non-intrusive fixes. The 'Effect Size' column indicates whether the observed difference after removing the bad JMH practice had a small (S), medium (M) or large (L) effect size.

Bad JMH Practice	Project	Benchmarks		Effect Size		
		Impacted	%	S	M	L
RETU	netty	1/1	100.0			1
	gs-collections	9/37	24.3		1	8
	logging-log4j2	5/5	100.0	1		4
	Total	**15/43**	**34.8**	**1**	**1**	**13**
LOOP	netty	10/10	100.0			10
	druid	6/8	75.0		1	5
	logging-log4j2	7/7	100.0	1	2	4
	Total	**23/25**	**92.0**	**1**	**3**	**19**
FINAL	netty	1/2	50.0			1
	druid	–	–			
	logging-log4j2	4/5	80.0	4		
	Total	**5/7**	**57.1**	**4**	**0**	**1**
INVO	netty	12/12	100.0		1	11
	druid	2/3	66.7			2
	h2o-3	6/6	100.0	2	2	2
	Total	**20/21**	**95.2**	**2**	**3**	**15**
FORK	pgdbc	5/5	100.0			5

Table 8 shows the analysis of the impact of bad JMH practices for all non-intrusive fixes, and Table 9 shows the result of removing the INVO bad practices through the intrusive fix. In both tables we show how often a benchmark was impacted (the 'Impacted' column) by removing the bad JMH practice and the 'Effect Size' of the observed differences compared to the original version.

5.4.1 Impact of RETU

Fix Patches. We can ensure that unconsumed variables will not be eliminated by the JIT in two ways: (1) we can return a variable at the end of the benchmark method, or (2) we can call `Blackhole.consume()` to consume the variables. We apply the first patch to every void benchmark with a single unconsumed variable, as we consider this solution more elegant (which is relevant as we contribute a subset of fixes to the projects, as discussed in Section 6). In all other cases, we consume the variable using a Blackhole object.

Results. As shown in Table 8, we evaluate 43 benchmarks containing the bad JMH practice. In our results, **15 of 43 benchmarks (34%) were impacted by fixing the RETU bad**

TABLE 9: Benchmarks impacted by the INVO bad practice, fixed by adding the fixture code inside the benchmark (intrusive fix).

Bad JMH Practice	Project	Benchmarks		Effect Size		
		Impacted	%	S	M	L
INVO	netty	3/4	75.0			3

practices. For 13 benchmarks the difference has a large effect size. The differences for the other two impacted benchmarks have smaller effect sizes.

```
@Benchmark
public void baseline() {
    consume(bytes); // return is not used
}

private static long consume(final byte[] bytes) {
    long checksum = 0;
    for (final byte b : bytes) {
        checksum += b;
    }
    return checksum;
}
```

Listing 6: Source-code of a benchmark affected by the RETU bad practice in the `logging-log4j2` project.

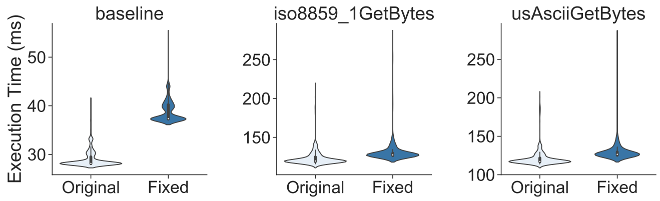

Fig. 5: Performance counters before and after fixing a RETU bad JMH practice in 3 benchmarks in the `AbstractStringLayoutEncoding` class from `logging-log4j2`. The effect size is large for the baseline benchmark, and medium and small for the other 2 benchmarks.

The highest impact after fixing a RETU bad practice came from a baseline benchmark of `logging-log4j2` presented in Listing 6. Developers called a `static` self-implemented consume method, to consume the computation, but ignored the method return. After changing the code to consume the computation, the resulting performance counters showed that the execution time increased by 32% (see Figure 5), which indicates that the developers originally failed to appropriately take JIT optimization into account. In addition, other benchmarks from the same class (e.g., `usAsciiGetBytes`) were also impacted by our fix, although with smaller effect sizes.

> 35% of the benchmarks containing the RETU bad practice were significantly impacted by the bad practice. In 30% of the cases, the impact had a large effect size.

5.4.2 *Impact of LOOP*

Fix Patches. We refactor the accumulation inside a loop into a `Blackhole.consume` method call.

Results. Table 8 shows how often an instance of the LOOP bad JMH practice impacted a benchmark. **23 of 25 benchmarks (92%) had their performance counters impacted by fixing this bad JMH practice**. For 19 of these 23 impacted benchmarks, the difference in execution time before and after removing the LOOP bad practice has a large effect size.

