

# Automated Generation, Evaluation, and Enhancement of JMH Microbenchmark Suites from Unit Tests

by

Mostafa Jangali

A thesis  
presented to the University of Waterloo  
in fulfillment of the  
thesis requirement for the degree of  
Master of Applied Science  
in  
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2024

© Mostafa Jangali 2024

### **Author's Declaration**

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

## Statement of Contributions

In all chapters and related publications of the thesis, my contributions include proposing the initial research idea, investigating background knowledge and related literature, proposing and implementing research methods, conducting experiments and analyzing experimental results, and writing and polishing the manuscript. My co-authors supported me in refining the initial research idea, providing feedback on research methods and experimental results, and offering advice for polishing the manuscript. Earlier versions of the work in the thesis were published as listed below:

- M. Jangali, Y. Tang, N. Alexandersson, P. Leitner, J. Yang and W. Shang, "Automated Generation and Evaluation of JMH Microbenchmark Suites From Unit Tests," in *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 1704-1725, 1 April 2023, doi: 10.1109/TSE.2022.3188005. keywords: Benchmark testing;Computer bugs;Manuals;Throughput;Java;Codes;Time measurement;Performance;performance testing;performance microbenchmarking;JMH;performance mutation testing

## Abstract

Ensuring the performance of software systems is a cornerstone of modern software engineering, directly influencing user satisfaction and reliability. Despite its critical role, performance testing remains resource-intensive and difficult to scale, particularly in large projects, due to the complexity of microbenchmark creation and execution. Microbenchmarking frameworks like the Java Microbenchmark Harness (JMH) offer precise performance insights but require significant expertise, limiting their adoption. This thesis addresses these challenges by introducing *ju2jmh*, a novel framework that automates the transformation of JUnit tests into JMH microbenchmarks, bridging the gap between functional and performance testing.

The contributions of this thesis are threefold. First, *ju2jmh* automates the generation of high-quality JMH microbenchmarks from widely used JUnit test suites, enabling developers to adopt performance microbenchmarking with minimal manual effort. Results demonstrate that the generated microbenchmarks exhibit stability comparable to manually crafted ones and effectively detect real-world performance bugs. Second, the Performance Mutation Testing (PMT) framework is developed to systematically evaluate the robustness of microbenchmarks in detecting artificial performance bugs, achieving competitive mutation scores. Third, a clustering approach is proposed to optimize the execution of microbenchmarks by grouping functionally similar tests based on code coverage information. This strategy reduces execution time by 81.2% to 86.2% across three large-scale projects while preserving accuracy and reliability.

Evaluated on three diverse open-source Java projects, the proposed solutions address stability, detection capabilities, and scalability challenges in performance testing workflows. The findings highlight the potential of *ju2jmh* and its associated methodologies to transform performance microbenchmarking practices, providing developers with practical tools to integrate reliable and efficient performance testing into modern software development pipelines. These advancements pave the way for future research into extending automated performance testing across different programming languages and development ecosystems.

## Acknowledgements

I would like to express my heartfelt gratitude to my supervisor, Dr. Weiyi Shang, for his invaluable guidance, encouragement, and support throughout this journey. His unwavering belief in my abilities and his insightful advice have been instrumental in shaping my research and personal growth. I am deeply honored to have had the opportunity to learn from his expertise and dedication.

I am also profoundly grateful to Dr. Diego Elias Costa, Dr. Yiming Tang and Dr. Yao Kundi for their invaluable contributions to my work. Their insightful feedback, thoughtful discussions, and expertise have significantly enriched this research. Their guidance and encouragement have been a constant source of inspiration, and I am deeply appreciative of their time and support.

Finally, I would like to extend my sincere thanks to my family for their unconditional love, encouragement, and sacrifices. Their steadfast support and belief in me have been my greatest source of strength and motivation. Without their understanding and patience, this achievement would not have been possible. This work is as much a testament to their support as it is to my efforts.

## **Dedication**

To my parents, whose love and sacrifices have been my greatest source of strength, and to my supervisor, Dr. Weiyi Shang, whose guidance and support have shaped my journey.

# Table of Contents

Author’s Declaration	ii
Statement of Contributions	iii
Abstract	iv
Acknowledgements	v
Dedication	vi
List of Figures	x
List of Tables	xi
<b>1 Introduction</b>	<b>1</b>
1.1 Purpose of the study . . . . .	2
1.2 Thesis statement . . . . .	3
1.3 Contributions . . . . .	4
<b>2 Background</b>	<b>5</b>
2.1 Performance Microbenchmarking in Java . . . . .	5
2.2 Java Microbenchmarking Harness (JMH) . . . . .	6
2.3 Mutation Testing . . . . .	7
2.4 Detecting Real-world Performance Bugs . . . . .	7
<b>3 Related Works</b>	<b>9</b>
3.1 Performance Microbenchmarking . . . . .	9
3.2 Empirical Studies on Performance Bugs . . . . .	10

3.3	Assessing the Quality of Performance Tests . . . . .	10
3.4	Optimizing Microbenchmark Execution . . . . .	11
<b>4</b>	<b>Study Overview</b>	<b>12</b>
4.1	Overview of the Study Workflow . . . . .	12
4.2	Finding Tests and Preparing Environment . . . . .	12
4.2.1	Human-Written JMH Microbenchmarks . . . . .	13
4.2.2	JUnit Test Suites . . . . .	13
4.2.3	AutoJMH Microbenchmarks . . . . .	14
4.2.4	Comparing AutoJMH and <i>ju2jmh</i> . . . . .	14
4.3	Generating Microbenchmarks Using <i>ju2jmh</i> . . . . .	15
4.3.1	Designing Mutation Operators for PMT . . . . .	18
4.3.2	Generating and Validating Performance Mutants . . . . .	21
4.3.3	Calculating Mutation Score . . . . .	21
4.3.4	Preliminary Analysis of PMT . . . . .	22
4.4	Clustering Approach for Optimization . . . . .	24
4.5	Summary . . . . .	26
<b>5</b>	<b>Study Setup</b>	<b>28</b>
5.1	Study Subjects . . . . .	28
5.2	Experiment Settings . . . . .	29
5.3	Execution environment . . . . .	31
<b>6</b>	<b>Results</b>	<b>32</b>
6.1	RQ1: How stable are automatically generated performance microbenchmarks? . . . . .	32
6.1.1	Motivation . . . . .	32
6.1.2	Approach . . . . .	33
6.1.3	Results . . . . .	33
6.1.4	Conclusion . . . . .	37
6.2	RQ2: Can performance microbenchmarks detect artificial performance bugs from mutation testing? . . . . .	38
6.2.1	Motivation . . . . .	38
6.2.2	Approach . . . . .	38
6.2.3	Results . . . . .	39
6.2.4	Conclusion . . . . .	43
6.3	RQ3: Can performance microbenchmarks detect real-world performance bugs? . . . . .	43
6.3.1	Motivation . . . . .	43

6.3.2	Approach . . . . .	44
6.3.3	Results . . . . .	45
6.3.4	Conclusion . . . . .	46
6.4	RQ4: What are the major factors affecting a microbenchmark’s ability to detect performance bugs? . . . . .	46
6.4.1	Motivation . . . . .	46
6.4.2	Approach . . . . .	47
6.4.3	Results . . . . .	48
6.4.4	Conclusion . . . . .	51
6.5	RQ5: How effective is the clustering strategy for performance microbenchmarking? . . . . .	52
6.5.1	Motivation . . . . .	52
6.5.2	Approach . . . . .	53
6.5.3	Results . . . . .	53
6.5.4	Conclusion . . . . .	55
<b>7</b>	<b>Conclusion</b> . . . . .	<b>56</b>
7.1	Summary of Results . . . . .	56
7.2	Conclusion . . . . .	57
7.3	Future Work . . . . .	57
7.4	Threats to validity . . . . .	58
7.4.1	External Validity . . . . .	58
7.4.2	Internal Validity . . . . .	59
	<b>References</b> . . . . .	<b>60</b>

# List of Figures

2.1	An example of a JMH benchmark class from the <a href="#">Eclipse-collections</a> project (adapted from [35]). . . . .	6
4.1	Workflow of our methodology for generating and evaluating performance microbenchmarks. . . . .	13
4.2	Overview of the <i>ju2jmh</i> approach and its inputs and outputs . . . . .	15
4.3	A real-life generated <i>ju2jmh</i> benchmark from <a href="#">Eclipse-collections</a> , and how it actually performs JMH rules . . . . .	16
4.4	Overview of our Performance Mutation Testing framework. . . . .	21
4.5	Preliminary analysis of the five mutation operators. Each of paired boxes contains obtained 100 data points for two cases of original source code executions and mutant executions, from each of five designed JMH benchmarks, across increasing <code>hitting_ratio</code> . Each box contains 100 data points obtained from a relevant benchmark. Black boxes represent data from original source code executions, and white boxes represent data from mutant executions. . . . .	23
6.1	Distribution of the variability, using RSD (in %) . . . . .	34
6.2	The box-plots of calculated RCIW for all benchmarks of a testing framework, ranging the iteration number $i$ , $i \in \{2, 3, \dots, 15\}$ . . . . .	37
6.3	The calculated $U_{RCI}$ for each of 21 <i>ju2jmh</i> benchmarks and 21 JUnit tests, against the mutant and the real bug. . . . .	45

# List of Tables

4.1	Examples of the five mutation operators. . . . .	20
4.2	The upper bound of RCI for the five mutation operators with the increasing of <code>hitting_ratio</code> . . . . .	24
5.1	Overview of the study subjects. . . . .	28
5.2	Overview of the selected mutants and covering tests that are involved in our experiment. . . . .	30
6.1	Stable and unstable benchmarks . . . . .	35
6.2	The percentage of tests that kill any of the mutants from five operators, assuming that the mutant is killed if the <i>bug_size</i> $\geq$ 1%, 5%, and 10%. . . . .	40
6.3	Mutation score of tests against the generated mutants from five operators, assuming that the mutant is killed if the <i>bug_size</i> $\geq$ 1%, 5%, and 10%. . . . .	41
6.4	Microbenchmarks that kill mutant and/or detect performance bug, assuming that the mutant is killed or the bug is detected if the <i>bug_size</i> $\geq$ 1%, 5%, and 10%. . . . .	45
6.5	The number of tests that killed any of the mutants, across the four groups of the three causes, for the three testing frameworks, and across the three study subjects. . . . .	49
6.6	Time saved using batch-executed strategy . . . . .	54
6.7	RSD of clusters and individuals, the percentage of Stable ( $\leq$ 1%) and Unstable ( $\geq$ 5%) microbenchmarks . . . . .	55

# Chapter 1

## Introduction

Performance is a critical non-functional requirement that directly impacts user experience and system reliability. Metrics such as execution time, throughput, resource utilization, and stability influence how users perceive software responsiveness and system efficiency [75, 18]. Performance testing enables developers to evaluate these attributes, providing insights into potential bottlenecks and system behavior under various conditions. Early identification of performance bottlenecks is essential to maintain software quality and meet user expectations. However, traditional performance testing often relies on large-scale, long-running system-level tests, which are resource-intensive, time-consuming, and difficult to integrate into modern agile and continuous integration (CI) practices [37, 44]. These limitations hinder the early detection of performance bugs, leaving critical issues undetected until later stages of development. Various studies have attempted to address these challenges by introducing optimizations in test execution [47, 39] or alternative performance testing methodologies [69, 48]. However, significant gaps remain in balancing testing precision, benchmark generation, and scalability.

To address these challenges, performance microbenchmarking frameworks, such as the Java Microbenchmarking Harness (JMH), have gained traction as tools for evaluating performance at a granular level, enabling precise measurement of isolated code segments, such as individual methods or algorithms [16, 17]. Developers must invest considerable effort in creating and maintaining benchmarks, which involves adhering to complex best practices to avoid inaccuracies and inefficiencies [18, 47]. Furthermore, JMH benchmarks can become resource-intensive at scale, making them challenging to integrate into continuous integration (CI) pipelines and modern agile workflows [43]. These challenges highlight the need for automated tools and optimized methodologies to reduce the costs and complexities associated with adopting JMH in large-scale software projects.

JUnit, on the other hand, is a popular testing framework widely used for functional unit testing in Java projects. It provides developers with a straightforward way to validate the correctness of their code through test assertions and integration tests. JUnit tests are lightweight, easy to write, and integrate seamlessly into CI pipelines, making it a preferred choice for ensuring functional quality. However, JUnit is inherently not designed for performance testing. Studies have shown that JUnit’s testing paradigm lacks the precision needed to accurately measure execution time, throughput, or resource utilization under controlled conditions [22, 52, 48, 74]. These limitations make JUnit unsuitable for detailed performance evaluations, especially in scenarios requiring statistical rigor and repeatability. Furthermore, while JMH offers robust performance testing capabilities, it lacks the simplicity and accessibility of JUnit for functional testing. This gap presents an opportunity to create a solution that bridges functional testing and performance benchmarking.

## 1.1 Purpose of the study

This thesis proposes a unified solution to streamline performance microbenchmarking in Java applications. We introduce *ju2jmh*, a novel framework that automates the conversion of JUnit tests into JMH microbenchmarks. By leveraging the widespread use of JUnit functional tests in software development, *ju2jmh* bridges the gap between functional testing and performance benchmarking, enabling developers to rapidly construct high-quality microbenchmarks with minimal effort. *ju2jmh* ensures that benchmarks retain the context of the original JUnit tests while enhancing their performance results precision through JMH’s benchmarking features.

Building upon this foundation, we further investigate the quality of benchmarks generated by *ju2jmh* and compare them to three other performance testing approaches: hand-crafted JMH benchmarks written by system developers, JUnit tests executed in loops to obtain their performance metrics, and AutoJMH benchmarks that evaluate code segments out of context [69]. Through this study, the quality of performance tests is evaluated using metrics such as stability and their ability to detect performance bugs. A high-quality performance test exhibits greater stability (consistent results across repeated runs) and a higher likelihood of identifying performance issues effectively and reliably. To assess these benchmarks comprehensively, the Performance Mutation Testing (PMT) framework is introduced as a tool to evaluate the quality of performance microbenchmarks. PMT introduces controlled variations or “mutants” to the code under test to evaluate how effectively benchmarks detect these changes, offering insights into the robustness and accuracy of the tested approaches.

Finally, we introduced a clustering strategy for an effective performance microbenchmarking approach. This strategy is based on code coverage measurements to group smaller microbenchmarks with similar contexts. By clustering benchmarks, we optimize their execution, reduce variability, and enhance scalability, making performance microbenchmarking more effective and efficient [43, 47]. The clustering strategy dynamically adapts to workload characteristics, ensuring consistent and efficient benchmarking even for large test suites.

The unified contributions of this thesis address critical gaps in performance microbenchmarking by automating benchmark generation, enhancing result reliability, and reducing execution overhead. These advancements provide developers with a practical and scalable approach to integrate performance benchmarking into modern development workflows, supporting the early detection of performance issues and improving overall software quality.

## 1.2 Thesis statement

Our research and prior experience have led to the formulation of the following hypothesis:

Automating the generation of performance microbenchmarks and optimizing their execution through clustering strategies can significantly enhance the efficiency, reliability, and scalability of performance testing workflows.

By streamlining the traditionally manual and resource-intensive process of creating microbenchmarks, automation has the potential to reduce developer effort while ensuring consistency and accuracy in benchmark design. Meanwhile, clustering strategies can address the challenges of long execution times and resource constraints by grouping similar benchmarks. By leveraging the widespread adoption of JUnit functional tests, the proposed ju2jmh framework is hypothesized to bridge the gap between functional testing and performance microbenchmarking, enabling developers to create high-quality benchmarks with minimal manual effort. Furthermore, it is posited that the introduction of a Performance Mutation Testing (PMT) framework can systematically evaluate the robustness of generated microbenchmarks, ensuring their ability to detect both artificial and real-world performance issues. Finally, the clustering strategy is hypothesized to enhance the practicality of performance microbenchmarking by reducing execution overhead and improving scalability without compromising test reliability or precision. Collectively, these approaches aim to address critical gaps in existing performance testing methodologies and advance the state of the art in software performance evaluation.

## 1.3 Contributions

The primary contributions of this thesis are:

- The design, implementation, and empirical validation of *ju2jmh*, a framework that bridges functional testing and performance benchmarking by automating the generation of JMH microbenchmarks from JUnit tests.
- The development of the Performance Mutation Testing (PMT) tool, providing a systematic method to evaluate the precision, reliability, and effectiveness of performance microbenchmarks, addressing critical gaps in benchmarking quality assurance.
- The creation of a clustering strategy that optimizes benchmark execution by reducing overheads, improving scalability, and ensuring efficient integration into continuous integration workflows.

# Chapter 2

## Background

This chapter provides an overview of performance microbenchmarking in the Java ecosystem and introduces key concepts and techniques, including performance mutation testing and real-world performance bug detection, which form the foundation of this thesis.

### 2.1 Performance Microbenchmarking in Java

Performance testing [75] evaluates non-functional attributes of software systems, such as execution time, response latency, resource usage, and stability. These metrics are essential for ensuring software quality and meeting user expectations. This thesis focuses on performance microbenchmarking, a specialized form of testing that analyzes small, isolated units of code, such as individual methods or algorithms. Unlike system-level performance testing, which assesses the overall application behaviour, or load testing, which examines system response under extreme workloads, microbenchmarking provides granular insights into specific components' efficiency and throughput.

Several frameworks have been developed to support performance microbenchmarking in Java, including Caliper [29], JMH (Java Microbenchmark Harness) [16], AutoJMH [69], and JUnitPerf [14]. These tools streamline the creation, execution, and analysis of microbenchmarks. However, microbenchmarking remains underutilized compared to functional unit testing and system-level performance assessments [74]. Studies reveal that many developers bypass specialized frameworks, instead relying on repetitive executions of functional tests to approximate performance metrics. This gap highlights the need for more accessible and integrated solutions that lower the barrier to adopting robust performance microbenchmarking practices.