To illustrate, we present a benchmark class from the `druid` project, where we observed the largest effect sizes in our experiment. `LongCompressionBenchmark` has two benchmarks defined: the first benchmark reads sequentially from an array (see Listing 7) while the second randomly skips array positions. The first example can have its loop unrolled and the operations merged by JIT, artificially speeding-up the benchmark with a median of 22%, and up to 9 times in one extreme case. We present the impact of our fix of the sequential read benchmark in Figure 6 for the execution with all 20 benchmark parameters. The second benchmark cannot easily be optimized by JIT, and was not impacted by our fix (see our online appendix [17]).

```
@Benchmark
public void readContinuous(Blackhole bh) {
    ColumnarLongs columnarLongs = supplier.get();
    int count = columnarLongs.size();
    long sum = 0;
    for (int i = 0; i < count; i++) {
        sum += columnarLongs.get(i);
    }
    bh.consume(sum);
    columnarLongs.close();
}
```

Listing 7: Source-code of a benchmark in `druid` project affected by the LOOP bad practice.

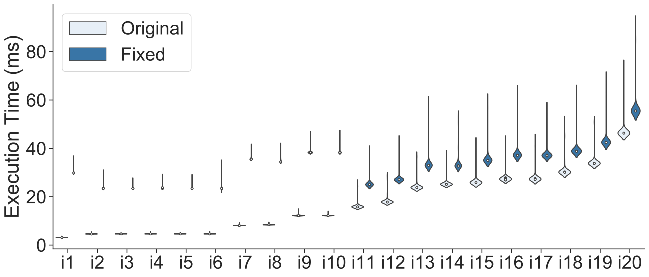

Fig. 6: Performance counters before and after the fix of an instance of the LOOP bad practice in a `druid` benchmark. The effect size of the differences is large for all 20 benchmark parameters (i1 to i20).

> 23 out of 25 benchmarks containing the LOOP bad JMH practice were impacted by the bad practice. For 19 benchmarks the impact had a large effect size.

5.4.3 *Impact of FINAL*

Fix Patches. To fix FINAL bad JMH practices, we simply removed the `final` modifier of primitive variables.

Results. As shown in Table 8, we found that **5 of 7 (71%) benchmarks had their performance counters impacted by fixing the FINAL bad JMH practice**. The effect size of the impact was large for one benchmark, and small in the other four cases. The four benchmarks with small effect size measure the execution time of logging an event in the `logging-log4j2` project. Developers used a final boolean variable to check whether to perform further logging operations as depicted in Listing 8. JVM can optimize the first check of the `if`-clause away. The impact of such an optimization is statistically noticeable, and may be relevant in some practical use cases, but the absolute impact of the bad FINAL bad JMH practice is considerably lower than for the previously discussed bad JMH practices (see Figure 7).

5 out of 7 benchmarks were impacted by the FINAL bad practice, but the impact typically has only a small effect size.

```
private final additive = true;
//Method called by a benchmark
private void logParent(final LogEvent event) {
    if (additive && parent != null) {
        parent.log(event);
    }
}
```

Listing 8: Source-code of a benchmark affected by FINAL bad practice in the `logging-log4j2` project.

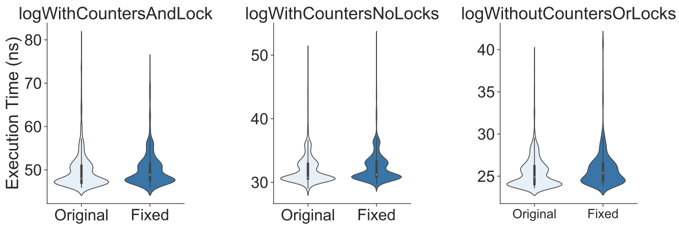

Fig. 7: Performance counters before and after the fix a FINAL bad practice on 3 benchmarks in `logging-log4j2`. The effect size is small for all 3 benchmarks.

5.4.4 Impact of INVO

Fix Patches. The process of fixing the INVO bad JMH practice requires both the analysis of the benchmark code and a preliminary set of experiments. The JMH documentation explicitly mentions the requirement of an ad-hoc evaluation of the invocation level usage.

We first determined which objects need to be created or cleaned up on every invocation. Every object that does not require this was moved to a fixture method with `Level.Iteration`. Then, we ran a set of preliminary experiments to identify fixture methods that run in less than one millisecond, which we define as short-running. In such cases, we moved the fixture method code into the relevant benchmark method, as suggested by the JMH documentation. We made sure to never refactor class fields into local variables to prevent the JVM from performing further optimizations. The JVM can identify if a local variable is accessed in a restricted scope (Escape Analysis) and avoid allocating the object in the heap (Scalar Replacement).

`SpotJMHBugs` reported 16 INVO instances, which affected 25 benchmarks in total. In 21 benchmarks, the invocation level fixture could be removed or reduced without adding code to the benchmark method (non-intrusive fix). In the remaining 4 benchmarks, we added the setup/teardown code inside the benchmark (intrusive fix).

Results of the non-intrusive fix. Table 8 shows the impact analysis for instances of the INVO bad practice that could be removed without changing the benchmark code. In our evaluation, **20 of 21 (95%) benchmarks had their performance counters impacted** by the INVO bad practice, and in 15 benchmarks the impact had a large effect size.