## 2.2 Java Microbenchmarking Harness (JMH)

Performance microbenchmarking presents several challenges, including variability in results [40, 55], the need for expertise in benchmarking methodologies [27, 31], and a lack of user-friendly tools [57, 44]. JMH addresses these challenges by providing a structured framework for designing and executing reliable benchmarks.

JMH, developed under the OpenJDK umbrella, employs Java annotations to simplify the definition of benchmarks while ensuring precision and repeatability. Each JMH benchmark method is tagged with `@Benchmark`, indicating the code segment to be measured [69]. Additional annotations, such as `@Setup` and `@TearDown`, allow developers to define pre- and post-benchmark routines to ensure a consistent testing environment. Figure 2.1 illustrates a JMH benchmark class, demonstrating its annotation-based structure and setup routines.

```
public class MaxByIntTest extends AbstractJMHTestRunner
{
    Test fixture
    private final Positions positions = new Positions(SIZE).shuffle();
    private ExecutorService executorService;
    @Setup
    public void setUp()
    {
        this.executorService = Executors.
            newFixedThreadPool(
                Runtime.getRuntime()
                    .availableProcessors());
    }
    @TearDown
    public void tearDown() throws InterruptedException
    {
        this.executorService.shutdownNow();
        this.executorService.
            awaitTermination(1L, TimeUnit.SECONDS);
    }
    Test case
    @Benchmark
    public Position maxByQuantity_serial_lazy_direct_methodref_jdk()
    {
        return this.positions
            .getJdkPositions().stream()
            .max(QUANTITY_COMPARATOR_METHODREF).get();
    }
}
```

Figure 2.1: An example of a JMH benchmark class from the [Eclipse-collections](#) project (adapted from [35]).

JMH benchmarks measure metrics such as throughput or execution time with high

configurability. A single iteration consists of repeated workload executions, with "measurement time" (e.g., `@Measurement(time = 1)`) serving as the default evaluation period. The throughput metric, defined as operations per second is used throughout this study.

Compared to running JUnit tests in loops for performance measurement [48, 74], JMH benchmarks provide several advantages. They incorporate warm-up iterations to mitigate noise from just-in-time (JIT) compilation, support benchmark fixtures for consistent test setups, and enable flexible configurations for threads, iterations, and forks. These features make JMH benchmarks a superior choice for accurate and reliable performance evaluations.

## 2.3 Mutation Testing

Mutation testing evaluates the effectiveness of test suites by introducing artificial faults, called "mutants," into the codebase using mutation operators [63]. This approach identifies weaknesses in test coverage and improves fault detection. In recent years, mutation testing has been extended to performance testing, giving rise to performance mutation testing (PMT). Unlike traditional mutation testing, PMT introduces performance-specific changes, such as modifying loop conditions or altering resource allocations, to assess a performance test's ability to detect degradations [35, 21, 44].

PMT provides a structured way to evaluate the robustness of performance benchmarks. For example, Laaber and Leitner [44] highlighted how PMT can uncover shortcomings in benchmarks, while Delgado-Pérez et al. [21] demonstrated its practical application in real-world microbenchmarking scenarios. Despite its potential, PMT poses challenges, such as balancing the cost of generating mutants with their benefits and designing mutation operators for diverse performance concerns. While prior work has introduced tailored mutation operators [35], further efforts are needed to broaden their applicability and address emerging performance evaluation needs.

## 2.4 Detecting Real-world Performance Bugs

Performance bugs can be identified by analyzing test results for significant degradations. For instance, a substantial drop in throughput (e.g., 1%, 5%, or 10%) indicates a performance issue detected by the test.

In mutation testing, this process involves running tests on two versions of the system: one containing artificially introduced performance bugs (mutants) and the other without. By comparing the results, the presence of performance degradations indicates that the mutant has been effectively "killed."

Similarly, in real-world scenarios, performance tests are run on a "parent" system (assumed to contain a bug) and its "child" system, where the bug fixes are applied. Comparing the performance metrics of these two versions helps identify whether the parent system exhibited significant performance degradations, indicating the existence of a real-world performance bug. This systematic approach enhances the ability to detect and address performance issues in complex systems.

# Chapter 3

## Related Works

This chapter reviews related studies that form the basis of this thesis, focusing on three main areas: performance microbenchmarking, empirical studies on performance bugs, and approaches for assessing the quality of performance tests.

### 3.1 Performance Microbenchmarking

Performance microbenchmarking has become an essential tool for evaluating the performance of isolated code units, such as methods and algorithms. Despite these advances, microbenchmarking remains underutilized in practice. Studies show that many open-source projects rely on ad hoc solutions, such as repetitive execution of JUnit tests, to approximate performance metrics [48, 74]. This is partly due to the complexity of designing robust microbenchmarks. Developers often encounter challenges related to JIT optimizations, garbage collection interference, and result variability [40, 55, 47]. Tools like AutoJMH have attempted to automate benchmark generation but are limited to specific configurations and have not been maintained since 2016 [69].

The challenges in usability, configuration, and integration with modern workflows underscore the need for more accessible and automated solutions. This thesis addresses these gaps by introducing *ju2jmh*, a framework that bridges functional testing and performance benchmarking through automated generation of JMH benchmarks from JUnit tests.

## 3.2 Empirical Studies on Performance Bugs

Performance bugs, which degrade system performance without impacting functional correctness, are a significant concern for developers. These bugs often manifest in resource bottlenecks, inefficient algorithms, or poorly optimized code paths [38, 51, 59]. Identifying and addressing performance bugs is particularly challenging due to their non-deterministic nature and sensitivity to runtime conditions [77]. Empirical studies highlight that it can take years to discover and resolve performance bugs, emphasizing the need for better detection mechanisms [38].

Several studies have catalogued performance bug patterns to aid developers in reproducing and addressing these issues. For example, [67] introduces the *NFBugs* dataset, which identifies eight common performance bug patterns. Similarly, [21] reviews prior studies and summarizes seven performance anti-patterns. These findings provide valuable insights but fall short of offering automated tools for detecting performance bugs in practice.

Recent work has explored integrating performance testing into CI pipelines to enable early detection of performance issues [22]. However, existing solutions often rely on manually crafted tests or require substantial expertise, limiting their scalability. This thesis contributes to this area by demonstrating how *ju2jmh* can automate performance test creation and enhance the detection of performance bugs through systematic mutation testing.

## 3.3 Assessing the Quality of Performance Tests

The quality of performance tests is a critical factor in ensuring accurate and reliable detection of performance issues. Stability—defined as the consistency of results across repeated runs—is a key quality attribute [44, 46]. Another important metric is the ability of tests to detect performance degradations, often evaluated using techniques like mutation testing [21].

Mutation testing for performance, or Performance Mutation Testing (PMT), has emerged as a novel approach to assess the robustness of performance benchmarks. Studies like [44] and [21] have demonstrated how PMT can identify weaknesses in benchmarks by introducing controlled slowdowns or anti-patterns. However, these studies often rely on manual processes, limiting their practicality in real-world settings.

This thesis extends the state-of-the-art by proposing an automated PMT framework that evaluates the quality of performance benchmarks generated by *ju2jmh*. The framework introduces performance-specific mutation operators to systematically assess test robustness and sensitivity to performance changes, addressing limitations in previous approaches.

### 3.4 Optimizing Microbenchmark Execution

Benchmarking, particularly microbenchmarking, is computationally intensive, often requiring significant resources to execute and generate meaningful results [70]. Sequential execution on limited resources further exacerbates execution time [70]. An additional challenge arises from redundant or ineffective microbenchmarks, which offer limited value to performance testing processes. [44] highlighted the prevalence of benchmarks with overlapping or narrow coverage, emphasizing the inefficiencies they introduce into testing workflows. Addressing these inefficiencies is critical to improving the scalability and practicality of performance microbenchmarking.

Efforts to enhance performance testing efficiency have spurred various methodologies and tools [47, 65, 44, 46, 35]. Dynamic termination of microbenchmark execution upon achieving stability, as proposed by [47], has shown promise in reducing execution time without compromising quality. Similarly, [31] demonstrated the effectiveness of risk-based test prioritization in optimizing regression testing workflows. Automated approaches, such as those by [4], have introduced stability-based stopping criteria to further streamline testing processes. While batch execution strategies have been explored in broader contexts, including continuous integration systems [24] and clustered test execution [?], their application to microbenchmarking remains underexplored. This study addresses this gap by proposing a tailored batch execution strategy for microbenchmarks, demonstrating significant improvements in execution efficiency while maintaining result quality.

# Chapter 4

## Study Overview

This chapter presents the methodology employed in this thesis, which encompasses three key phases: (1) generating microbenchmarks using the *ju2jmh* framework, (2) evaluating the quality of these microbenchmarks through the Performance Mutation Testing (PMT) framework, and (3) leveraging clustered execution to improve the scalability and effectiveness of the generated microbenchmarks.

### 4.1 Overview of the Study Workflow

Figure 4.1 provides an overview of the workflow underlying our methodology. The process begins by extracting JUnit test suites and existing JMH benchmarks from the selected study subjects. These components form the foundation for generating new microbenchmarks and evaluating their capability to detect performance bugs. The *ju2jmh* framework is utilized to convert JUnit test suites into JMH microbenchmarks, while the PMT framework introduces artificial performance bugs (mutants) to assess the quality and robustness of the generated benchmarks. Additionally, AutoJMH, known for its ability to generate microbenchmarks from single code segments and their covering JUnit tests, is employed for comparative analysis. To further optimize performance microbenchmarking, we propose a clustering strategy that facilitates the grouped execution of microbenchmarks.

### 4.2 Finding Tests and Preparing Environment

Before conducting experiments, it is essential to prepare the test environment. This involves extracting existing JMH microbenchmarks and JUnit test suites from the study subjects,

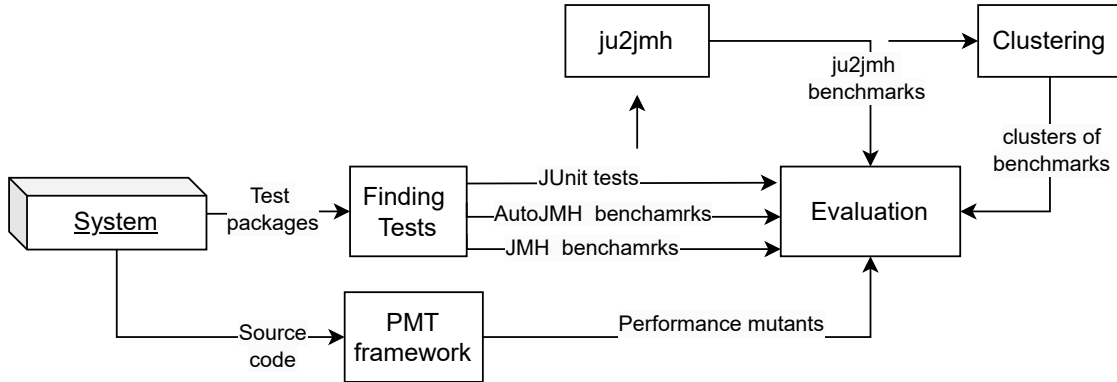


Figure 4.1: Workflow of our methodology for generating and evaluating performance microbenchmarks.

which serve as the basis for subsequent analysis.

#### 4.2.1 Human-Written JMH Microbenchmarks

The Java Microbenchmarking Harness (JMH) is a robust framework for creating reliable microbenchmarks, demonstrating significant promise in scaffolding performance-critical code segments, often referred to as payloads [69]. Human-written JMH microbenchmarks are typically crafted by the system’s original developers, who possess deep domain knowledge of both the system and the JMH framework. However, designing and maintaining these benchmarks require substantial expertise and effort, often discouraging their widespread adoption. In this study, we analyze existing human-written JMH microbenchmarks of various systems. To facilitate their execution, an automated build tool generates an executable jar file, allowing testers to seamlessly run the JMH benchmarks.

#### 4.2.2 JUnit Test Suites

JUnit test suites are primarily intended to verify the functional correctness of individual units within a software system. These tests are not inherently designed for performance measurement and are particularly susceptible to environmental noise, making their use for performance evaluation less reliable. Nevertheless, prior work [22] has demonstrated how JUnit tests can be adapted for performance assessment. In this study, we measure the performance of JUnit tests by calculating their throughput, defined as the number of executions completed within a specified duration (e.g., one second).

To achieve this, we employ JUnit `@Rules` to monitor individual test executions and record elapsed time in nanoseconds, thereby determining the throughput of each test method. Our experiments involve using a nested loop structure: the inner loop repeatedly executes the test within a fixed duration to calculate throughput, while the outer loop performs this measurement multiple times to generate a reliable dataset for comparison. Each test method’s throughput is measured 30 times to establish a baseline.

### 4.2.3 AutoJMH Microbenchmarks

The AutoJMH framework [69] offers an automated approach for generating JMH microbenchmarks by using a single code segment and its covering unit test as input. AutoJMH benchmarks focus on evaluating the performance of individual statements rather than executing the entire program. The framework extracts variable values from the covering unit test execution and incorporates them into the generated benchmark. However, AutoJMH imposes strict constraints: only single statements covered by unit tests can be used as input, and significant manual effort is required for configuration. Moreover, the framework has not been actively maintained since August 2016, rendering it outdated.

### 4.2.4 Comparing AutoJMH and *ju2jmh*

Both AutoJMH and *ju2jmh* generate JMH microbenchmarks using JUnit tests as input, but their methodologies differ significantly. AutoJMH executes a JUnit test only once to gather the required data for constructing benchmarks, while *ju2jmh* generates benchmarks designed to repeatedly execute JUnit tests as payloads, capturing their performance characteristics more thoroughly.

Additionally, AutoJMH benchmarks isolate and execute single code statements without running the entire program, whereas *ju2jmh* benchmarks run the full JUnit test within its intended program context. AutoJMH also requires developers to manually specify individual statements as input, limiting its scalability. Consequently, the number of benchmarks generated by AutoJMH in this study is significantly lower than those produced by *ju2jmh*, leading to reduced program coverage. In contrast, *ju2jmh* leverages the extensive JUnit test suites typically available in well-known programs, enabling broader and more comprehensive benchmarking.

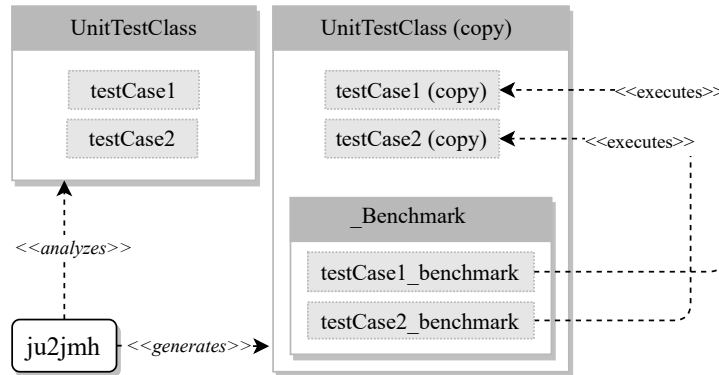


Figure 4.2: Overview of the *ju2jmh* approach and its inputs and outputs

### 4.3 Generating Microbenchmarks Using *ju2jmh*

To demonstrate the feasibility of generating JMH benchmarks from unit tests, we implement a tool dubbed *ju2jmh* that can automatically generate ready-to-execute JMH benchmarks from JUnit4 test suites. The main goal of *ju2jmh*'s design is to be able to generate functionally correct benchmarks from a wide variety of real-world unit tests while minimizing performance overhead and complexity.

Figure 4.2 depicts the basic benchmark generation procedure of *ju2jmh*. The procedure consists of two main steps: (1) **Analysis step**: the *ju2jmh* tool analyzes the existing unit test classes in order to identify individual JUnit test methods and other relevant test features; (2) **Benchmark generation step**: the tool generates JMH benchmarks with each benchmark method responsible for repeatedly executing (a copy of) a single unit test method as its payload. For example, as shown in Figure 4.2, if *ju2jmh* is applied to a unit test class called `UnitTestClass`, which contains two unit test methods named `testCase1` and `testCase2`, *ju2jmh* first generates a copy of the `UnitTestClass` class, and then generates a JMH benchmark class called `_Benchmark` that is placed within this copy. The new benchmark class contains two JMH benchmark methods, `testCase1_benchmark` and `testCase2_benchmark`, whose responsibilities are to repeatedly execute `testCase1` and `testCase2` as their payloads. Similarly, as shown in Figure 4.3, the *ju2jmh* tool is applied to a JUnit test case inside [Eclipse-collections](#) project and generates a new class containing the copied JUnit test, the associated generated JMH benchmark and the benchmark's fixtures.