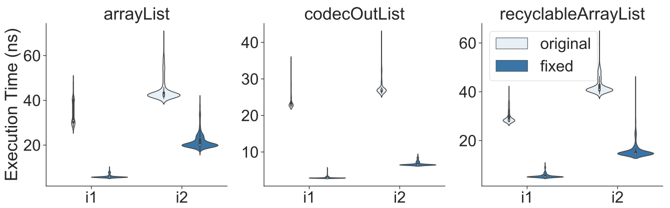

Fig. 8: Performance counters before and after the non-intrusive fix for the INVO bad practice on 3 benchmarks from `CodecOutputListBenchmark` class. The effect size is large for all benchmarks, each evaluated with 2 sets of parameters (i1 and i2).

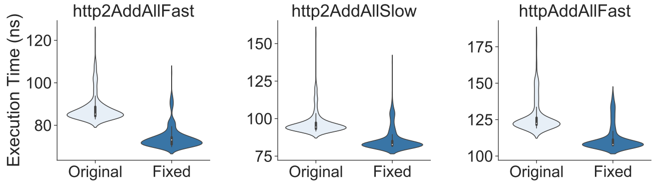

Fig. 9: Performance counters before and after the fix for 3 benchmarks that had their INVO bad practice removed through an intrusive fix.

For instance, the `CodecOutputListBenchmark` class from the `netty` project had its fixture methods unnecessarily configured to invocation level. The created objects were not modified during the benchmark execution and could instead be instantiated on every iteration. After our fix, every benchmark executed on average three times faster (see Figure 8).

Results of the intrusive fix. As shown in Table 9, **3 of 4 benchmarks had significantly faster performance counters after moving the setup code inside of the benchmark**. This speedup indicates that the JMH overhead was considerably higher than the time spent in the setup phase.

Figure 9 shows the comparison of the performance counters in three of the benchmarks. The benchmarks were up to 20% faster after our fix. More importantly, such benchmarks were defined in a single class `HeadersBenchmark`, which contained nine other benchmarks that were indirectly affected by the invocation level fixture. Therefore, our fix eliminated the JMH invocation overhead from the remaining nine benchmarks as well.

23 out of the 25 benchmarks that used the INVO bad practice were misconfigured which impacted their performance counters. Including the setup code in the benchmark itself, actually accelerated benchmark execution in 3 of 4 cases.

5.4.5 Impact of FORK

Fix Patches. Benchmarks configured with zero forks can be reconfigured by modifying the annotation `@Fork` or by overriding the fork parameter of JMH directly when starting the benchmark through the command line. We opt for the

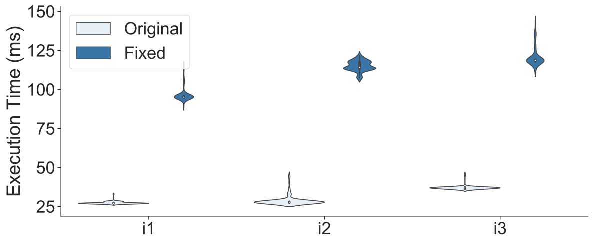

Fig. 10: Performance counters before and after fixing the FORK bad practice in the `AddingPaddingZeroes` benchmark of the `pgjdbc` project. The impact of removing the FORK bad practice had a large effect size in all three instances.

first approach in our study (as it allows us to fix the bad practice directly in the source-code).

Results. We evaluated two instances of the FORK bad JMH practice, which configured the fork parameter for five benchmarks (see Table 8). **All five benchmarks were impacted by the FORK bad JMH practice**. In all cases, the difference in execution time after removing the FORK bad practice had a large effect size.

In our evaluation, the `AddingPaddingZeroes` class from `pgjdbc` showed the highest impact after our fix, with differences of up to three times (see Figure 10).

> All five benchmarks configured with zero forks had their performance counters largely impacted by removing the FORK bad practice.

6 Submitting Pull Requests

After evaluating the impact of bad JMH practices, we submitted a number of pull requests to the maintainers of a subset of the studied projects. In this study, the purpose of submitting the pull requests are twofold: (1) To validate our bad JMH practice detection and impact assessment with the benchmark creators, and (2) to contribute to open-source projects by fixing potentially misleading benchmark implementations.

We restricted our pull requests to the cases where our analysis has shown that the effect size of removing the bad JMH practice was large. This limited our efforts to benchmarks where misleading results could have consequences on the project's performance. We also refrained from submitting a pull request for one benchmark from the `pgjdbc` project, because this benchmark did not evaluate project code, but only compared the performance of bitwise operations instead. Table 10 shows a description of the pull requests that we submitted to each project, per bad JMH practice. In total, our pull requests applied changes to 57 benchmarks across 6 open-source projects.

Accepted pull requests. 6 of 7 issued pull requests were well received and accepted by developers and maintainers of the studied projects. In these pull requests, developers agreed on merging the recommended patch into the main branch. In one case, a developer mentioned having previously identified such unsafe loops, but never had the chance to fix it [15]. In another case [14], the developers agreed with the fix but asked to remove the benchmark altogether instead since it was not in use anymore. The `gs-collections` project has the largest benchmark suite, with almost every class making a good use of Blackhole and variable sinking options. Still, we found 23 cases of the harmful RETU bad JMH practice, fixed by our patch [10].