In the analysis step, *ju2jmh* first identifies the individual JUnit test methods that are expected to be converted to JMH benchmarks. Next, *ju2jmh* detects any other features of the test methods that are required for proper execution, such as fixture methods, test

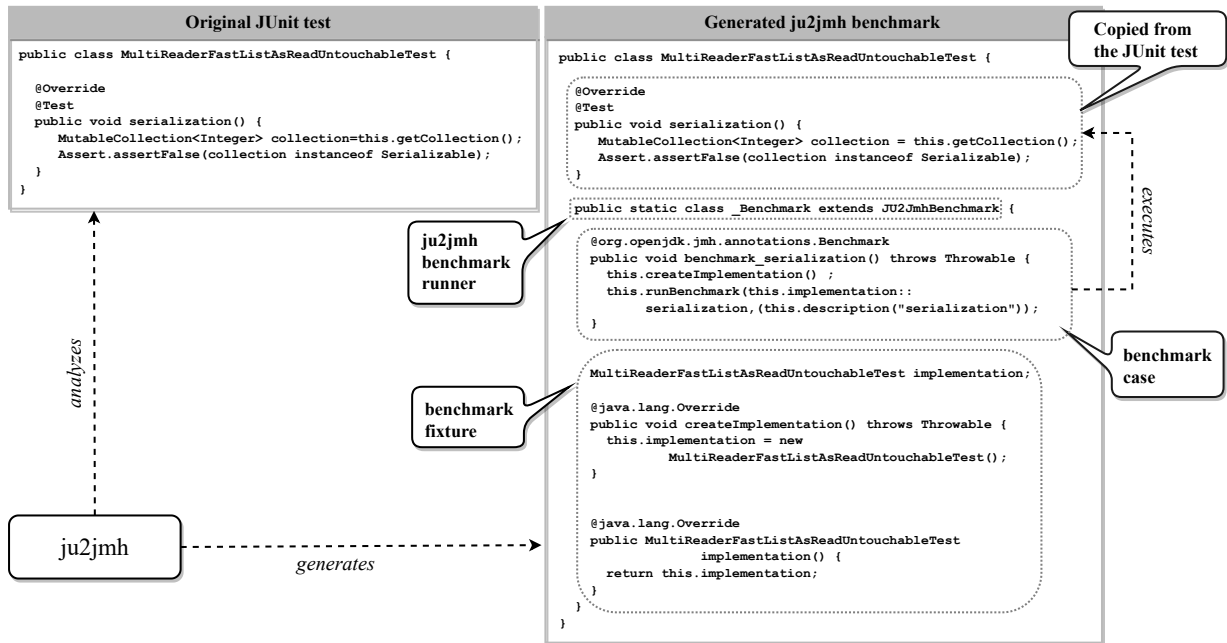


Figure 4.3: A real-life generated `ju2jmh` benchmark from `Eclipse-collections`, and how it actually performs JMH rules

rules, and expected exceptions, all of which should be handled explicitly by the generated benchmarks. The analysis is performed using *Apache Commons BCEL*<sup>1</sup> to statically inspect the bytecode of all relevant JUnit test classes (specified as input for the tool) in order to find test methods (annotated by `org.junit.Test`), fixture methods (annotated by `org.junit.Before`, `org.junit.After`, `org.junit.BeforeClass`, or `org.junit.AfterClass`), test rules (annotated by `org.junit.Rule` or `org.junit.ClassRule`), expected exceptions (extracted from the arguments of the `org.junit.Test` annotation), and ancestor classes containing any of the above.

In the benchmark generation step, `ju2jmh` uses the *JavaParser*<sup>2</sup> library to parse Java source code and generate the Abstract Syntax Tree (AST). This step consists of the following activities: (1) copying the AST of each relevant JUnit test class, (2) generating the AST for a corresponding `_Benchmark` class that is customized based on the information gathered in the analysis step, (3) inserting the generated benchmark class into the AST of

<sup>1</sup>A Java bytecode analysis tool from The Apache Software Foundation: <https://commons.apache.org/proper/commons-bcel/>

<sup>2</sup>A tool for analyzing, transforming and generating Java codebase: <https://javaparser.org/>

the copied JUnit test class as a static member class, and (4) writing the results to a Java source file. The generated benchmark class contains a JMH benchmark method for each of the JUnit test methods in the unit test class.

Each generated benchmark inherits the same superclass, which implements the functionality required to execute JUnit tests properly within the benchmark. The *ju2jmh* benchmark superclass contains methods to help instantiate and access a JUnit test class instance, methods to invoke the fixture methods of JUnit test class (including all fixture methods from its superclasses), methods to apply all rules of the JUnit test class and its superclasses, and a benchmark execution method to execute the JUnit test methods. In the cases where tests specify expected exceptions, the *ju2jmh* benchmark superclass executes a different benchmark execution method with the expected exceptions as an additional parameter.

Concerning the structure of the generated benchmarks, the use of copies of the original unit tests as benchmark payloads is preferred over directly referencing the original classes, since this provides potential future versions of *ju2jmh* more flexibility in being able to modify the benchmark payloads without impacting the original tests. Future versions may for example modify the benchmark payloads to try preventing optimizations that may distort performance measurement, such as dead code elimination, without having to make the same modifications to the original unit tests. Moreover, rather than making the generated benchmark class a separate top-level class, it is inserted as a static member of the copy to reduce the risk of naming conflicts and to keep the benchmarks organized closely with the tests they use as payloads.

Our approach *ju2jmh* has been implemented in Java and integrated into the Gradle build system. *JavaParser* and *Apache Commons BCEL* are the primary libraries used for the unit test analysis and benchmark generation. The generated benchmarks are dependent on the JMH [16] and JUnit4<sup>3</sup> libraries.

In this section, we present the workflow of our PMT framework for evaluating unit-level performance tests. First, the framework implements and employs five performance mutation operators that are derived from common real-world performance bugs [31, 21, 44]. Second, it automatically injects performance bugs (based on the five mutation operators) into the analyzed software system and generates performance mutants (i.e., artificial performance bugs). Then, the generated performance mutants are executed against the to-be-evaluated performance microbenchmarks. Furthermore, to better classify between an actual slowdown (i.e., a performance bug) and a performance fluctuation, we leverage hierarchical re-sampling [47, 40] and two statistical measures, namely, *Relative Standard*

---

<sup>3</sup>A framework to write repeatable Java unit tests: <https://junit.org/junit4/>

*Deviation* (RSD) [47, 49, 46] and *Relative Confidence Interval of means* (RCI) [47, 40]. Last, the PMT framework computes the *mutation score*, which is defined as the proportion of the killed (detected) performance mutants, to assess the effectiveness of performance microbenchmarks in detecting performance bugs.

### 4.3.1 Designing Mutation Operators for PMT

Our study aims at assisting developers in alleviating the challenges of constructing and evaluating performance tests rather than detecting performance bugs. To achieve our study goal, we use PMT to inject artificial performance bugs for further analysis. We evaluate performance tests by comparing the test results on the systems with and without performance bugs. Performance tests can help developers reveal the effects that mutants have on the systems, detect performance bugs and complement the omission of static tools that may treat some code as syntactically correct but cannot reveal its negative impact on performance.

A PMT framework requires specialized mutation operators for generating performance mutants (i.e., software versions with artificial performance bugs or slowdowns). In this study, we expose five performance mutation operators based on previous research on summarizing performance anti-patterns [67, 21, 44], which we then implement in our PMT framework.

In particular, [21] review prior empirical studies on performance bugs [62, 51, 59, 71, 50, 56] and summarize seven performance anti-patterns, of which we employ two patterns for designing our performance mutation operators and exclude five others (three memory-related bug patterns and two for future work). [67] present the NFBugs dataset, which contains eight performance bug patterns discovered after analyzing 36 performance bug fixes from a total of 138 non-functional bug fixes in 67 open-source Java or Python projects. We adopt three performance anti-patterns from NFBugs and leave the remaining five as future work. Furthermore, [44] propose inserting `Thread.sleep(...)` into source code to slow down program execution for performance mutation testing, which we also accept for the design of our performance mutation operators. We selected the five performance bug patterns because of the high availability of source code statements that a pattern can be applied to. Moreover, the five patterns do not require manipulating multiple lines of source code, and generated bugs rarely tend to endanger the overall functionality.

In summary, we introduce a total of five performance mutation operators based on prior studies on analyzing performance bugs and implement them in our PMT framework. Such performance mutation operators are not specific to certain software systems and can

be applied to any Java system without manual effort. Moreover, the PMT framework is extensible, allowing for the easy integration of new performance mutation operators.

Below we describe the five performance mutation operators in detail, with a summary in Table 4.1.

- **Primitive to Wrapper** (PTW). Replacing a primitive type (e.g., `long`) with its corresponding wrapper class (i.e., `Long`) can lead to a performance bug [67], since primitive types are stored on the stack and provide faster access. We include the built-in wrapper classes for all primitive types, i.e., `Byte`, `Boolean`, `Short`, `Integer`, `Long`, `Float`, `Double`, and `Character`.
- **StringBuilder to StringBuffer** (STS). Replacing a `java.lang.StringBuilder` object with a `java.lang.StringBuffer` object can result in a performance bug because `StringBuilder` is not synchronized [67]. Apart from the difference in performance, the functionality of these two classes is equivalent. Therefore, we perform the operator only at method level to prevent propagation of changes.
- **Enhanced For Loops** (EFL). Replacing a traditional `for`-loop with a `for-each` loop to iterate over an array or a `Collections` class can introduce a performance bug because of extra calls. The semantics of the code is usually unaffected by such replacements. However, if the loop counter is used in the `for`-loop, the replacement may cause a performance bug due to the additional cost of calculating it.
- **Swap of Operands in Condition** (SOC). Reordering the two operands in a compound OR condition if the left operand is a variable or a Boolean literal (i.e., `True` and `False`), and the right operand is a method invocation. Forcing the evaluation of the right operand (i.e., a method invocation) by placing it as the left operand in a compound OR condition may introduce a slowdown since the method invocation may not be executed originally as the right operand [21].
- **Simulation of Heavy-Weight Operations** (HWO). Injecting a delay (e.g., `Thread.sleep(t)`) can slow down the program execution. In our study, we consider injecting a delay as the first statement of each JMH microbenchmark (annotated by `@Benchmark`). We chose the first statement as the candidate injection location since we expect to see that the delay is executed definitively, i.e., the injected delay cannot be affected by the data-flow and control-flow for the code in this method. For example, if a delay is injected into a branching statement, the delay may never be executed depending on the branching conditions. The HWO has been used in prior studies [21, 44] to simulate slowdowns, however, it does not represent real-world performance bugs. Nevertheless, we believe the HWO complements the other mutation operators based on real-world performance bugs and thus include HWO in this study.

Table 4.1: Examples of the five mutation operators.

Operator	Conditions	Example
PTW	Candidate expression should be a variable from any primitive types; i.e., byte, boolean, short, int, long, float, double, and char.	<pre>- return this.c; + return (Long.valueOf(this.c)).longValue();</pre> <p>(a) Counter.java (Commit: #8948b46, <a href="#">Eclipse-collections</a>)</p>
STS	This operator should manipulate one method at a time to prevent propagation of changes.	<pre>- StringBuilder result = new StringBuilder(); + StringBuffer result = new StringBuffer(); ... return result.toString();</pre> <p>(b) QueryRequest.java (Commit: #4fb8d5a, <a href="#">Zipkin</a>)</p>
EFL	The candidate loop should be a traditional for-loop that contains a loop counter.	<pre>int i = 0; - for (; i &lt; subscribers.length; i++) { + for (Subscriber&lt;? super R&gt; o : subscribers){ + i = subscribers.indexOf(o); ...}</pre> <p>(c) ParallelMap.java (Commit: #0df952e, <a href="#">Rxjava</a>)</p>
SOC	1) The left operand should be a variable or a Boolean literal. 2) The right operand should be a method invocation that never return Null.	<pre>- if (done    emitter.isCancelled()) + if (emitter.isCancelled()    done)</pre> <p>(d) FlowableCreate.java (Commit: #0df952, <a href="#">Rxjava</a>)</p>
HWO	This pattern is not subject to any conditions.	<pre>+ Thread.sleep(1); ... return TreeBag.newBag();</pre> <p>(e) ImmutableEmptyBag.java (Commit: #8948b46, <a href="#">Eclipse-collections</a>)</p>

### 4.3.2 Generating and Validating Performance Mutants

Figure 4.4 depicts an overview of the steps that the PMT framework takes to evaluate the quality of performance benchmarks by deploying the performance mutation testing technique. The PMT framework, firstly, deploys mutation operators to generate performance mutants inside the source code. Next, it runs unit tests of the system against mutants to verify that the functionality of the system remained valid. Lastly, it executes performance benchmarks against the original system and generated mutants, then calculates the mutation score of benchmarks. The PMT framework parses the source code of a target system into Abstract Syntax Trees (ASTs) and looks for opportunities to apply the five performance mutation operators on the parsed ASTs. For each applicable AST location and each mutation operator, the PMT framework generates one performance mutant. For example, the EFL mutation operator can be applied to a subset of traditional `for`-loops in one system; in `Rxjava`, the PMT framework finds 62 applicable locations and generates a total of 62 performance mutants.

Note that if a mutant is killed by any *functional* test, it is an invalid performance mutant since it breaks functionality. The system’s regular behavior could be disrupted by the presence of such mutants, which leads to the system’s performance not being properly measured. Therefore, in this study, we only consider performance mutants that do not change system functionality and require performance tests to detect. As a result, we introduce a validation step in the PMT framework: All the generated performance mutants are executed against the functional unit tests and are excluded from the further step (i.e., calculating mutation score) if one cannot make all the functional tests pass.

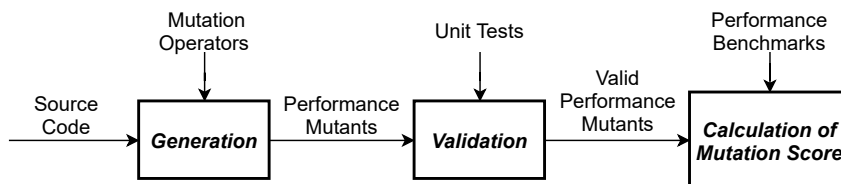


Figure 4.4: Overview of our Performance Mutation Testing framework.

### 4.3.3 Calculating Mutation Score

Relying on the purpose of mutation testing in performance that is introducing performance bugs deliberately, we looked over the benchmark’s results against a valid mutant to find significant differences. Recent studies [21, 44] have been advocating *statistical hypothesis*

*tests* (e.g., Wilcoxon rank-sum test) to label a significant difference in results as a performance bug, slow down, or the presence of artificial bugs. However, prior studies [46] concluded that testing with Wilcoxon rank-sum tests is not a suitable vehicle for detecting performance degradation in cloud environments due to high false-positive reported.

To better classify between an actual performance slowdown and a performance fluctuation, we leverage hierarchical re-sampling [47, 40] and a statistical measure, i.e., **Relative Confidence Interval of means** (RCI) [47, 40].

For a benchmark that covers and executes any of the generated performance bugs, the PMT framework deploys *pa* [42] to estimate the throughput’s RCI (of before and after the bug injection) with bootstrap (i.e., re-samples) [68, 19] using 10,000 bootstrap iterations [30]. Respectively, the size of a performance bug can be defined as  $[1 - U_{upper\_RCI}]$  (in %), which reflects the effectiveness of the mutant against the benchmark. If the bug size is significant enough (e.g.,  $\geq 5\%$ ), we can consider that the benchmark could detect the injected performance bug (i.e., a killed mutant).

Last, the PMT framework calculates the mutation score of performance microbenchmarks as the percentage of the killed performance mutants. The calculated mutation score helps evaluate the quality of one or a group of microbenchmarks. The microbenchmarks’ quality (and efficiency) in discovering bugs increases as their mutation score rises.

One of the advantages of mutation testing is to find deficiencies in tests and improve them, and improved tests can be further utilized to detect real-world bugs [66]. That is, we use artificial performance bugs to build and enhance the tests, and such tests could be used in the future to detect real-world performance bugs. If a performance test can detect an artificial performance bug, it is more likely to detect a real-world version of that bug.

#### 4.3.4 Preliminary Analysis of PMT

To investigate the impacts of the five mutation operators, we devise a preliminary analysis based on five JMH benchmarks. We define the `hitting_count` as the total number of times that a benchmark executes (hits) a mutant statement. Furthermore, the `hitting_ratio` is the total number of times that a mutant statement is executed per second.

To investigate how large the effect of each mutation operator is when increasing *hitting\_ratio*, we designed five JMH benchmarks. Each related benchmark is executed for 20 iterations against the original source code and the mutant version five times, yielding 100 data points containing measured throughput for one second. If differences between original results and mutant results increase, we assume that the effect of the performance bug is increased.

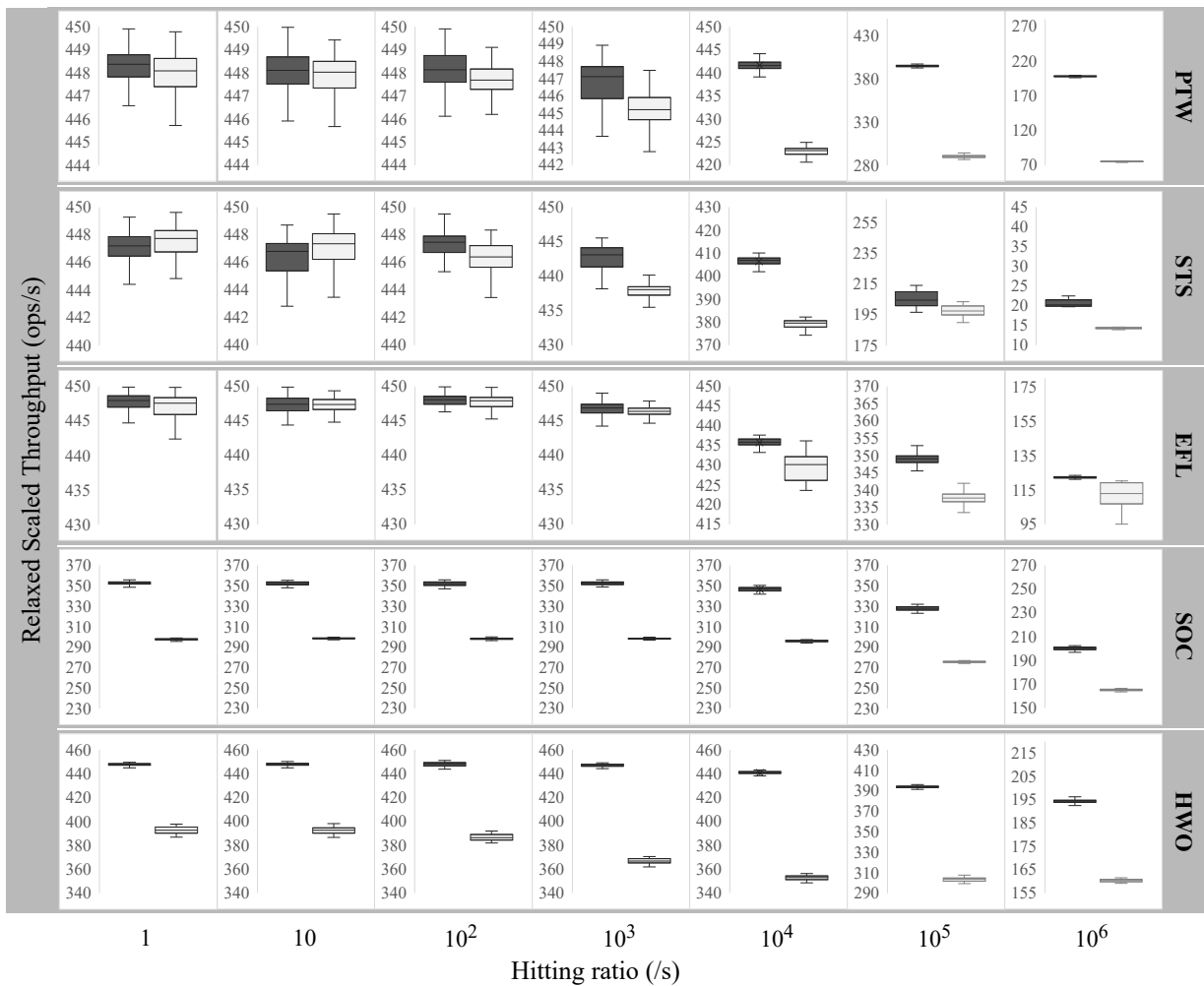


Figure 4.5: Preliminary analysis of the five mutation operators. Each of paired boxes contains obtained 100 data points for two cases of original source code executions and mutant executions, from each of five designed JMH benchmarks, across increasing `hitting_ratio`. Each box contains 100 data points obtained from a relevant benchmark. Black boxes represent data from original source code executions, and white boxes represent data from mutant executions.

Figure 4.5 presents box plots of the five benchmarks’ results for original and mutant executions, across increasing the `hitting_ratio`. To study the difference between the original microbenchmark execution and the microbenchmark executions after bug injections, we present the upper bound of the estimated RCI in Table 4.2. The upper bound of the RCI reflects the minimum performance degradation caused by bugs.

In general, for all five mutation operators, when we increased `hitting_ratio`, differences between original results and mutant results become more significant (the upper bound of the RCI decreases). However, we also observed that the RCI upper bound of HWO (0.824) for  $10^6$  in Table 4.2 is greater than  $10^5$  (0.771), indicating that this conclusion may not be applicable to a very high `hitting_ratio` for a certain operator and the results may be limited by some external factors, such as overheads and variations occurring due to hardware-specific limitations. When `hitting_ratio` is lower than  $10^3$  times, differences are not significant in PTW, STS, and EFL, with less than 1% performance degradation in all cases. On the other hand, in SOC and HWO, differences are significant in all `hitting_ratios`. In SOC and HWO, `hitting_count` is fixed while `hitting_ratio` is increased. This conveys that the effect of the two operators is high, even with one execution.

To summarize, when a benchmark executes a buggy statement (i.e., mutant statement) more times, the observed difference in results generally increases. In addition, differences are significant in all `hitting_ratios` in SOC and HWO.

Table 4.2: The upper bound of RCI for the five mutation operators with the increasing of `hitting_ratio`.

Operator	Hitting ratio						
	1	10	$10^2$	$10^3$	$10^4$	$10^5$	$10^6$
<b>PTW</b>	0.999	0.999	0.999	0.997	0.958	0.736	0.379
<b>STS</b>	1.0	1.0	0.998	0.990	0.933	0.959	0.688
<b>EFL</b>	0.998	1.0	0.999	0.999	0.987	0.968	0.920
<b>SOC</b>	0.844	0.848	0.847	0.848	0.854	0.841	0.827
<b>HWO</b>	0.879	0.878	0.864	0.821	0.801	0.771	0.825

## 4.4 Clustering Approach for Optimization

Performance testing is essential for identifying bottlenecks and optimizing software systems but often remains time-consuming and resource-intensive. Large performance test suites, common in industry projects, involve executing numerous microbenchmarks individually.

While small microbenchmarks are precise and efficient for evaluating targeted functionality, their limited scope often fails to detect broader performance trends or anomalies [35]. Additionally, running small, isolated microbenchmarks incurs inefficiencies due to repeated overhead from initialization, execution, and result analysis. This repetitive processing consumes significant resources, making the testing process impractical, especially for large-scale microbenchmarking suites.

Existing tools like *ju2jmh* reduce developers' effort in generating microbenchmarks but do not address these inefficiencies or the broader limitations of small microbenchmarks. Our research proposes enhancing *ju2jmh* microbenchmarking efficiency by batching functionally similar microbenchmarks into clusters for performance testing. This clustering approach offers several benefits:

- **Broader Analysis:** Enhances the collective utility of microbenchmarks, enabling them to contribute to identifying broader performance trends.
- **Efficiency Gains:** Reduces redundant overhead, optimizing the use of computational and testing resources.
- **Improved Scalability:** Makes the performance testing process more practical and scalable, particularly for large-scale projects.

Leveraging batch execution for microbenchmarks can significantly enhance the efficiency of the execution process. Moreover, since batch execution does not directly alter the microbenchmarks, this approach minimizes the risks associated with improving test execution efficiency, which makes it possible to enhance microbenchmarking efficiency while maintaining the original accuracy. Ultimately, our goal is to streamline performance testing workflows, enabling faster feedback loops, reduced computational burdens, and more accessible performance testing for developers.

To enable batch-executing microbenchmarks, we leverage code coverage information to cluster functionally similar microbenchmarks. The *JaCoCo agent* [1], a library for calculating code coverage in Java projects, is employed to measure the percentage of overlapping covered code lines, which serves as an indicator of functional similarity. The similarity score ranges from 0% (no overlap) to 100% (complete overlap where the smaller test's coverage is a subset of the larger one). While a 100% similarity score reflects substantial alignment in covered code segments, it does not necessarily imply identical functionality. Instead, it indicates a shared scope that allows for meaningful grouping of related benchmarks. This nuanced understanding of coverage ensures that clusters are cohesive and functionally relevant. Furthermore, batch execution of microbenchmarks maintains the same code coverage as individual runs, preserving the collective coverage achieved by each microbenchmark within the cluster.

Algorithm 1 illustrates our methodology for batch execution of microbenchmarks. It consists of two steps: 1) The first step involves ranking the most similar *ju2jmh* microbenchmarks for each hand-crafted JMH microbenchmark based on code coverage similarly. The more similar a microbenchmark is, the higher the chance it will be clustered into the same cluster. 2) For each hand-crafted JMH microbenchmark, we group the top-K most similar *ju2jmh* microbenchmarks to enable batch execution. The value of K is determined by incrementally summing the sizes of individual microbenchmarks until their combined execution time reaches an empirically observed threshold sufficient for detecting performance bugs (i.e., 5  $\mu$ s) [35].

---

**Algorithm 1** Benchmark Batching

---

```

1: Input: Handcrafted JMH Microbenchmarks, ju2jmh Microbenchmarks, Threshold  $T$ 
2: Output: Clusters of Microbenchmarks
3: // Similarity Scoring
4: for each  $JMH$  Handcrafted Microbenchmarks do
5:   for each  $ju2jmh$  Microbenchmarks do
6:     Compute and store similarity( $JMH, ju2jmh$ )
7:   end for
8:   Rank  $ju2jmh$  by similarity for  $JMH$ 
9: end for
10: // Cluster Formation
11: for each  $JMH$  Hand-crafted Microbenchmarks do
12:   Create  $cluster$ ;
13:   Add top  $ju2jmh$  until size( $cluster$ )= $K$ 
14:   If size( $cluster$ )  $\leq K$  then Continue
15:   Add  $cluster$  to clusters
16: end for
17: Return: Clusters

```

---

## 4.5 Summary

This chapter outlined the methodology employed to enhance and evaluate performance microbenchmarking in Java applications. We introduced the *ju2jmh* framework, which automates the conversion of JUnit functional tests into JMH microbenchmarks, providing an accessible pathway for integrating performance testing into modern development workflows. To ensure the quality and reliability of the generated benchmarks, we implemented the Performance Mutation Testing (PMT) framework, which systematically evaluates benchmark robustness using carefully designed mutation operators. Additionally, we proposed a

clustering strategy based on code coverage analysis to optimize benchmark execution, improving scalability and reducing redundancy in performance testing. These methodologies collectively address critical challenges in benchmark generation, evaluation, and execution, forming a cohesive and practical approach to advancing software performance testing.

# Chapter 5

## Study Setup

In this section, we describe our empirical study to test our hypothesis that the derived performance microbenchmarks from *ju2jmh* are preferable to using JUnit tests directly as performance proxies. Furthermore, we compare our derived microbenchmarks to human-written microbenchmarks to evaluate if the derived ones are superior than human-written ones. Moreover, we compare our derived microbenchmarks with the microbenchmarks automatically generated by AutoJMH <sup>1</sup> to evaluate if *ju2jmh* outperforms the existing framework.

### 5.1 Study Subjects

Table 5.1: Overview of the study subjects.

	Version	Stars	Contr.	SLOC	# JMH	# JUnit
RxJava	3	44.7k	277	311,975	1,217	9,825
Ec-collections	10.4.0	1.7k	88	135,017	986	24,758
ZipKin	2.7	14.4k	145	7,467	59	501

In this section, we introduce the three study subjects involved in this study. We experimented and evaluated our two frameworks, i.e., *ju2jmh* and PMT, on three open-source Java projects, [Rxjava](#), [Eclipse-collections](#), and [Zipkin](#), which have readily available JUnit test cases and JMH microbenchmarks. The selected three open-source projects are

<sup>1</sup><https://github.com/DIVERSIFY-project/autojmh-source-code>

well-known, well-maintained, and were widely studied in prior studies on performance microbenchmarking [21, 44, 47, 18, 46, 48]. In addition, a recent study [18] lists the three subjects among the top 25 projects with the highest number of JMH micro-benchmarks.

Table 5.1 provides detailed information of our study subjects: the studied version (“Version”) (i.e., the most recent version at the time our study began), extracted metadata from **GitHub** including the numbers of stars (column **Stars**) and the number of contributors (column **Contr.**), the number of the source lines of code (column **SLOC**), the total number of JMH benchmarks (column **# JMH**) and selected JUnit test cases (column **# JUnit**).

## 5.2 Experiment Settings

We deployed *ju2jmh* on the three subjects’ JUnit test suites to build 171,366 *ju2jmh* benchmarks, of which 35,084 were evaluated in this study, accounting for 48% of total 72,430 tests evaluated. Our *PMT* framework analyzed a total of ~454K SLOC and generated ~149K artificial performance mutants, each of which was executed in a single position of the source code. During the validation procedure, ~99% of the generated mutants were recognized as *possibly valid* mutants.

**Eclipse-collections** includes a significant number of JUnit tests that take roughly four months to run once; as a result, we randomly selected ~15% of them for examination while studying all tests from two other subjects. Afterward, in order to have a fair and feasible evaluation approach, we selected a subset of the generated mutants with a wide variety of characteristics while considering all possible tests that cover them. To increase the chance of evaluating more tests against a specific mutant, we selected each mutant from the set of a class’s mutants in different packages of the source code, as well as selected mutants that are covered by more JMH benchmarks and *ju2jmh*/JUnit tests.

Table 5.2 provides an overview of the mutants and tests involved in our measurements. For each of five designed mutation operators and across the three study subjects, we present the number of selected mutants (column **# Mutants**), all contained JMH benchmarks (column **JMH**), *ju2jmh*/JUnit tests that cover any of the mutants (column **JUnit/ju2jmh**), and generated AutoJMH benchmarks (column **Auto.**).

Our study establishes three performance oracles: (1) If there is no source code update, the performance tests results should remain relatively unchanged; (2) If the existence of a performance bug is confirmed and covered by a performance test, there should be a variation in the testing results and such variation should be consistent with the characteristics of the performance bug; (3) If a mutation is injected and covered by a performance test,

Table 5.2: Overview of the selected mutants and covering tests that are involved in our experiment.

Mutation Operator	<i>RxJava</i>			<i>Eclipse-collections</i>			<i>Zipkin</i>			
	# Mutants	JMH	ju2jmh* Auto.†	# Mutants	JMH	ju2jmh* Auto.†	# Mutants	JMH	ju2jmh* Auto.†	
<b>PTW</b>	42	147	3,074	42	18	418	30	21	119	30
<b>STS</b>	2	0	205	2	11	55	2	0	31	2
<b>HWO</b>	9	193	337	9	9	1,226	2	6	21	2
<b>EFL</b>	8	121	573	8	8	174	6	11	33	6
<b>SOC</b>	5	0	10	5	3	132	0	0	0	0
<b>Total</b>	66	461	4,199	66	49	2,005	40	38	204	40

\* equals to the number of JUnit tests

† AutoJMH benchmarks that is equal to the number of Mutants

the testing results should vary, and the magnitude of the results should follow the same trend as the mutation triggered times.

### 5.3 Execution environment

To perform the execution procedure, we took advantage of cloud computing resources to simulate a real-world environment. To have a consistent measurement process, for all the experiments in this study, we deployed c2-standard-4 (4 vCPUs, 16 GB memory) instances provided by *Google Cloud*<sup>2</sup>. Instances are run on Debian GNU/Linux 10 (buster) and OpenJDK 1.8. In total, our experiment consists of 99,406 measurements, each containing 30 data points measured, i.e., each data point is the number of executions of one JMH microbenchmark case (annotated by @benchmark) in one second. It took  $\sim 1,656$  machine hours to complete all the experiments in this study.

---

<sup>2</sup><https://cloud.google.com/>

# Chapter 6

## Results

In this study, we aim to answer five research questions. For each RQ, we present the motivation to answer the RQ, our approach to addressing the RQ, and the corresponding results.

### **6.1 RQ1: How stable are automatically generated performance microbenchmarks?**

#### **6.1.1 Motivation**

When adopting performance microbenchmarking, the first challenge is assessing whether the benchmarks—be it JMH, AutoJMH, JUnit, or those generated by *ju2jmh*—are stable enough to reveal meaningful performance variations. Non-determinism has been repeatedly identified in recent studies as a significant hurdle, undermining the repeatability of measurements and the reliability of results. Without stability, microbenchmarks risk reporting misleading variations where none exist, rendering them ineffective. To address this, various methodologies and tools have been proposed to reduce non-determinism at different layers of abstraction [28, 17, 40]. Among these, JMH has emerged as a preferred choice for producing reliable performance metrics, while attempts to use JUnit tests for benchmarking often face persistent instability. As such, evaluating the stability of microbenchmarks is a critical first step in determining their quality and practical utility.

## 6.1.2 Approach

To address RQ1, we analyzed benchmarks with a focus on result variability. Each benchmark was executed individually in isolation, producing 30 iterations of results per measurement—consistent with recent practices [21].

To evaluate the stability of benchmarks, we followed a two-step process: the first compares benchmark stability across different benchmarks, while the second examines how quickly benchmarks reach a stable state (see Section 4.3.3 for further details).

**(1) Comparative Stability of Benchmarks.** In this step, we assessed the relative stability of benchmarks using the relative standard deviation (RSD) across the 30 iterations of results. Benchmarks with an RSD below 1% were deemed stable enough to detect performance bugs [47], while those exceeding 5% were categorized as unstable due to significant variability [49]. Benchmarks with RSD between 1% and 5% exhibited moderate variability: while they were unsuitable for detecting small-scale performance bugs, they could still identify larger-scale issues. However, given that most generated performance bugs are not large in scale, benchmarks within this range often fail to detect subtle bugs.

**(2) Stability Tendencies of Benchmarks.** Benchmarks that reach a stable state earlier provide actionable insights more efficiently. To analyze this, we developed a heuristic statistical method to determine which benchmarks stabilize first [46].

For each benchmark, we selected  $i$  data points from the 30 iterations to form an initial dataset ( $P_{original_i}$ ) and similarly selected another  $i$  data points to create a microbenchmarking dataset ( $P_{microbm_i}$ ). The selection size  $i$  varied from 2 to 15 ( $i \in 2, 3, \dots, 15$ ), and the process was repeated 100 times to generate 100 pairs of  $P_{original_i}, P_{microbm_i}$ . Following established methodologies [47], we applied bootstrap resampling [19] using 100 bootstrap iterations per pair, consistent with prior recommendations of 10,000 iterations [30].

From this, we calculated the relative confidence interval width (RCIW) for each benchmark with  $i$  iterations. The RCIW quantifies how variable a benchmark’s results are around its mean, offering a measure of its stability over time. This approach provides a nuanced understanding of how quickly and reliably benchmarks stabilize.

## 6.1.3 Results

**(1) Comparative Stability of Benchmarks.** Figure 6.1 is a box-plot chart representing the distribution of RSD calculated (in %) for all benchmarks, across three testing frameworks and three subjects.