Rejected pull requests. The only rejected pull request was the patch that reconfigured the `@Fork` parameter from zero to one. The developers acknowledged that configuring a benchmark with zero forks could lead to misleading results, but justify that such configuration was only used to debug the benchmark in the IDE. Furthermore, the pull request was not accepted and merged into the `pgjdbc` project for two reasons: (1) The benchmarks were not part of the continuous integration process and are executed on a case-by-case basis, thus would not impact the quality of the main product and (2) the developers intend to remove JMH from the project due to licensing issues.

7 Discussion

We now discuss the implications of our results for software developers and researchers, as well as threats to the validity of our study.

7.1 Implications

Our results show that existing JMH microbenchmarks even in prominent open source software systems contain bad practices that impact the benchmark results.

Although well-documented in research [20] as well as in the JMH documentation, many open source developers still appear to be struggling to correctly account for the many intricacies of benchmarking Java applications. The fact that our pull requests containing fixed benchmarks have been merged back in 6 of 7 cases indicates that developers generally care about the bad JMH practices we have presented (i.e., they appear to not be cases of conscious trade-offs), but often fail to avoid them in practice. This should be addressed in the following orthogonal ways:

Improve developer training and documentation. Based on our results, we speculate that the information that developers currently have is not effective in guiding them towards rigorous benchmarking solutions. One reason may be that without detailed knowledge of the inner workings of

TABLE 10: Pull requests submitted to developers of studied open-source projects. Column Ref links the Github pull-request and Issue description, while column Status shows how developers received our patch.

Ref	Project	Bad JMH Practice	# of Benchmarks	Status
[10]	gs-collections	RETU	23	Accepted
[12]	logging-log4j2	RETU	5	Accepted
[15]	druid	LOOP	4	Accepted
[14]	druid	INVO	2	Accepted
[13]	h2o-3	INVO	2	Accepted
[9]	netty	INVO	16	Accepted
[11]	pgjdbc	FORK	5	Rejected
		Total	**57**	

the JVM and just-in-time compilation, many bad JMH practices may appear obscure and unimportant to developers. It is possible that developers are aware that their code contains bad JMH practices according to the documentation, but fail to see how these bad practices impact their own projects. Improving this understanding is difficult, but may require more explicitly and extensively discussing the effects of bad practices (for example in the JMH documentation) rather than only listing what they are. However, ultimately, better developer training with regards to performance engineering for Java applications will be required.

Improve developer tooling. In addition to training, better tooling should be provided to help developers avoiding bad JMH practices. Our own tool, `SpotJMHBugs`, is a good starting point. As a SpotBugs plugin, `SpotJMHBugs` is easy to integrate into standard IDEs and can be used already during development to point out bad JMH practices. Another angle may be to extend JMH itself to, for instance, analyze the configuration input and benchmark code before the execution of the benchmarks. Through this analysis, JMH could produce warning messages when it discovers configurations or benchmark code that appears to contain bad JMH practices (similar to what it already does for the FORK bad practice). A significant advantage of this approach is that at runtime, JMH has access to a much richer set of metrics than our static analysis approach to determine whether any given execution is likely to lead to trustworthy benchmark results. For instance, JMH knows how many warmup iterations, benchmark iterations, and forks are actually being executed, and can use statistical power analysis to evaluate if this configuration is trustworthy given the benchmark value dispersion it observes.

Studying approaches for automatic benchmark repair. In our work, we have manually fixed a number of instances of bad JMH practices. However, in doing so, it has become evident that for a subset of bad practices, (e.g. LOOP, FINAL and RETU) fixes actually only require a fairly static and simple transformation of the code. Future studies should investigate automatic benchmark repair, which could help to fix bad JMH practice instances without direct developer involvement. Such a tool would have a large positive impact on performance testing practices, as it (1) could be employed to provide widespread fixes of the many instances of bad JMH practices we have identified in our study, and (2) act as another tool for developer training. That is, such an automatic benchmark repair tool could educate developers in how a rigorous implementation of their benchmark would look.

7.2 Threats to Validity

The *external validity* concerns the generalizability of our work. One threat is that we selected open source projects from GitHub only. Future studies are necessary to identify the frequency and impact of bad JMH practices in other types of projects, such as industrial projects.

In addition, our study focused on JMH microbenchmarks. While JMH has become one of the most popular microbenchmarking frameworks for JVM-based languages (such as Java, Scala and Clojure), future studies are necessary to investigate whether our findings hold for other microbenchmarking frameworks in Java and other programming languages. In addition, future studies should investigate the impact of these bad practices when using non-default JIT compilers, such as the Graal compiler.

The *internal validity* concerns the confidence that we have in our findings. One threat is that we consider the fixes that are suggested by the JMH documentation as the current 'industry-standard' for addressing bad JMH practices. It is possible that these JMH-proposed fixes themselves impact the performance of the benchmark. Regardless, our tool can detect the bad JMH practices; if in the future a different 'industry-standard' fix arises, future studies should re-evaluate the impact of the bad JMH practices that our tool can detect.