According to Figure 6.1, the box-and-whisker of RSD of *ju2jmh* benchmarks range from

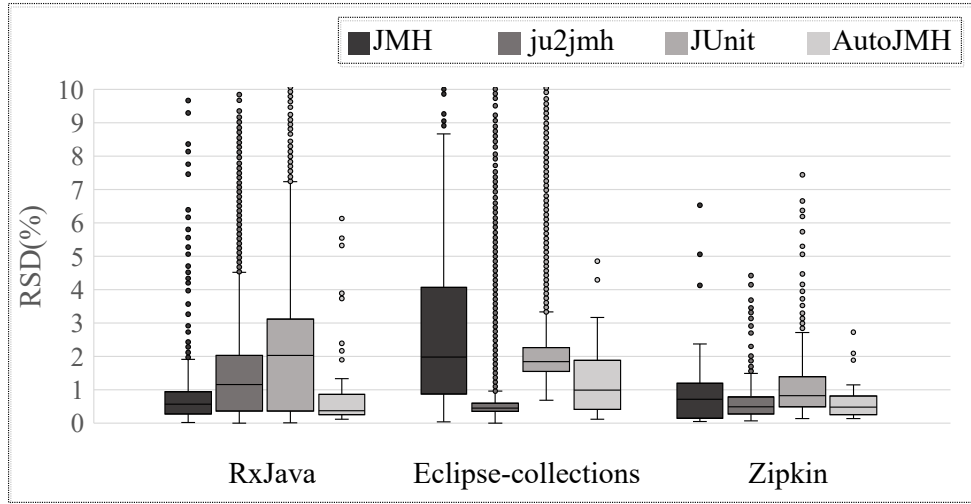


Figure 6.1: Distribution of the variability, using RSD (in %)

0% to 4.5% in `Rxjava` with a median of 1.2%, while the box-and-whisker of RSD of JUnit tests range from 0.0% to 7.2% with a median of 2.0% in the same subject. As we can see, JUnit tests have a higher median and maximum of RSD than JMH and AutoJMH benchmarks in `Rxjava`. `Zipkin` faces the same situation. For `Eclipse-collections`, JUnit tests have a higher median RSD (1.8%) than `ju2jmh` benchmarks (0.5%) and a higher maximum value in the box-and-whisker (3.3% vs. 1.0%), respectively. Because a larger RSD indicates that a benchmark is more unstable, these results imply that `ju2jmh` benchmarks are much more stable than JUnit tests in the three study subjects.

Likewise, we can also compare the stability of `ju2jmh` benchmarks and manually written JMH benchmarks based on Figure 6.1. The median RSD of JMH benchmarks (0.6%) in `Rxjava` is smaller than `ju2jmh` benchmarks (1.2%), however, it is greater than `ju2jmh` benchmarks in `Eclipse-collections` (2.0% vs. 0.5%) and `Zipkin` (0.7% vs. 0.5%), as shown in this figure. If only the `Eclipse-collections` and `Zipkin` are considered, it appears that `ju2jmh` benchmarks are more stable than manually written JMH benchmarks. However, when all three subjects are considered, we can only conclude that `ju2jmh` benchmarks beat JMH benchmarks in terms of stability in some subjects and not in others. As a result, `ju2jmh` benchmarks and manually written JMH benchmarks are comparable.

Lastly, the stability of `ju2jmh` benchmarks can be compared with AutoJMH benchmarks according to Figure 6.1. In terms of the median RSD, in `Rxjava`, AutoJMH benchmarks have a smaller value (0.4%) than `ju2jmh` benchmarks (1.2%). However, in `Eclipse-collections`, the median RSD of `ju2jmh` benchmarks (0.5%) is smaller than AutoJMH benchmarks

(1.0%), and the box-and-whisker length of *ju2jmh* benchmarks (0.96%) is also smaller than AutoJMH benchmarks (3.0%), indicating that most *ju2jmh* benchmarks are more stable than AutoJMH benchmark in this study subject. In [Zipkin](#), the stability of most *ju2jmh* benchmarks and most AutoJMH benchmarks are comparable because of the close median RSD values (0.5% vs. 0.5%). Therefore, the stability of *ju2jmh* benchmarks are comparable to AutoJMH benchmarks based on the evaluation of these three subjects.

In conclusion, based on the comparisons between automatically generated *ju2jmh* benchmarks and the other three microbenchmarks, we can infer that, although *ju2jmh* leverages JUnit tests to generate microbenchmarks, the *ju2jmh* microbenchmarks are more stable than the original JUnit tests and their stability is comparable to manually written JMH benchmarks and AutoJMH benchmarks.

In Table 6.1, we focus on whether benchmarks are stable or unstable for the purpose of detecting performance bugs. The table presents the proportion of benchmarks that are **stable** enough ( $RSD \leq 1\%$ ) or **unstable** ( $RSD \geq 5\%$ ).

Table 6.1: Stable and unstable benchmarks

Subject	Relative Standard Deviation							
	Stable ( $\leq 1\%$ )				Unstable ( $\geq 5\%$ )			
	JMH	ju2jmh	JUnit	Auto. <sup>†</sup>	JMH	ju2jmh	JUnit	Auto. <sup>†</sup>
RxJava	77.2%	43.8%	43.8%	81.1%	2.3%	3.9%	9.5%	7.5%
Ec-col.*	29.0%	91.5%	5.0%	51.0%	18.2%	1.7%	5.1%	6.1%
Zipkin	61.0%	84.6%	58.4%	82.5%	9.0%	0.0%	1.9%	0.0%

\* Eclipse-collections

<sup>†</sup> AutoJMH

In [Eclipse-collections](#) and [Zipkin](#), 91.5% and 84.6% of *ju2jmh* benchmarks are recognized as stable, significantly higher than 5.0% and 58.4% for JUnit tests, 29.0% and 61.0% for JMH benchmarks, and 51.0% and 82.5% for AutoJMH benchmarks. In [Rxjava](#), both *ju2jmh* and *JUnit* microbenchmarks produced an equal number of stable benchmarks, but 9.5% of the JUnit tests are recognized as unstable, higher than 3.9% for *ju2jmh* benchmarks. JMH benchmarks (77.2%) and AutoJMH benchmarks (81.1%) are more stable than *ju2jmh* benchmarks (43.8%), but in terms of unstable benchmarks, *ju2jmh* benchmarks and JMH benchmarks have very close values (3.9% vs. 2.3%) and *ju2jmh* benchmarks are better than AutoJMH benchmarks (3.9% vs. 7.5%). Based on these results, we can infer that *ju2jmh* benchmarks are generally more stable than JUnit tests and comparable to (or slightly more stable) manually written JMH benchmarks and AutoJMH benchmarks.

Furthermore, we also check whether JUnit tests remain stable after being converted to *ju2jmh* benchmarks. In [Rxjava](#), both *ju2jmh* and JUnit have the same proportion of stable tests. Specifically, 86.4% of JUnit tests that were recognized as stable remain stable after being converted to *ju2jmh* benchmarks, 0.9% of JUnit tests that were recognized as unstable become stable, and 0.6% of JUnit tests that were recognized as stable become unstable. For the unstable JUnit tests that become stable, we rerun the associated *ju2jmh* benchmarks without warm-up iterations, and all of the new results achieve a higher RSD (%) ranging from 0.2% to 7.1% than the results with warm-up iterations, thus the test stability has deteriorated in all cases. As a result, warming up the system before running the benchmark is critical to achieve the stability of *ju2jmh* benchmarks in comparison to JUnit tests without warm-up iterations. For the stable JUnit tests that become unstable, we manually checked the source code and found that all of these tests contain tasks that deal with asynchronous executions, multiple threads or multiple processors, which are restricted in *ju2jmh* benchmarks. Moreover, all the JUnit tests in [Eclipse-collections](#) and [Zipkin](#) remain stable in the form of *ju2jmh* benchmarks, none of the JUnit tests that were recognized as unstable become stable, and none of JUnit tests that were recognized as stable become unstable. In conclusion, a large portion of JUnit tests remain stable after being converted to *ju2jmh* benchmarks.

**(2) Stability Tendencies of Benchmarks.** Figure 6.2 presents the results of the second step, which discloses how benchmarks grow more stable as the number of iterations increases. Every box represents the distribution of calculated RCIWs for all benchmarks of a testing framework with  $i$  iterations. The greater the RCIW, the more variation (less stability) there is in the result set. Accordingly, the larger the box-and-whisker, the more variation of stability there is between benchmarks of one type. For example, according to Figure 6.2, a benchmark usually produces more variable results through two iterations than through a larger number of trials, such as 15 iterations. It is to be noted that, to facilitate a better comparison and clarification, we zoom in the chart so that a few whiskers and one box are shown out of the presenting range, but this does not affect the conclusion.

As illustrated in Figure 6.2, the median RCIW of JUnit tests is always greater than *ju2jmh*, JMH, and AutoJMH benchmarks, implying that JUnit tests are generally more unstable than the other three types of microbenchmarks. Furthermore, based on the following analysis, we can conclude that JMH, *ju2jmh*, and AutoJMH RCIWs are comparable among these three study subjects. The median RCIW of *ju2jmh* benchmarks is always greater than JMH in [Rxjava](#) and [Zipkin](#), but their differences are significantly smaller than the RCIW differences between JUnit tests and JMH/*ju2jmh* benchmarks, indicating that although *ju2jmh* benchmarks are more unstable than JMH benchmarks in these two subjects, their difference is very limited. [Rxjava](#) presents the same conclusions when *ju2jmh*

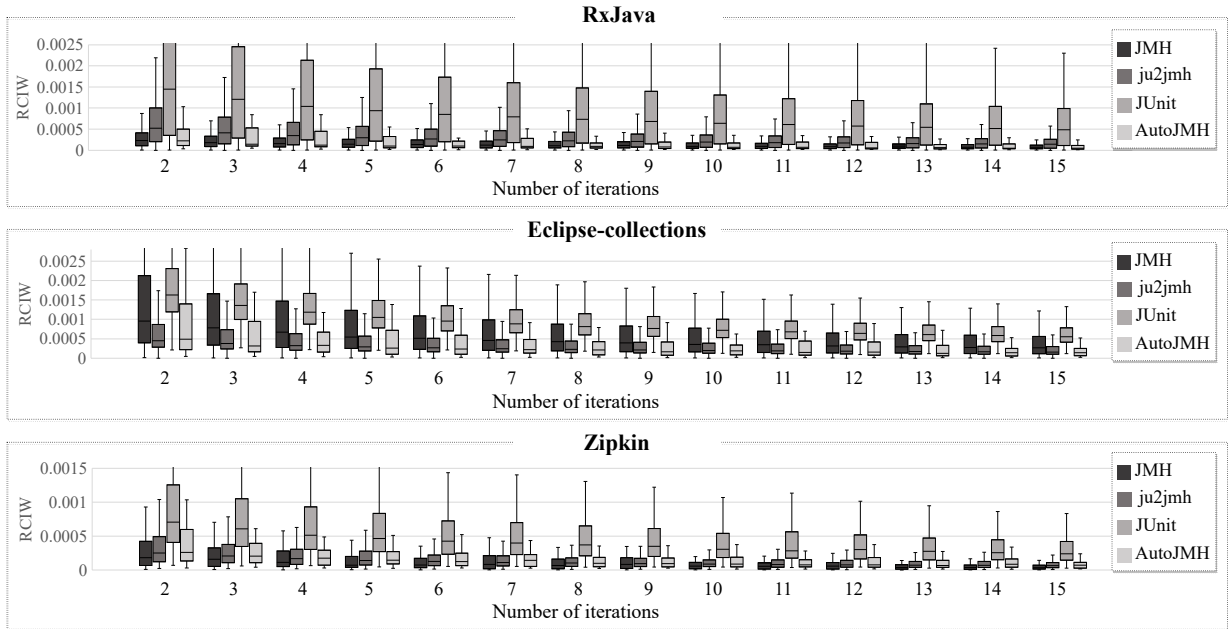


Figure 6.2: The box-plots of calculated RCIW for all benchmarks of a testing framework, ranging the iteration number  $i$ ,  $i \in \{2, 3, \dots, 15\}$

is compared with AutoJMH. In addition, the median RCIW of *ju2jmh* benchmarks is lower than JMH and JUnit in [Eclipse-collections](#), suggesting that those benchmarks are more unstable than *ju2jmh* benchmarks in this subject. Moreover, the median RCIW of *ju2jmh* benchmarks is always comparable with AutoJMH benchmarks in the two subjects of [Eclipse-collections](#) and [Zipkin](#). Lastly, in all three subjects, *ju2jmh* benchmarks' RCIWs are converging with JMH and AutoJMH benchmarks' RCIWs to a higher level of stability, with a very similar level of stability after 15 iterations.

### 6.1.4 Conclusion

In conclusion, regarding stability assessment, both steps yield identical results: (1) in all three study subjects, the generated *ju2jmh* benchmarks outperform JUnit tests; (2) *ju2jmh* and manually written JMH benchmarks are comparable, but *ju2jmh* benchmarks outperform manually written JMH benchmarks in two out of three study subjects. (3) *ju2jmh* benchmarks are also comparable to AutoJMH benchmarks.

## 6.2 RQ2: Can performance microbenchmarks detect artificial performance bugs from mutation testing?

### 6.2.1 Motivation

While executing performance microbenchmarking, a performance bug can make developers question whether it comes from a noisy environment [46, 49], the benchmark’s unstable quality [40], or even a bug. In RQ1, we assessed the stability of benchmarks. However, the stability itself is not sufficient for detecting performance bugs. A further evaluation of benchmarks is needed concerning their ability to detect performance bugs.

### 6.2.2 Approach

To address benchmarks’ ability to detect bugs, we leverage the *PMT* framework presented in Section 4.3.4 to inject artificial performance bugs into the source code. An artificial performance bug could be observed by a covering performance test by searching for degradations in its results. If the performance test can detect an artificial bug, it is more likely that the test can also detect a similar real-world performance bug.

First, to generate artificial performance bugs, we apply the *PMT* framework to the subject’s source code. We build up a large number of systems versions, each containing one performance bug injected at a single source code location. Then, we extract all benchmarks that cover any of the selected mutants. Afterwards, we execute benchmarks that cover an injected performance bug, before and after injection, 30 times. Benchmarks are run in a sequence within an isolated and controlled environment.

Employing the technique of performance mutation testing, which deliberately introduces performance bugs, we can look at the benchmarks’ results to find any significant difference. Recent studies [21, 44] have advocated using *hypothesis statistical tests* (e.g., Wilcoxon rank-sum test) to label a significant difference in results as a performance bug, slowdown, or artificial bug. However, the prior study [46] concludes that testing with Wilcoxon rank-sum tests, due to high false-positive reported, is not a suitable vehicle for detecting performance degradation in cloud environments.

In order to accurately assess the difference between the microbenchmarking results before and after bug injection, we use mutation scores (i.e., the percentages of killed mutants defined in Section 4.3.3) to compare the analyzed subjects. We first assume that a benchmark can only kill a mutant if the throughput of the benchmark with the mutant is

significantly lower than the throughput in the original system. For a studied benchmark, we estimate its **confidence interval for the ratios of means** (RCI), and the ratios are calculated based on the benchmark’s throughput (from before and after the bug injection) and by deploying the bootstrap technique [68, 19], using 10,000 bootstrap iterations [30] with a confidence level of 99%.

Therefore, the size of performance degradation caused by a performance bug can be defined as

$$bug\_size = [1 - U_{RCI}]$$

, where  $U_{RCI}$  is the upper bound of the estimated RCI. The upper bound of the estimated RCI, opposite to its lower bound, denotes the minimum bug occurring in the target system. We calculate the bug size in % and it reflects the effectiveness of mutant against the benchmark. We then assume **three different thresholds** as the minimum performance degradation that bugs produce, namely 1%, 5%, 10%, which has been used in prior study [21]. The thresholds are chosen from 1% to 10% because: (1) According to the preliminary analysis results of PMT in Table 4.2, bug size in the mutation operators SOC and HWO is always greater than 10%, which is why we set a 10% threshold for them. Although PTW, STS, and EFL mutation operators can/may achieve a bug size of 10% as the `hitting_ratio` increases by a large value, for consistency among different mutation operators, we still keep 10% as the upper bound for all mutations. (2) In terms of selecting the lowest threshold, 1% is the lowest threshold used in the prior work [21], and the bugs with bug sizes of less than 1% are difficult to detect. The three thresholds are chosen from 1% with a high falsely reported positive (bugs not caused by mutants) to 10% with a low falsely reported positive. If a bug size is greater than the threshold, there is a significant variation in results, which conveys that the benchmark could kill the mutant (detect the injected bug), and vice versa. The mutation score is then calculated to compare *ju2jmh* benchmarks to manually written JMH benchmarks, JUnit tests, and AutoJMH benchmarks.

### 6.2.3 Results

Table 6.2 presents the percentage of tests that kill any of the mutants and Table 6.3 presents the calculated mutation score over the three thresholds (columns **1%**, **5%** and **10%**), for the four testing frameworks (column **FW**) and the five mutation types (columns **PTW(%)**, **STS(%)**, **EFL(%)**, **SOC(%)** and **HWO(%)**).

**Comparing *ju2jmh* benchmarks and JUnit tests:** According to column **Total (%)** in Table 6.2, across the three subjects and for all three thresholds, more *ju2jmh* benchmarks kill a mutant than JUnit tests. In total, from 4.2% to 34.6% of all *ju2jmh* benchmarks kill a mutant, significantly higher than 3.5% to 15.7% for JUnit. Furthermore, over large

Table 6.2: The percentage of tests that kill any of the mutants from five operators, assuming that the mutant is killed if the *bug\_size*  $\geq$  1%, 5%, and 10%.

Subject	FW	PTW (%)			STS (%)			EFL (%)			SOC (%)			HWO (%)			Total (%)		
		1%	5%	10%	1%	5%	10%	1%	5%	10%	1%	5%	10%	1%	5%	10%	1%	5%	10%
RxJava	JMH	34.6	6.1	0.6	-	-	-	37.1	8.2	3.3	-	-	-	55.4	41.4	35.7	44.0	21.4	16.0
	ju2jmh	29.2	3.0	0.6	40.0	12.1	2.9	44.4	6.4	2.7	20.0	0.0	0.0	56.3	42.1	39.4	34.0	7.0	4.1
	JUnit	6.5	0.7	0.1	21.4	4.3	0.9	10.8	3.3	0.3	0.0	0.0	0.0	42.1	39.4	38.5	10.6	4.3	3.2
	AutoJMH	64.2	11.9	0.0	100	100	100	50.0	25.0	12.5	20.0	0.0	0.0	66.7	44.4	33.3	64.7	19.6	11.8
Ec-col.*	JMH	21.4	8.2	4.1	-	-	-	-	-	-	0.0	0.0	0.0	54.5	50.0	50.0	28.9	18.4	15.6
	ju2jmh	31.5	8.3	1.6	33.9	9.4	5.6	37.3	12.6	6.8	43.1	11.3	3.7	37.0	9.2	4.8	37.1	9.4	4.3
	JUnit	20.3	5.5	1.1	35.8	11.3	7.5	36.7	13.2	6.8	40.9	10.6	3.7	18.2	7.3	4.2	25.2	7.8	3.9
	AutoJMH	66.7	16.7	5.6	54.5	18.1	9.1	62.5	25.0	12.5	100	66.7	33.3	66.7	66.7	66.7	66.7	33.3	21.2
Zipkin	JMH	19.0	4.7	0.0	-	-	-	9.0	9.0	9.0	-	-	-	33.3	33.3	33.3	18.4	7.8	5.2
	ju2jmh	12.6	2.5	0.8	25.8	16.1	0.0	30.3	0.0	0.0	-	-	-	47.6	33.3	33.3	21.0	7.3	3.9
	JUnit	11.7	2.5	0.8	25.8	16.1	0.0	0.0	0.0	0.0	-	-	-	42.8	33.3	33.3	16.1	7.3	3.9
	AutoJMH	36.7	10.0	3.3	100	100	0.0	50.0	0.0	0.0	-	-	-	100	100	100	43.2	16.2	8.1
Total	JMH	28.3	9.3	2.4	-	-	-	35.3	8.3	3.8	0.0	0.0	0.0	54.9	43.0	38.5	38.8	20.1	15.5
	ju2jmh	29.0	3.7	0.8	37.6	12.4	3.4	42.3	7.7	3.7	42.0	10.5	4.2	42.5	16.8	12.7	34.6	7.9	4.2
	JUnit	8.7	1.4	0.3	25.5	7.2	2.4	16.9	5.5	1.9	38.5	10.5	4.2	27.4	14.6	12.0	15.7	5.6	3.5
	AutoJMH	55.6	12.2	2.2	66.7	40.0	20.0	54.5	18.1	9.1	50.0	25.0	12.5	70.0	60.0	55.0	58.7	22.3	13.2
	<b>Total (%)</b>	19.5	2.7	0.5	32.1	10.2	2.9	30.2	6.6	2.7	38.9	10.3	3.6	32.9	15.0	11.7	25.8	7.4	4.4

\* Eclipse-collections

degradations ( $\geq 10\%$ ), low and close percentages of *ju2jmh* benchmarks and JUnit tests kill a mutant, while in small degradations ( $\geq 1\%$ ), *ju2jmh* significantly outperforms JUnit.

Furthermore, according to column **Total (%)** in Table 6.3, *ju2jmh* achieves a higher mutation score than JUnit, with the exception of [Eclipse-collections](#) at the threshold of 1%. In total, *ju2jmh* achieves a mutation score from 29.7% to 80.0%, higher than JUnit with a mutation score of 23.9% to 69.0%. In [Eclipse-collections](#) and over the threshold of 1%, the difference between *ju2jmh* benchmarks and JUnit tests is slight (87.8 vs. 91.8). In other cases, *ju2jmh* outperforms JUnit.

In conclusion, while both *ju2jmh* and JUnit tests cover all of the mutants in the study, the percentage of *ju2jmh* benchmarks that kill a mutant is higher than JUnit tests, implying that for a given mutant operator, it is more likely to be killed by *ju2jmh* benchmarks than JUnit tests. In addition, *ju2jmh* benchmarks generally achieve higher mutation scores than JUnit tests. Therefore, *ju2jmh* is more effective in detecting performance bugs, regardless of the analysis perspective of mutation operators or tests.

**Comparing *ju2jmh* and JMh benchmarks:** We first compare the percentage of *ju2jmh* and JMh benchmarks that kill a mutant using the analysis of column **Total (%)** in Table 6.2. In some cases, the percentage of *ju2jmh* benchmarks is greater than that of JMh benchmarks, while in other cases, it is lower. For example, 37.1% and 21.0% of *ju2jmh* benchmarks in [Eclipse-collections](#) and [Zipkin](#) kill a mutant with bug size  $\geq 1\%$ , higher than the 28.9% and 18.4% of JMh benchmarks. However, in other cases, *ju2jmh*

Table 6.3: Mutation score of tests against the generated mutants from five operators, assuming that the mutant is killed if the *bug\_size*  $\geq$  1%, 5%, and 10%.

Subject	FW	PTW (%)			STS (%)			EFL (%)			SOC (%)			HWO (%)			Total (%)			Coverage
		1%	5%	10%	1%	5%	10%	1%	5%	10%	1%	5%	10%	1%	5%	10%	1%	5%	10%	
RxJava	JMH	100	42.8	14.3	-	-	-	100	100	50.0	-	-	-	100	100	100	100	73.3	46.7	15 (22.7%)
	ju2jmh	95.2	50.0	16.7	100	100	100	100	87.5	75.0	40.0	0.0	0.0	100	88.9	88.9	92.4	57.6	34.8	66 (100%)
	JUnit	64.3	19.0	9.5	100	100	100	100	75.0	25.0	0.0	0.0	0.0	88.9	88.9	88.9	68.2	36.4	24.2	66 (100%)
	AutoJMH	64.2	11.9	0.0	100	100	100	50.0	25.0	12.5	20.0	0.0	0.0	66.7	44.4	33.3	60.6	19.7	9.1	66 (100%)
Ec-col.*	JMH	71.4	42.9	28.6	-	-	-	-	-	-	0.0	0.0	0.0	100	100	100	69.2	53.8	46.1	13 (26.6%)
	ju2jmh	94.4	61.1	22.2	81.8	36.4	27.3	62.5	62.5	50.0	100	100	100	100	77.8	66.7	87.8	61.2	40.8	49 (100%)
	JUnit	94.4	38.9	11.1	90.9	45.4	27.2	75.0	62.5	50.0	100	100	100	100	77.8	66.7	91.8	55.1	36.7	49 (100%)
	AutoJMH	66.7	16.7	5.6	54.5	18.1	9.1	62.5	25.0	12.5	100	66.7	33.3	66.7	66.7	66.7	65.3	30.6	20.4	49 (100%)
Zipkin	JMH	16.7	8.3	0.0	-	-	-	25.0	0.0	0.0	-	-	-	50.0	50.0	50.0	22.2	11.1	5.6	18 (45.0%)
	ju2jmh	40.0	10.0	3.3	100	100	0.0	66.7	0.0	0.0	-	-	-	100	100	100	50.0	17.5	7.5	40 (100%)
	JUnit	36.7	10.0	3.3	100	100	0.0	33.3	0.0	0.0	-	-	-	100	100	100	42.5	17.5	7.5	40 (100%)
	AutoJMH	36.7	10.0	3.3	100	100	0.0	50.0	0.0	0.0	-	-	-	100	100	100	45.0	17.5	7.5	40 (100%)
Total	JMH	53.8	26.9	11.5	-	-	-	62.5	50.0	25.0	0.0	0.0	0.0	90.0	90.0	90.0	60.9	43.5	30.4	46 (29.7%)
	ju2jmh	76.7	38.9	13.3	86.7	53.3	33.3	77.2	54.5	45.4	62.5	37.5	37.5	100	85.0	80.0	80.0	48.4	29.7	155 (100%)
	JUnit	61.1	20.0	7.8	93.3	60.0	33.3	72.7	50.0	27.2	37.5	37.5	37.5	95.0	85.0	80.0	69.0	37.4	23.9	155 (100%)
	AutoJMH	55.6	12.2	2.2	66.7	40.0	20.0	54.5	18.1	9.1	50.0	25.0	12.5	70.0	60.0	55.0	58.0	22.6	12.2	155 (100%)
<b>Total (%)</b>		83.3	46.7	17.8	93.3	73.3	46.7	100	59.1	50.0	62.5	37.5	37.5	100	95.0	90.0	89.0	56.8	35.5	155 (100%)

\* Eclipse-collections

benchmarks achieve a smaller percentage.

We then compare the mutation scores of *ju2jmh* and JMH benchmarks according to Table 6.3. Column **Total (%)** illustrates that *ju2jmh* benchmarks are more efficient than JMH benchmarks in killing mutants in general. On average, 29.7% to 80.0% of mutants are killed by *ju2jmh* benchmarks, higher than 30.4% to 60.9% by JMH benchmarks. In more than half of the cases, *ju2jmh* benchmarks achieve higher mutation scores than JMH benchmarks. Moreover, although manually written JMH benchmarks are not far behind *ju2jmh* benchmarks in terms of mutation scores, they cover far fewer mutants (29.7% vs. 100%).

As presented in Table 6.2 and Table 6.3, although there is a higher percentage of JMH benchmarks that kill mutants than *ju2jmh* benchmarks, the overall coverage of all JMH benchmarks is much less than *ju2jmh*, implying that *ju2jmh* benchmarks have more coverage diversity, whereas manually written JMH benchmarks are more specific to certain mutants with less coverage diversity. The manually written JMH benchmarks would produce better results if they could achieve higher coverage. However, this necessitates a great deal of manual effort, which is impractical in real-life development. Therefore, we can deduce that *ju2jmh* benchmarks cover more mutants than manually written JMH benchmarks, and that the JMH benchmarks necessitate mutant coverage enhancements in order to achieve better results.

**Comparing AutoJMH benchmarks with other tests:** Similarly, when the percentage of AutoJMH benchmarks that kill a mutant is compared to those of other tests, AutoJMH

benchmarks can have higher percentages than *ju2jmh* benchmarks, according to column **Total (%)** in Table 6.2. However, both benchmarks can generally achieve 100% mutant coverage, as presented in Table 6.3. In addition, AutoJMH benchmarks have limitations that prevent them from being trusted and valuable. First, according to Table 5.2, the number of generated AutoJMH benchmarks is significantly lower than the other tests, resulting in a limited data set for study. Second, AutoJMH is obsolete as we mentioned in Section 4.5. Third, AutoJMH is limited to manually configuring specific single statements. Such reasons limit the spread and application of AutoJMH.

Moreover, *ju2jmh* benchmarks achieve a higher mutation score than AutoJMH benchmarks in all cases across the three subjects and for all three thresholds, according to column **Total (%)** in Table 6.3. In total, *ju2jmh* achieves a mutation score of 29.7% to 80.0%, significantly higher than 12.2% to 58.0% for AutoJMH. In particular, in [Rxjava](#) and [Eclipse-collections](#), *ju2jmh* benchmarks have mutation scores of at least 20.4% higher than AutoJMH benchmarks.

In conclusion, while both *ju2jmh* and AutoJMH benchmarks cover all mutants, *ju2jmh* benchmarks achieve higher mutation scores than AutoJMH benchmarks, indicating that *ju2jmh* benchmarks are more effective than AutoJMH benchmarks in detecting performance bugs.

**Different effects of mutation operators’ diversity:** According to row **Total (%)** in Table 6.2, which presents the percentage of benchmarks that kill a mutant, across the threshold of 1%, 61.2% of JMH benchmarks, 65.4% of *ju2jmh* benchmarks, 84.3% of JUnit tests, and 41.3% of AutoJMH benchmarks failed in killing mutants. The highest percentage of benchmarks kill mutants from SOC (see the total mutation score in column **SOC**) with the percentage of 3.6% to 38.9%, and the smallest percentage of benchmarks kill mutants from PTW (column **PTW**) with the percentage of 0.5% to 19.5%. Moreover, mutants from HWO (column **HWO**) have a strong effect on benchmarks, particularly where 11.7% of benchmarks with a large bug size 10% is significantly greater than 0.5% of benchmarks that kill a mutant from PTW.

Table 6.3 yields the same finding, proving that different mutation operators have different effects on microbenchmarks. According to row **Total (%)**, which presents the calculated mutation score across all tests of the three study subjects, 35.5% to 89.0% of mutants are killed by the tests in total. The highest mutation score is achieved by HWO (column **HWO**) where 90.0% to 100% of mutants are killed. The smallest mutation score is for SOC (column **SOC**) where 37.5% to 62.5% of mutants are killed. Such large variations in the results illustrate the diversity of mutant operators.

**Exclusive mutation score:** In order to provide more insight into *ju2jmh* benchmarks, we

define the *exclusive mutation score*, which is the percentage of mutants that are exclusively killed by a framework and not by others. We calculate the exclusive mutation score for the four frameworks, across the three subjects, for the five mutation operators, and over the three thresholds. In total, over the threshold of 1%, *ju2jmh* benchmarks achieve an exclusive mutation score of 3.2%, higher than 2.2% for JMH benchmarks and 0.6% for JUnit tests, but less than 6.5% for AutoJMH benchmarks. Over the threshold of 5%, *ju2jmh* benchmarks achieve an exclusive mutation score of 9.0%, considerably higher than 2.2% for JMH, 0.6% for JUnit, and 6.5% for AutoJMH. Over the large threshold of 10%, *ju2jmh* benchmarks achieve an exclusive mutation score of 5.8%, higher than 0.6% for JUnit and 3.6% for AutoJMH, but less than 8.7% for JMH. In conclusion, most mutants are covered by multiple tests, with *ju2jmh* benchmarks covering more mutants exclusively than JUnit tests in all cases and JMH and AutoJMH benchmarks in the majority of cases.

## 6.2.4 Conclusion

*ju2jmh* benchmarks are more effective in detecting performance bugs than JUnit tests and AutoJMH benchmarks. *ju2jmh* benchmarks cover more mutants than manually written JMH benchmarks and JMH benchmarks necessitate mutant coverage enhancements to achieve better results. Furthermore, we discover that different mutation operators can have various effects on benchmarks. In general, *ju2jmh* benchmarks can detect a larger proportion of mutants exclusively than other tests.

## 6.3 RQ3: Can performance microbenchmarks detect real-world performance bugs?

### 6.3.1 Motivation

One of the advantages of mutation testing is its ability in uncovering deficiencies in tests and improving them, allowing the improved tests to be further utilized for detecting real-world bugs [66]. We are motivated to know if benchmarks with higher performance mutation score can detect real-world performance bugs better. Additionally, we wish to know how similarly the generated mutants and real-world performance bugs behave in affecting a covering performance test.

### 6.3.2 Approach

To answer the motivation questions, we check whether a performance test is more likely to detect a similar real-world performance bug if it is able to detect a mutant.

For a specific real performance bug inside a system’s source code, we first find all JUnit tests and JMH benchmarks that cover it. Next, we deploy *ju2jmh* to build benchmarks from the JUnit tests and deploy the PMT tool to generate a mutant similar to the bug on the same source code location. Last, we execute the covering *ju2jmh* and JMH benchmarks 30 times against the original system where the bug is fixed, the parent system that still contains the bug, and the system where the bug is replaced with a generated mutant. Then, we compare mutation results with real bug results. Accordingly, the efficacy of performance tests in detecting a mutant is compared with their efficacy in detecting real bugs.

Similarly to the previous research question, we compute the RCI for both the original system and the system containing real-world bugs ( $RCI_{bug}$ ) and compute the RCI for the original system and its mutant version ( $RCI_{mutant}$ ). Then, we compare both calculated RCIs to find similarities between them.

The explored real-world performance bugs are extracted from the NFBugs dataset [67]. *NFBugs* is a dataset of 138 non-functional bug fixes in 67 open-source projects in Java or Python. NFBugs contains eight common performance bug patterns from performance bugs that are already fixed by the projects’ communities. The building of our PMT tool is based on the real-world performance bugs collected from the NFBugs dataset.

In total, there are 13 fixes of real-world performance bugs from 11 different projects in the NFBugs dataset that are supported in our PMT tool (i.e., the bug can be reproduced by one of the five developed patterns). The 13 bug fixes are from the three bug patterns of PTW, EFL, and STS. However, only one bug fix can be studied through this research question’s experiment. For the remaining 12 bug fixes, in 11 of them, there is either no JMH/JUnit test in the system or there is no JMH/JUnit test that covers the fixed bugs, and for the last bug fix, the corresponding system is no longer publicly available.

The performance bug exists in [Storio](#)<sup>1</sup> and is fixed through commit #566d3e9. There are 21 JUnit tests in the system that cover the bug. Accordingly, we build 21 *ju2jmh* microbenchmarks and a mutant associated with the bug. Lastly, we compare the mutation score of the *ju2jmh* benchmarks and the JUnit tests.