Another threat is that we use the thresholds that were proposed by Romano et al. [38] for Cliff's Delta effect size to quantify the impact of a bad JMH practice on the benchmark results. Some projects may require different thresholds to meet their performance requirements. Future studies should investigate these differences in performance requirements.

Because our detection rules are heuristic-based, manually verified instances of bad practices investigated in this study could represent only a subset of all existing cases in the studied projects. If a bad practice manifests itself in a different way our tool will not detect it. Future studies should investigate how our detection rules can be extended and improved.

The *construct validity* concerns the construction of our experiments. One threat is that our tool for identifying bad JMH practices is based on heuristics, and is therefore inherently susceptible to false positives. The major challenge is that developer intent is an important factor in deciding whether an identified instance of a bad JMH practice is indeed a bad practice. For example, a developer might actually want to benchmark the impact of JVM optimization on variable accumulation inside a loop. Hence, our tool should be used as a guideline by developers to identify potential instances of bad JMH practices.

In addition, our rules do not cover all possible cases of undesired JVM optimizations. JVM also uses runtime analysis to perform optimizations, e.g., it might eliminate dead code after inlining a method call during the benchmark execution. Because our tool uses static analysis only, it cannot detect instances of bad JMH practices that depend on runtime information.

8 Related Work

Below we discuss the related work on (1) microbenchmarking and performance unit testing, (2) errors in performance evaluation, and (3) methodologies for robust performance analysis.

Microbenchmarking and performance unit tests

Microbenchmarking and performance unit testing are two related approaches to assess the performance of program code. Microbenchmarks evaluate the performance of a (typically small) code segment, such as a single method [1], [2], [30], [37]. The outcome of a microbenchmarking run is a set of one or multiple performance counters. A developer or quality engineer uses these performance counters to compare different implementation alternatives, or detect slowdowns (i.e., by statistically comparing the performance counters produced in a new version of the product with counters produced by previous releases). Performance unit tests similarly operate on small code segments, but they entail concrete target values, akin to asserts in unit testing for functional defects [18], [39]. In this paper, we focus on microbenchmarking. However, given the conceptual proximity of the concepts, much of our work can arguably also be applied to performance unit testing.

As discussed in this paper, a major challenge of correct microbenchmarking is to prevent misleading results due to compiler and JVM optimizations, including constant folding, loop unrolling and method inlining [2]. Frameworks such as JMH [1] are designed with an intricate knowledge of JVM optimizations and can help benchmark designers in avoiding related pitfalls. For example, JMH provides the `Blackhole` class, which consumes return values and circumvents dead code elimination. However, as demonstrated in this work, JMH alone cannot preclude spurious evaluation results if used incorrectly.

Especially close to our work are approaches and methodologies which ensure the correctness of microbenchmarks or performance unit tests [2], [25], [26], [37]. Rodriguez-Cancio et al. [37] proposed a combination of static and dynamic analysis and code generation to synthesize microbenchmarks evaluating code segments extracted from large applications. Their AutoJMH tool generates payloads which prevent dead code elimination and constant folding, optimizations which can lead to common mistakes in microbenchmarks. In our work, we consider a larger set of problems related to JMH, including more complex ones, e.g., a loop inside a benchmark. We also focus on detecting such bad practices via a static analysis tool and conduct empirical studies on the prevalence of bad practices and the impact of their fixes.

Further work focusing on the correctness of microbenchmarking discusses the issues that can hinder the experiment or mislead the evaluation in Java [25], or describe how JMH can be used to avoid typical pitfalls [2], [26].

Other contributions in this area propose methods for generating benchmarks for scenarios in which JVM optimizations are not harmful [29], [36]. Kuperberg et al. [29] proposed an automated solution for benchmarking any set of APIs, e.g., the Java Platform API. Contrary to the above-mentioned approaches, this solution specifically induces the JIT optimizations to "obtain realistic benchmarking results". Pradel et al. [36] introduced SpeedGun, a technique which can automatically discover performance regressions in thread-safe classes. Also in this scenario, JVM optimizations do not affect the results (assuming that both code versions are optimized).

Performance evaluation errors

A large amount of work studies reasons for incorrect or biased performance evaluation results. A commonly identified problem is non-determinism of individual executions caused by the complexity of the runtime environments, in particular the Java JVM [5]. Another source of performance variations is the OS Jitter phenomenon [34], e.g., the impact of the thread affinity values and settings on the execution time.

While the impact of non-determinism on the results can be addressed by repetition of experiments and rigid statistical procedures [6], [20], a more serious problem is a measurement bias caused by presumably innocuous factors [33]. Such so-called *hidden factors* can take various forms, such as link order of code segments/libraries, or UNIX environment size [7].

Mytkowicz et al. [33] conducted one of the first comprehensive studies on hidden factors and their causes, and proposed a method for their detection (via causal analysis) and for avoiding them (setup randomization). Curtsinger et al. [18] addressed the impact of the layouts of code, stack, and heap objects at runtime on the performance. They developed a tool for randomizing the memory layout, which is combined with statistical techniques such as ANOVA for sound performance optimization (in particular related to the impact of compiler optimization levels).