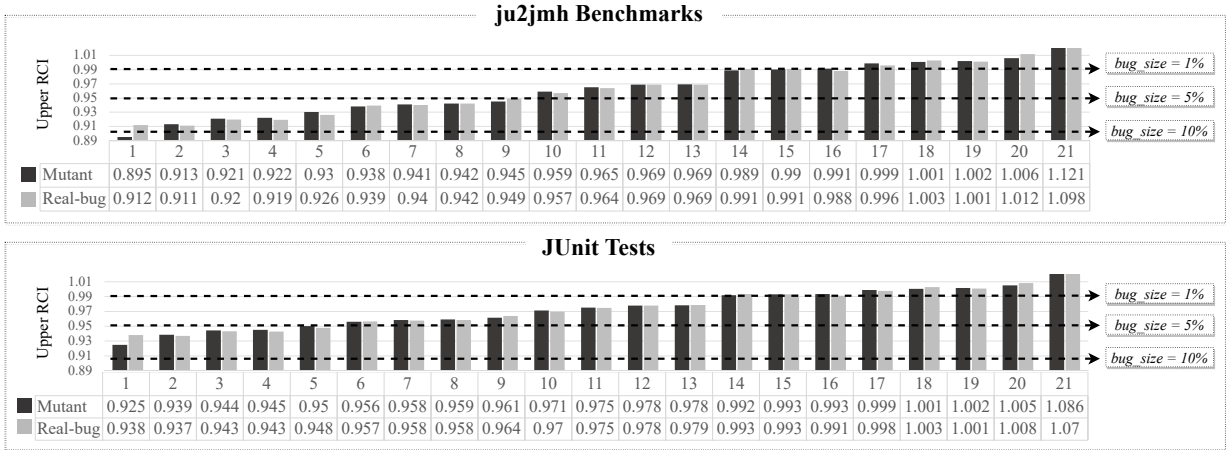


Figure 6.3: The calculated  $U_{RCI}$  for each of 21 *ju2jmh* benchmarks and 21 JUnit tests, against the mutant and the real bug.

### 6.3.3 Results

Figure 6.3 presents the calculated  $U_{RCI}$  for each of 21 numbered *ju2jmh* benchmarks and the corresponding 21 JUnit tests, against the mutant and the real bug. Similarly to the previous research question, three thresholds (1%, 5% and 10%) are marked as the three minimum *bug-sizes* that trigger the detection of bug. If the calculated  $U_{RCI}$  is below a given threshold’s line, we conclude that the degradation is larger than the minimum *bug-size*. Table 6.4 summarizes Figure 6.3.

Table 6.4: Microbenchmarks that kill mutant and/or detect performance bug, assuming that the mutant is killed or the bug is detected if the *bug-size*  $\geq$  1%, 5%, and 10%.

ju2jmh						JUnit					
Mutant			Bug			Mutant			Bug		
1%	5%	10%	1%	5%	10%	1%	5%	10%	1%	5%	10%
66.7	42.8	4.7	66.7	42.8	0.0	61.9	19.0	0.0	61.9	23.8	0.0

According to Figure 6.3 and Table 6.4, *ju2jmh* appears to perform better than JUnit both in killing the mutant and in detecting the real-world bug. For the three thresholds of bug sizes 1%, 5% and 10%, the percentage of *ju2jmh* benchmarks that kill the mutants is 4.7% (benchmark #1), 42.9% (benchmarks #1 to #9), and 66.7% (benchmarks #1 to

<sup>1</sup><https://github.com/pushtorefresh/storio.git>

#14) respectively, better than JUnit tests with the percentage of 0.0%, 19.0% (benchmarks #1 to #4), and 61.9% (benchmarks #1 to #13), indicating that *ju2jmh* benchmarks can detect the mutant better than JUnit tests. Similarly, *ju2jmh* performs better than JUnit in detecting real-world bugs. Over smaller degradations (1% and 5%), 66.6% and 42.8% of *ju2jmh* microbenchmarks detect the presence of the bug, higher than 61.9% and 23.8% for JUnit tests. None of the *ju2jmh* or JUnit tests can detect real-world bugs with a larger bug size threshold (10%). As a result, *ju2jmh* outperforms JUnit both in killing the mutant and in detecting the bug.

According to Figure 6.3, both *ju2jmh* benchmarks and JUnit tests produce results with a similar distribution against the mutant and the bug. On average, the difference of calculated  $U_{RCI}$  between the mutant and the bug is 0.4% for *ju2jmh* benchmarks and 0.3% for JUnit tests, showing the similarities between results of benchmarks against the mutant and results of benchmarks against the bug. However, in all cases, the efficacy of *ju2jmh* benchmarks in killing the mutant or detecting the bug is higher than that of JUnit tests. *ju2jmh* benchmarks achieve a 0.3% to 3.0% lower value of the calculated  $U_{RCI}$  (higher efficacy) than JUnit tests in killing the mutant and a 0.2% to 2.6% lower value in detecting the bug. In conclusion, the distribution of *ju2jmh* benchmarks’ results and JUnit test results is similar, but the efficacy of *ju2jmh* benchmarks both in killing the mutant and in detecting the bug is higher than JUnit tests.

### 6.3.4 Conclusion

According to our experiment of a real-world performance bug and a similar generated mutant, *ju2jmh* performs better than JUnit in killing the mutant and detecting the bug. Furthermore, the mutant and the real-world bug affect benchmarks in a very similar way.

## 6.4 RQ4: What are the major factors affecting a microbenchmark’s ability to detect performance bugs?

### 6.4.1 Motivation

To determine whether a performance microbenchmark is effective or unsatisfactory, there are a number of metrics that require further investigation and comparison. In this section, we highlight three causes that significantly affect the ability of microbenchmarks to detect bugs. The causes **c1** and **c3** in the following are derived from prior research [22]. We examine three of the eight causes they identified and merge two of them into **c1**. In

addition, **c2** is derived based on both the prior work [44] and the aforementioned analysis of RQ results.

**c1. Too low workload:** performance bugs could not be detected in benchmarks with a modest workload of payloads. In other words, most of mutants are killed by the benchmarks with a relatively large workload.

[22] point out that not enough microbenchmark execution repetitions could have a significant impact on a performance microbenchmark’s ability to find bugs. During a microbenchmark execution, if we raise the number of iterations of the microbenchmark code, the workload can be increased without exceeding the hardware limit. As a result, we have combined these two reasons into one.

**c2. Unstable microbenchmarks:** [44] reveal that the variability of a benchmark has an impact on benchmarking results. Furthermore, both RQ1 and RQ2 indicate that *ju2jmh* benchmarks outperform JUnit tests in all study subjects and are comparable to JMH benchmarks when two separate aspects are taken into account (i.e., microbenchmark stability and their ability to detect bugs). The same conclusions may imply that microbenchmark stability affects microbenchmark ability to detect bugs, which could be of interest.

**c3. Limited mutant coverage:** performance bugs are generated by the *PMT* framework, however they only affect one point in the source code (mostly one line of source code). Therefore, to investigate this cause, we considered the total number of times that the source code line containing bugs (mutants) was hit during the microbenchmark execution as a coverage metric. We argue that the more times a benchmark hits the buggy line, the more significant a performance degradation will be detectable in the microbenchmarking results.

**Other causes:** there are other known/unknown causes for future study that might impact benchmarks in exposing bugs. For example, benchmarks accessing IO/network/other resources [22], natural internal and external noises [46, 49], partial branch coverage [22], and the design of the benchmark itself [18]. These other causes are not evaluated here, and are left for future research.

## 6.4.2 Approach

To answer RQ4, we examine each of the three aforementioned causes against all of RQ2’s benchmarks that killed a mutant. Since there were not many benchmarks that killed a mutant with a bug size  $\geq 5\%$  and  $10\%$ , we only studied the threshold of  $1\%$  as the minimum bug size.

To examine workload (**c1**), we take the *throughput (ops/s)* of benchmarks as the metric. In addition, we take the *RSD (%)* of benchmarks and the *hitting\_count (/s)* to study the second (**c2**) and third (**c3**) causes respectively. For each metric (i.e., *throughput/RSD/hitting\_ratio*), we calculate the number of benchmarks that could kill the mutant with different values of metrics for subsequent analysis. Specifically, we split the range of all measurements (of a testing framework and across all five mutant types) into four equal-size groups from the *minimum* value to the *maximum* value in the data-set. Then, we count the total number of benchmarks that kill a mutant in each group.

In prior RQs, we compare *ju2jmh* to other testing frameworks and find that *ju2jmh* outperforms other testing frameworks in general. Therefore, the question of why *ju2jmh* outperforms other testing frameworks arises. The *ju2jmh* benchmarks achieve 100% mutant coverage, which means that each mutant is executed. For mutants merely covered by *ju2jmh* benchmarks, it is obvious that *ju2jmh* benchmarks perform better due to mutant coverage. For mutants covered by multiple tests, we perform a study. We first select all mutants that are covered by all four types of tests. We choose the top mutant cases where *ju2jmh* benchmarks outperform other tests the most and conduct a manual analysis on them. For each of the five mutation types, we extract a ranking list of the mutants for which *ju2jmh* outperforms the other three frameworks in terms of killing the mutants. The ranking is determined by the largest *bug\_size* difference between *ju2jmh* benchmarks and the tests from the other three frameworks. If a mutant is covered by multiple tests that are specific to a given test type, we choose the best result as a representation of this test type. The top five mutants with the largest *bug\_size* difference between *ju2jmh* benchmarks and the other tests are then selected. Lastly, we manually inspect the top five cases with the help of techniques, such as static analysis, to determine why *ju2jmh* benchmarks outperform other tests.

Similarly, we follow the procedure from the manual analysis above to study the reasons why the mutants can only be killed by *ju2jmh* benchmarks but not by other tests. We first extract all mutants that can only be killed by *ju2jmh* benchmarks with the three different thresholds for throughput reduction (1%, 5% and 10%). For each threshold and mutation operator type, after ranking the *ju2jmh* benchmarks by comparing the maximum difference of *bug\_size* between them and the other tests, we choose the top cases to perform the manual analysis.

### 6.4.3 Results

Table 6.5 presents the number of microbenchmarks that killed a mutant across each of the four groups of the three causes.

Table 6.5: The number of tests that killed any of the mutants, across the four groups of the three causes, for the three testing frameworks, and across the three study subjects.

Subject	Framework	c1: too low workload				c2: unstable tests				c3: limited coverage				Total
		g1	g2	g3	g4	g1	g2	g3	g4	g1	g2	g3	g4	
<b>RxJava</b>	JMH	38	50	82	33	195	2	3	3	29	52	91	31	203
	ju2jmh	32	115	193	1,088	1,328	83	7	4	79	870	424	55	1,428
	JUnit	14	34	100	319	453	8	3	3	18	244	160	45	467
	AutoJMH	3	1	4	32	35	1	0	4	3	2	28	7	40
<b>Ec-col.*</b>	JMH	23	23	2	2	40	5	4	1	2	22	18	8	50
	ju2jmh	10	15	63	653	717	18	5	1	5	390	335	11	741
	JUnit	9	10	42	441	485	12	4	1	4	149	336	13	502
	AutoJMH	6	0	2	24	31	0	0	1	14	11	4	3	32
<b>Zipkin</b>	JMH	6	0	0	1	5	0	1	1	4	2	0	1	7
	ju2jmh	1	4	9	29	39	3	0	1	2	12	12	17	43
	JUnit	1	1	12	19	31	1	0	1	6	5	9	13	33
	AutoJMH	2	0	6	10	11	5	0	2	1	7	4	6	18
<b>Total</b>	145 (4.1%)	253 (7.1%)	461 (12.9%)	2,651 (74.4%)	3,370 (94.6%)	138 (3.9%)	27 (0.8%)	23 (0.6%)	167 (4.7%)	1,766 (49.6%)	1,421 (39.9%)	210 (5.9%)	3,564	

\* Eclipse-collections

### **c1. Too low workload:**

In total, 74.4% of mutants are killed by the benchmarks with the largest workload (column **c1.g4**), which is significantly higher than the other three groups. In all cases of column **c1**, *ju2jmh*, JUnit, and AutoJMH strongly support our claim that benchmarks with the largest workload are better in detecting bugs, while JMH contradicts our claim.

In *Rxjava*, 1,088 (76.2%) *ju2jmh* benchmarks, 319 (68.3%) JUnit tests, and 32 (80.0%) AutoJMH benchmarks belong to the largest workload group (**g4**), while JMH benchmarks are normally distributed among the four groups and the largest workload’s group contains only 16.3% of the benchmarks. Furthermore, in *Eclipse-collections* and *Zipkin*, similar findings are obtained in comparing *ju2jmh*, JUnit, and AutoJMH. However, JMH completely stands against our claim in these two subjects where 23 (46%) *Eclipse-collections* benchmarks and 6 (85.7%) *Zipkin* benchmarks belong to the smallest workload’s groups (**g1**).

In conclusion, *ju2jmh*, JUnit, and AutoJMH all confirm that benchmarks with the largest workload are better in detecting performance bugs, but JMH does not necessarily support this claim.

### **c2. Unstable microbenchmarks:**

Column **c2** of table 6.5 depicts a highly significant effect of stability, with the most stable benchmarks (column **c2.g1**) killing 94.6% of mutants. All four test frameworks confirm that most mutants are killed by the most stable benchmarks. Specifically, across the three subjects, 71.4% to 96.1% of JMH benchmarks, 90.7% to 96.8% of *ju2jmh* benchmarks, 93.9% to 97.0% of JUnit tests, and 61.1% to 96.9% of AutoJMH benchmarks that killed a mutant belong to the most stable benchmarks.

In conclusion, microbenchmark stability has a significant impact on detecting performance bugs. The more stable benchmarks can better detect performance bugs.

### **c3. Limited mutant coverage:**

In general, the results do not necessarily confirm that benchmarks need higher `hitting_ratio` (e.g., column **c3.g4**) to detect performance bugs. However, if the benchmark hit the buggy statement with a lower ratio (e.g., column **c3.g1**), the mutant could not be killed by the benchmark. In total, two middle groups (**c3.g2** and **c3.g3**) killed 89.4% of all mutants, and only 4.7% of mutants are killed in a situation with the lowest `hitting_ratio` (**c3.g1**). As a result, we can conclude that in order to detect performance bugs, microbenchmarks require appropriate `hitting_ratio` values (not too high or too low). In other words, mutants with too low `hitting_ratio` cannot sufficiently degrade microbenchmarks. Moreover, microbenchmarks rarely hit a mutant with a large enough `hitting_ratio` to make significant

degradations. The data trend indicating that the number of microbenchmarks detecting bugs decreases in an interval with a high `hitting_ratio` reinforces the view stated in section 4.3.4, implying that some external factors, such as hardware limitations, may have an impact on microbenchmarks results.

According to Table 6.5, we can conclude that the positive effect of each of the three discussed causes can result in better detection of the performance bug. We conclude that benchmarks with sufficient workload, higher stability, and appropriate coverage can better kill the mutant. The three approaches of *ju2jmh*, JUnit, and AutoJMH confirm our claim that benchmarks with the largest workload can better detect bugs, while JMH does not necessarily confirm it. In all cases, all four approaches confirm that the most stable benchmarks can better detect bugs. Lastly, results do not confirm that benchmarks need high coverage, but they need an appropriate coverage of the bug to be able to detect the bug.

Our manual study identified three reasons why *ju2jmh* benchmarks outperform others. In particular, the experiment yields five top-five cases (out of a total of 25) in which the *ju2jmh* benchmark outperforms other tests the most based on *bug\_size*. The maximum *bug\_size* difference between *ju2jmh* benchmarks and other tests ranges from 0.5% to 97.2%. Among the 25 cases, AutoJMH benchmarks are involved in the majority of cases (16 of 25), while JMH benchmarks are involved in the fewest (4 of 25 cases). In addition, we manually analyze another 27 cases where mutants that can only be killed by *ju2jmh*. Based on the total 52 (i.e., 25 + 27) cases, we found the reasons why *ju2jmh* benchmarks outperform other tests, which can be summarized as follows: (1) In 28 cases, *ju2jmh* benchmarks hit the mutants more often than the other frameworks (more than 4× to 400,000×); (2) in 22 cases, *ju2jmh* benchmarks have higher stability (lower RSD) than the other frameworks; (3) in 13 cases, *ju2jmh* benchmarks have a more proper workload size (not too high or too low) than the other frameworks. The detailed experiment results are attached to the table in the Appendix section.

#### 6.4.4 Conclusion

In summary, to answer RQ4, we used prior work to derive three causes and conducted three experiments to measure these three causes. The results indicate that: (1) Too low workload can prevent *ju2jmh* benchmarks from detecting performance bugs effectively. The same conclusion applies to JUnit tests and AutoJMH benchmarks, but not to JMH benchmarks. (2) Unstable microbenchmarks can also impede microbenchmarks from detecting bugs, and this finding is not limited to a particular testing framework. (3) Performance bugs are more likely to be detected in microbenchmarks with appropriate execution frequency

(i.e., *hitting\_ratio*) for bugs during the microbenchmarking period. Too low or too high execution frequency can cause microbenchmarks to fail to detect performance bugs.

## 6.5 RQ5: How effective is the clustering strategy for performance microbenchmarking?

### 6.5.1 Motivation

Performance testing is indispensable for identifying bottlenecks and optimizing software systems. However, it remains a resource-intensive process, especially in large-scale projects where test suites often consist of numerous microbenchmarks. While tools like *ju2jmh* simplify the generation of microbenchmarks, they do not address the inefficiencies inherent in executing these tests individually. The repetitive overhead of initialization, execution, and result analysis in isolated microbenchmarks can consume significant resources, rendering performance testing impractical in many cases.

To address these challenges, our research introduces a clustering-based approach to optimize *ju2jmh* microbenchmarking. By leveraging code coverage information to group functionally similar microbenchmarks into clusters, we aim to enhance the utility of performance testing in the following ways:

- **Broader Analysis:** Clustering enables microbenchmarks to collectively identify broader performance trends, compensating for the limited scope of isolated tests.
- **Efficiency Gains:** Batch execution of clusters reduces redundant overhead, optimizing the resource usage in performance testing.
- **Improved Scalability:** This approach ensures that performance testing remains practical, even for large-scale microbenchmarking suites.

Through this methodology, we aim to streamline the performance testing workflow, ensuring that the enhancements in execution efficiency preserve the accuracy of results. Ultimately, this enables faster feedback loops and reduced computational burdens, empowering developers to conduct more accessible and effective performance testing. RQ5 investigates the feasibility and efficacy of this clustering strategy, providing critical insights into its scalability and reliability.