Other studies reported different types of hidden factors. Kalibera et al. [27] demonstrated the impact of the initial state of the system on the performance results. Harji et al. [24] showed that the Linux kernel has had multiple serious performance-related regressions, resulting in performance variation of as much as 45% between two subsequent versions. Consequently, the choice of the kernel version might have significant impact on the benchmark results. In an earlier version of the JVM, changing the names of symbols significantly affected the cache miss count and thus performance of applications [23]. In another study of Java performance, the authors reported that simply restarting the virtual machine could cause performance variations as high as 3% [21].

Our work focuses on the incorrect usages of the JMH framework which can result in measurement biases. The reasons for such errors are in most cases unwanted JVM optimizations. Contrary to the approaches to avoid hidden factors such as setup randomization, unwanted optimizations can be eliminated via thorough understanding of the optimization process and correct benchmarking code. Nevertheless, detection of measurement biases due to novel optimizations or JVM internals remains an open challenge, similarly to generic detection approaches for hidden factors.

Methodologies for robust performance analysis

Prior work proposed methodologies, frameworks, and specific techniques to ensure correctness and robustness of performance evaluation results in face of non-determinism inherent in complex computer systems. In most cases, this work is complemented by empirical studies or literature analyses which demonstrate the problems.

Georges et al. [20] focus on the data analysis aspects of the performance evaluation of Java programs. Their study of reported Java performance results available at that time uncovered a need for a statistically rigorous evaluation methodology in face of the non-determinism of the Java runtime. They proposed and evaluated several statistical measures to address this problem while considering practical aspects, such as best practices for quantifying startup and steady-state performance. The conclusions of this work were partially extended by Bulej et al. [6], where the authors showed the pitfalls of applying basic statistical methods to data from a real performance benchmark (SPECjbb2015).

Blackburn et al. [5] showed how the complexity and large number of degrees of freedom of the Java runtime system can lead to misleading performance results. They argue that benchmark designers must use relevant workloads, principled experimental design, and rigorous analysis to produce meaningful results, and illustrate their reasoning on the design choices for the DaCapo benchmark [4].

Other work focused on specific aspects of performance evaluation. Alexander et al. [3] propose using nonlinear time series analysis techniques to capture the complex dynamics of computer performance data. Kalibera and Jones [28] addressed the trade-off between cost of experiments (in terms of number of repetitions) and statistical validity of the results. They introduced a mathematical model for adjusting the number of repetitions to the level of uncertainty and evaluate it using the DaCapo and SPEC CPU benchmarks. De Oliveira et al. [19] proposed DataMill, a distributed infrastructure for rigorous performance evaluation with particular focus on eliminating impact of hidden factors via their automated variation. Moreno and Fischmeister [32] discussed a simple technique to eliminate the systematic error introduced by the `get_current_time()` system call, a relevant problem in microbenchmarking.

Our work focuses on the particular domain of Java microbenchmarks, which is a relatively new class of evaluation methods. We quantify the prevalence and impact of misuse of the JMH benchmarks, attempting to raise the awareness of correct microbenchmarking. We also provide a set of recommendations for practitioners and researchers as a "soft" methodology for robust performance analysis.

9 Conclusions

In this paper, we studied bad practices of writing microbenchmarks using the JMH framework. We presented five bad JMH practices related to not consuming products of computation, using a loop to accumulate computations, using `final` primitives that are prone to constant folding, incorrect usage of test fixtures, and incorrect usage of JMH forks. We showed that these bad JMH practices are indeed prevalent in Java-based open source systems, and found that fixing them often leads to performance counters that are statistically significantly different from the original version with a large effect size. This indicates that bad JMH practices are indeed often severely impacting the outcome of benchmarks, as they lead to benchmark results that substantially deviate from the correct measurements. We submitted pull requests containing fixed benchmarks to developers of impacted open source projects to validate whether developers agreed with our assessment and analysis. 6 of 7 submitted pull requests were accepted and merged quickly (one was rejected as the developers plan to remove JMH from the project due to licensing issues), indicating that developers indeed confirmed the identified issues after being presented with the results of our study.

Our study results indicate that many open source developers still struggle to account for the many intricacies of benchmarking Java applications. Consequently, we suggest that we need (besides improving developer training as well as the JMH documentation) better tooling to guide developers towards rigorous benchmark implementations. As part of our study, we have developed `SpotJMHBugs`, a static analysis tool to identify bad JMH practices. As a plugin to SpotBugs, `SpotJMHBugs` is easy to integrate into standard Java IDEs. However, even more important may be to improve JMH itself. Such improvement could for example consist in generating warning messages when JMH discovers configurations or benchmark code that appears to contain bad JMH practices. Further, we suggest that our work can be used as a starting point for studying approaches for automatic benchmark repair.

Acknowledgments

Philipp Leitner acknowledges the financial support provided by the Swedish Research Council VR under grant number 2018-04127 (Developer-Targeted Performance Engineering for Immersed Release and Software Engineers).

References

[1] OpenJDK: Java Microbenchmark Harness. https://openjdk.java.net/projects/code-tools/jmh/. [Online; accessed 19-December-2018].

[2] Aleksey Shipilëv. Java Microbenchmark Harness - (the lesser of two evils). https://shipilev.net/talks/devoxx-Nov2013-benchmarking.pdf, 2013. [Online; accessed 19-December-2018].

[3] Z. Alexander, T. Mytkowicz, A. Diwan, and E. Bradley. Measurement and Dynamical Analysis of Computer Performance Data. In *Proceedings of the 9th International Conference on Advances in Intelligent Data Analysis*, pages 18–29. Springer-Verlag, 2010.

[4] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, pages 169–190. ACM, 2006.

[5] S. M. Blackburn, K. S. McKinley, R. Garner, C. Hoffmann, A. M. Khan, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanovik, T. VanDrunen, D. von Dincklage, and B. Wiedermann. Wake Up and Smell the Coffee: Evaluation Methodology for the 21st Century. *Commun. ACM*, 51(8):83–89, Aug. 2008.

[6] L. Bulej, V. Horký, and P. Tůma. Do We Teach Useful Statistics for Performance Evaluation? In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, pages 185–189. ACM, 2017.

[7] Y. Chen, Y. Huang, L. Eeckhout, G. Fursin, L. Peng, O. Temam, and C. Wu. Evaluating Iterative Optimization Across 1000 Datasets. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 448–459. ACM, 2010.

[8] N. Cliff. Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological Bulletin*, 114:494–509, 11 1993.

[9] D. Costa. Avoiding the usage of Jmh Invocation level on setup/teardown. https://github.com/netty/netty/pull/8005, 2018. [Online; accessed 19-December-2018].

[10] D. Costa. Fix 627 - Consume every object created in a benchmark to avoid DCE. https://github.com/eclipse/eclipse-collections/pull/628, 2018. [Online; accessed 19-December-2018].

[11] D. Costa. fix: configure benchmark forks to >= 1. https://github.com/pgjdbc/pgjdbc/pull/1214, 2018. [Online; accessed 19-December-2018].

[12] D. Costa. LOG4J2-2478 Return the computed variables on each benchmark to avoid DCE. https://github.com/apache/logging-log4j2/pull/219, 2018. [Online; accessed 19-December-2018].

[13] D. Costa. PUBDEV-6081: Reducing the Invocation JMH level setup/teardown to only the training. https://github.com/h2oai/h2o-3/pull/3070, 2018. [Online; accessed 19-December-2018].

[14] D. Costa. Reducing the usage of Jmh Invocation level on setup. https://github.com/apache/incubator-druid/pull/6459, 2018. [Online; accessed 19-December-2018].

[15] D. Costa. Use Blackhole objects to sink the method return in a benchmak loop (JMH). https://github.com/apache/incubator-druid/pull/5908, 2018. [Online; accessed 19-December-2018].

[16] D. Costa, C.-P. Bezemer, P. Leitner, and A. Andrzejak. Spotjmhbugs. https://github.com/DiegoEliasCosta/spotjmhbugs, 2018. [Online; accessed 21-December-2018].

[17] D. Costa, C.-P. Bezemer, P. Leitner, and A. Andrzejak. Studying bad practices in JMH benchmarks. https://github.com/DiegoEliasCosta/badJMHpractices-study, 2018. [Online; accessed 21-December-2018].

[18] C. Curtsinger and E. D. Berger. STABILIZER: Statistically Sound Performance Evaluation. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 219–228. ACM, 2013.

[19] A. B. de Oliveira, J.-C. Petkovich, T. Reidemeister, and S. Fischmeister. DataMill: Rigorous Performance Evaluation Made Easy. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, pages 137–148. ACM, 2013.

[20] A. Georges, D. Buytaert, and L. Eeckhout. Statistically Rigorous Java Performance Evaluation. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, pages 57–76. ACM, 2007.

[21] J. Y. Gil, K. Lenz, and Y. Shimron. A Microbenchmark Case Study and Lessons Learned. In *Proceedings of the Compilation of the Co-located Workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, & VMIL'11*, pages 297–308. ACM, 2011.

[22] V. Green. Impact of slow page load time on website performance. http://bit.ly/2SGmbM4, 2016. [Online; accessed 19-December-2018].

[23] D. Gu, C. Verbrugge, and E. Gagnon. Code Layout as a Source of Noise in JVM Performance. *Stud. Inform. Univ.*, 4(1):83–99, 2005.

[24] A. S. Harji, P. A. Buhr, and T. Brecht. Our Troubles with Linux and Why You Should Care. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, pages 2:1–2:5. ACM, 2011.

[25] V. Horký, P. Libič, A. Steinhauser, and P. Tůma. DOs and DON'Ts of Conducting Performance Measurements in Java. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, pages 337–340. ACM, 2015.

[26] Julien Ponge. Avoiding Benchmarking Pitfalls on the JVM. *Java Magazine*, Aug. 2014.

[27] T. Kalibera, L. Bulej, and P. Tuma. Benchmark precision and random initial state. In *in Proceedings of the 2005 International Symposium on Performance Evaluation of Computer and Telecommunications Systems*, pages 853–862. SCS, 2005.

[28] T. Kalibera and R. Jones. Rigorous Benchmarking in Reasonable Time. In *Proceedings of the 2013 International Symposium on Memory Management*, pages 63–74. ACM, 2013.

[29] M. Kuperberg, F. Omri, and R. Reussner. Automated Benchmarking of Java APIs. Paderborn, Germany, 2010.

[30] C. Laaber and P. Leitner. An Evaluation of Open-source Software Microbenchmark Suites for Continuous Performance Assessment. In *Proceedings of the 15th International Conference on Mining Software Repositories*, pages 119–130. ACM, 2018.

[31] P. Leitner and C.-P. Bezemer. An exploratory study of the state of practice of performance testing in Java-based open source projects. In *Proceedings of the 8th ACM/SPEC of the International Conference on Performance Engineering*, pages 373–384. ACM, 2017.

[32] C. Moreno and S. Fischmeister. Accurate Measurement of Small Execution Times—Getting Around Measurement Errors. *IEEE Embed. Syst. Lett.*, 9(1):17–20, Mar. 2017.

[33] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Producing Wrong Data Without Doing Anything Obviously Wrong! In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 265–276. ACM, 2009.

[34] P. E. Nogueira, R. Matias, Jr., and E. Vicente. An Experimental Study on Execution Time Variation in Computer Experiments. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, pages 1529–1534. ACM, 2014.

[35] Oracle America Inc. JMH Samples. http://hg.openjdk.java.net/code-tools/jmh/file/tip/jmh-samples/src/main/java/org/openjdk/jmh/samples/, 2014. [Online; accessed 19-December-2018].

[36] M. Pradel, M. Huggler, and T. R. Gross. Performance Regression Testing of Concurrent Classes. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 13–25. ACM, 2014.

[37] M. Rodriguez-Cancio, B. Combemale, and B. Baudry. Automatic Microbenchmark Generation to Prevent Dead Code Elimination and Constant Folding. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 132–143. ACM, 2016.

[38] J. Romano, J. D. Kromrey, J. Coraggio, J. Skowronek, and L. Devine. Exploring methods for evaluating group differences on the NSSE and other surveys: Are the t-test and Cohen's d indices the most appropriate choices. In *Annual meeting of the Southern Association for Institutional Research*, 2006.

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

[40] The JUnit Team. JUnit 5. https://junit.org/junit5/. [Online; accessed 4-April-2019].

[41] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, 13(2):181–210, Apr. 1991.

[42] D. Weise, R. F. Crew, M. Ernst, and B. Steensgaard. Value dependence graphs: Representation without taxation. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 297–310. ACM, 1994.

[43] E. J. Weyuker and F. I. Vokolos. Experience with performance testing of software systems: Issues, an approach, and case study. *IEEE Trans. Softw. Eng.*, 26(12):1147–1156, Dec. 2000.

[44] F. Wilcoxon. *Individual Comparisons by Ranking Methods*, pages 196–202. Springer New York, 1992.

Diego Costa is a PhD Student at Heidelberg University, working at the Institute of Computer Science in the Parallel and Distributed Systems Group. He received his Bachelor and MSc degree at the Federal University of Uberlandia in Brazil, and worked in industry as a software developer for 3 years, developing high-performance trading solutions for the Stock Market. His research interests cover the intersection of software engineering and performance engineering related topics, including adaptive frameworks and data structures, performance testing, mining software repositories and memory leak detection.

Cor-Paul Bezemer is an assistant professor in the Electrical and Computering Engineering department at the University of Alberta in Canada. He heads the Analytics of Software, GAmes And Repository Data (ASGAARD) lab. His research interests cover a wide variety of software engineering and performance engineering-related topics. His work has been published at premier software engineering venues such as the TSE and EMSE journals and the ESEC-FSE, ICSME and ICPE conferences. He is one of the vice-chairs of the SPEC research group on DevOps Performance. Before moving to Canada, he studied at Delft University of Technology in the Netherlands, where he received his BSc (2007), MSc (2009) and PhD (2014) degree in Computer Science. More about Cor-Paul and the ASGAARD lab can be found at http://asgaard.ece.ualberta.ca/

Philipp Leitner is an assistant professor in the software engineering division of Chalmers University of Technology and the University of Gothenburg, Sweden. He leads the Internet Computing and Emerging Technologies lab. Philipp received a PhD from Vienna University of Technology in 2011. He has published around 100 papers in the area of software engineering for service-, cloud-, and web-based systems, with a particular focus on performance testing and improvement.

Artur Andrzejak has received a PhD degree in computer science from ETH Zurich in 2000 and a habilitation degree from FU Berlin in 2009. He was a postdoctoral researcher at the HP Labs Palo Alto from 2001 to 2002 and a researcher at ZIB Berlin from 2003 to 2010. He was leading the CoreGRID Institute on System Architecture (2004 to 2006) and acted as a Deputy Head of Data Mining Department at I2R Singapore in 2010. Since 2010 he is a W3-professor at University of Heidelberg and leads there the Parallel and Distributed Systems group. His research interests include reliability of complex software systems, scalable data analysis, and cloud computing.