## 6.5.2 Approach

The following two metrics are used to evaluate the efficiency and stability of the batch execution strategy.

### Microbenchmarking Execution Time:

This metric is used to measure the efficiency of microbenchmarking for batch execution. By using individually executed microbenchmarks as a baseline for comparison, we can examine how batch execution speeds up microbenchmarking.

**Microbenchmarking Stability:** To determine the reliability and consistency of the microbenchmarks in detecting performance-related issues, we assess the stability in microbenchmarking results across multiple executions of both individual and batch-executed microbenchmarks. Stability serves a critical metric that helps developers observe whether a program behaves consistently under performance testing and is useful for real-world performance testing enhancement.

We analyze the stability of microbenchmarks using the *Relative Standard Deviation (RSD)* of their execution time, where a lower RSD value indicates higher stability and vice versa. For each benchmark and cluster, we calculate RSD across 30 iterations and measure its stability. In particular,

- A  $\leq 1\%$  RSD indicates a stable result where clustering retains accuracy.
- A  $> 5\%$  RSD suggests an unstable result where clustering may compromise accuracy.
- For RSD between 1% and 5%, we evaluated whether the cluster's RSD was lower than most individual benchmarks in that cluster to assess sufficiency.

## 6.5.3 Results

We select 14,301 out of 35,084 *ju2jmh* microbenchmarks with execution time below  $2\mu s$ , as these microbenchmarks tend to have limited utility, due to their high variance and limited scope [35] that can benefit from batch execution. The 14,301 *ju2jmh* microbenchmarks are grouped into 2,124 clusters. Each cluster is represented as a JMH microbenchmark that sequentially invokes the individual microbenchmarks within its payload. Our evaluations compares the efficiency and stability between individually and batch-executed microbenchmarks.

### Execution Time Savings

Table 6.6 highlights the efficiency of the batch execution strategy across three Java projects: *RxJava*, *Eclipse-collections*, and *ZipKin*. The results demonstrate significant reductions in

execution time, with savings ranging from 81.2% to 86.2%.

- **RxJava:** Reduced from 51.3 hours to 9.5 hours (81.5% saved).
- **Eclipse-collections:** Reduced from 185.15 hours to 25.55 hours (86.2% saved).
- **ZipKin:** Reduced from 1.86 hours to 0.35 hours (81.2% saved).

**Scalability:** The clustering approach effectively groups benchmarks, with cluster sizes averaging 5 to 7 benchmarks. Larger projects like Eclipse-collections see the highest percentage of time saved (86.2%) due to greater opportunities for redundancy reduction.

**Impact:**

The observations from Table 6.6 indicate the significant efficiency improvement for microbenchmarks brought by batch execution. Since batch execution does not involve code changes, this strategy will not affect code coverage, which minimizes the risks of reducing test reliability. The batch-executed strategy scales well across projects of different sizes, ensuring faster feedback loops and reduced costs in performance testing workflows.

Table 6.6: Time saved using batch-executed strategy

	RxJava	Eclipse-collections	ZipKin
# of <i>ju2jmh</i> benchmarks	3,080	11,109	112
# of clusters	570	1,533	21
Avg. size of clusters	5.13	6.91	5.42
Total time for individuals (hours)	51.3	185.15	1.86
Total time for clusters (hours)	9.50	25.55	0.35
% of time saved (%)	81.50	86.20	81.20

**Benchmarks’ Stability**

Table 6.7 evaluates the stability of clustered and individual benchmarks by comparing the Relative Standard Deviation (RSD) of their execution time and the percentages of stable ( $\leq 1\%$  RSD) and unstable ( $\geq 5\%$  RSD) benchmarks across the studied subjects.

**RSD Comparison:** The average RSD for clusters is slightly higher than for individual benchmarks (e.g., 0.78% vs. 0.43% for RxJava), with a maximum increment of 0.48%, reflecting the slightly reduced stability from batching microbenchmarks. Despite this, the extra variability remains within acceptable bounds while the increased variability remains negligible, demonstrating the stability of batch execution for microbenchmarking.

**Stability:** The percentage of stable benchmarks is consistently high for clusters (78.3%-87.5%), although slightly lower than individual benchmarks (83.5%-97.3%). Unstable benchmarks remain rare for both clusters ( $\leq 0.9\%$ ) and individuals ( $\leq 0.6\%$ ).

**Impact:** These results show that batch execution retains sufficient stability. This ensures that clustering does not compromise the ability to detect performance regressions effectively.

Table 6.7: RSD of clusters and individuals, the percentage of Stable ( $\leq 1\%$ ) and Unstable ( $\geq 5\%$ ) microbenchmarks

	RxJava	Eclipse-collections	ZipKin
Average RSD of individuals (%)	0.43	0.39	0.56
Average RSD of clusters (%)	0.78	0.87	0.85
RSD difference	0.35	0.48	0.29
Stable individuals (%)	95.8	97.3	83.5
Stable clusters (%)	78.3	81.2	87.5
Stability difference (%)	-17.5	-16.1	4
Unstable individuals (%)	0.30	0.60	0.40
Unstable clusters (%)	0.90	0.60	0.35
Instability difference (%)	0.60	0.00	-0.05

## 6.5.4 Conclusion

To conclude, the results demonstrate that the proposed batch execution strategy significantly improves the efficiency of performance microbenchmarking. Table 6.6 highlights substantial time savings, with batch execution reducing execution times by over 80% across all three Java projects, making performance testing more scalable and cost-effective. Table 6.7 confirms that batch execution maintains sufficient stability, with the variability (measured as RDS) remaining negligible. While batch-executed microbenchmarks exhibit slightly higher RSD compared to individual benchmarks, the majority of microbenchmarks remain stable, and unstable benchmarks are rare. Therefore, these results indicate that batch execution achieves significant reductions in execution time without compromising the stability needed for effective performance testing, offering a practical solution to streamline performance test suites in diverse software projects.

# Chapter 7

## Conclusion

### 7.1 Summary of Results

This thesis explored five key research questions to enhance the effectiveness and efficiency of performance microbenchmarking in Java applications:

- **RQ1: Stability of Automatically Generated Benchmarks**

The study demonstrated that *ju2jmh*-generated benchmarks exhibit sufficient stability compared to manually crafted benchmarks. Relative Standard Deviation (RSD) measurements showed that while the automatically generated benchmarks had slightly higher variability, they remained within acceptable bounds, proving their reliability for performance testing.

- **RQ2: Detection of Artificial Performance Bugs**

Using the Performance Mutation Testing (PMT) framework, *ju2jmh* benchmarks achieved competitive mutation scores, indicating their capability to detect artificial performance bugs effectively. This validated the robustness and sensitivity of the generated benchmarks in identifying performance degradations introduced by controlled mutations.

- **RQ3: Detection of Real-World Performance Bugs**

*ju2jmh* benchmarks successfully detected several real-world performance bugs across the studied projects. Their performance was comparable to manually crafted benchmarks, affirming the practical utility of automated benchmark generation for identifying significant performance issues in real-world scenarios.

- **RQ4: Factors Affecting Benchmark Performance**

The study identified key factors influencing the effectiveness of performance benchmarks, including workload characteristics, environmental noise, and the complexity of the tested code. These insights provide actionable guidance for improving benchmark design and execution strategies.

- **RQ5: Effectiveness of the Clustering Strategy**

The proposed clustering strategy for batch-executing microbenchmarks significantly improved performance testing efficiency. Across three large-scale projects, execution time was reduced by over 80% without compromising reliability. This approach demonstrated scalability and practicality, making it a valuable addition to performance benchmarking workflows.

## 7.2 Conclusion

This research advances the state-of-the-art in performance benchmarking by addressing key challenges of automation, efficiency, and reliability. The *ju2jmh* framework bridges the gap between functional testing and performance microbenchmarking, empowering developers to integrate robust performance testing seamlessly into modern development workflows. Additionally, the clustering strategy enhances scalability, enabling the efficient execution of large-scale benchmark suites. These contributions provide a foundation for further innovations in software performance testing and underline the importance of automated and optimized methodologies in contemporary software engineering.

## 7.3 Future Work

While this study makes significant contributions, several opportunities for future research remain:

- **Extending Evaluations to Diverse Projects:** Expanding the scope to include a broader range of software projects, especially from different domains and programming paradigms, could validate the generalizability of the findings.
- **Improving Clustering Granularity:** Refining the clustering strategy to better balance efficiency and accuracy could further optimize execution time while maintaining high reliability.
- **Evaluating the Clustering Strategy with PMT:** Future work can leverage the Performance Mutation Testing (PMT) framework to assess the quality and robustness

of the clustering approach. This would provide deeper insights into its impact on detecting performance bugs and its overall effectiveness.

- **Integration with Continuous Integration Pipelines:** Exploring the integration of *ju2jmh* and the clustering strategy into modern Continuous Integration (CI) environments could streamline performance testing workflows for developers.
- **Support for Non-Java Environments:** Extending *ju2jmh* to support other programming languages and frameworks would increase its applicability across diverse software ecosystems.

## 7.4 Threats to validity

### 7.4.1 External Validity

We performed our evaluations on three popular open-source Java software systems from a limited system set that extensively deploys JMH along with JUnit. However, the subjects' JMH benchmarks are actively maintained by professional development teams and include sufficient microbenchmarks, thus they are suitable to be used in our experiments.

We developed five mutation operators in our *PMT* framework, but there would be many other known/unknown performance bug patterns in real-world systems. We randomly selected our study mutants from a broad set of generated mutants. Various mutants can have a wide range of characteristics that our chosen mutants may not cover. In the future, we will explore more mutants and extend our framework to support more mutant types.

During our study, we found that some JUnit tests are performance unit tests rather than functional unit tests. At the time of writing, our tool does not distinguish them since it is tricky to do so through static analysis of source code. Future work may consider addressing this issue to improve our existing approach.

In this paper, we leverage the code coverage as an indicator to cluster functionally similar microbenchmarks. While other methods such as static code analysis, dynamic call graphs, or semantic code embeddings could be used for similarity measurement, such approaches usually require complex code analysis frameworks or face challenges of dynamic language features. Therefore, we choose this lightweight yet effective approach to capture code with similar execution paths without deep analysis overhead, while maintaining high correlation with performance characteristics.

### 7.4.2 Internal Validity

Monitoring performance is always challenging due to noises [40]. To minimize such errors as much as possible, we: (1) executed benchmarks for 30 iterations each per measurement; (2) used isolated, controlled, and large-scale cloud computing resources provided by *Google Cloud*; and (3) reduced random bias by employing an evaluation strategy, i.e., bootstrapping.

# References

- [1] Jacoco: Java code coverage library. <https://www.jacoco.org>. Accessed: 2024-11-25.
- [2] Hammam M AlGhamdi. Automated approaches for reducing the execution time of performance tests. Master's thesis, Queen's University (Canada), 2017.
- [3] Hammam M. AlGhamdi, Cor-Paul Bezemer, Weiyi Shang, Ahmed E. Hassan, and Parminder Flora. Towards reducing the time needed for load testing. *Journal of Software: Evolution and Process*, page e2276, 2015. e2276 smr.2276.
- [4] Hammam M. Alghmadi, Mark D. Syer, Weiyi Shang, and Ahmed E. Hassan. An automated approach for recommending when to stop performance tests. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 279–289, 2016.
- [5] Alberto Avritzer and Elaine J Weyuker. The role of modeling in the performance testing of e-commerce applications. *IEEE Transactions on Software Engineering*, 26(12):1145–1156, 1998.
- [6] V. Bergmann. Contiperf, 2012. Accessed: 2021-09-24.
- [7] Cor-Paul Bezemer, Simon Eismann, Vincenzo Ferme, Johannes Grohmann, Robert Heinrich, Pooyan Jamshidi, Weiyi Shang, André van Hoorn, Monica Villavicencio, Jürgen Walter, and Felix Willnecker. How is performance addressed in DevOps? In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering, ICPE '19*, page 45–50, New York, NY, USA, 2019. Association for Computing Machinery.
- [8] Stephen M. Blackburn, Kathryn S. McKinley, Robin Garner, Chris Hoffmann, Asjad M. Khan, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovik, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. Wake up and smell the coffee: Evaluation methodology for the 21st century. *Commun. ACM*, 51(8):83–89, August 2008.

- [9] André B Bondi. Characteristics of scalability and their impact on performance. In *Proceedings of the 2nd international workshop on Software and performance*, pages 195–203. ACM, 2000.
- [10] L. Bulej, T. Bures, Vojtech Horký, Jaroslav Kotrc, L. Marek, Tomáš Trojánek, and P. Tuma. Unit testing performance with stochastic performance logic. *Automated Software Engineering*, 24:139–187, 2015.
- [11] Lubomír Bulej, Vojtech Horký, and Petr Tůma. Do we teach useful statistics for performance evaluation? In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, ICPE '17 Companion, page 185–189, New York, NY, USA, 2017. Association for Computing Machinery.
- [12] Joe Challenger, Paul Dantzig, Arun Iyengar, and Mark Squillante. Efficiently serving dynamic data at highly accessed web sites. *IEEE/ACM Transactions on Networking*, 8(2):233–243, 2000.
- [13] Dev K. Chhetri, Brian A. Malloy, James F. Power, and Lori Pollock. Performance mutation testing with pmt: Design, implementation, and evaluation. In *Proceedings of the 32nd International Conference on Software Engineering*, pages 176–185, 2010.
- [14] Inc. Clarkware Consulting. Junitperf, 2009. Accessed: 2021-07-02.
- [15] Cliff Click. The art of java benchmarking, 2010. Accessed: 2021-07-02.
- [16] Oracle Corporation. Java microbenchmarking harness (jmh), 2016. Accessed: 2021-07-02.
- [17] Charlie Curtsinger and Emery D. Berger. Stabilizer: Statistically sound performance evaluation. *SIGPLAN Not.*, 48(4):219–228, March 2013.
- [18] Diego Elias Damasceno Costa, Cor-Paul Bezemer, Philip Leitner, and Artur Andrzejak. What’s wrong with my benchmark results? studying bad practices in jmh benchmarks. *IEEE Transactions on Software Engineering*, pages 1–1, 2019.
- [19] A. C. Davison and D. V. Hinkley. *Bootstrap Methods and their Application*. Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge University Press, 1997.
- [20] Daniel Delaet, Hans Vandierendonck, and Koen De Bosschere. A quantitative study of jvm execution characteristics using java grande benchmarks. *Concurrency and Computation: Practice and Experience*, 16(7):555–577, 2004.
- [21] Pedro Delgado-Pérez, Ana Belén Sánchez, Sergio Segura, and Inmaculada Medina-Bulo. Performance mutation testing. *Software Testing, Verification and Reliability*, 31(5):e1728, 2021. e1728 stvr.1728.

- [22] Zishuo Ding, Jinfu Chen, and Weiyi Shang. Towards the use of the readily available tests from the release pipeline as performance tests: Are we there yet? In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, ICSE '20*, page 1435–1446, New York, NY, USA, 2020. Association for Computing Machinery.
- [23] Paul M. Duvall, Steve Matyas, and Andrew Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley Professional, 2007.
- [24] Emad Fallahzadeh, Amir Hossein Bavand, and Peter C Rigby. Accelerating continuous integration with parallel batch testing. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 55–67, 2023.
- [25] Apache Software Foundation. Apache jmeter, 2021. Accessed: 2021-09-24.
- [26] Erich Gamma and Kent Beck. *JUnit Pocket Guide*. O'Reilly Media, Inc., 2004.
- [27] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous java performance evaluation. *SIGPLAN Not.*, 42(10):57–76, October 2007.
- [28] Andy Georges, Lieven Eeckhout, and Dries Buytaert. Java performance evaluation through rigorous replay compilation. In *In ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 367–384, 2008.
- [29] Google Corporation. Caliper, 2015. Accessed: 2021-07-02.
- [30] Tim C. Hesterberg. What teachers should know about the bootstrap: Resampling in the undergraduate statistics curriculum. *The American Statistician*, 69(4):371–386, 2015. PMID: 27019512.
- [31] Peng Huang, Xiao Ma, Dongcai Shen, and Yuanyuan Zhou. Performance regression testing target prioritization via performance risk analysis. In *Proceedings of the 36th International Conference on Software Engineering, ICSE 2014*, page 60–71, New York, NY, USA, 2014. Association for Computing Machinery.
- [32] Karl Huppler. The art of building a good benchmark. In *Proceedings of the First TPC Technology Conference on Performance Evaluation & Benchmarking*, pages 18–30. Springer, 2009.
- [33] Raj Jain. *The art of computer systems performance analysis: Techniques for experimental design, measurement, simulation, and modeling*. Wiley, 1991.
- [34] Raj Jain and Imrich Chlamtac. The p performance model for computer systems. *IEEE Transactions on Computers*, C-34(4):329–336, 1985.

- [35] Mostafa Jangali, Yiming Tang, Niclas Alexandersson, Philipp Leitner, Jinqiu Yang, and Weiyi Shang. Automated generation and evaluation of jmh microbenchmark suites from unit tests. *IEEE Transactions on Software Engineering*, 49(4):1704–1725, 2022.
- [36] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011.
- [37] Zhen Ming Jiang and Ahmed E. Hassan. A survey on load testing of large-scale software systems. *IEEE Transactions on Software Engineering*, 41(11):1091–1118, 2015.
- [38] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. *SIGPLAN Not.*, 47(6):77–88, June 2012.
- [39] Tomas Kalibera and Richard Jones. Rigorous benchmarking in reasonable time. In *Proceedings of the 2013 international symposium on memory management*, pages 63–74, 2013.
- [40] Tomas Kalibera and Richard Jones. Quantifying performance changes with effect size confidence intervals, 2020. Accessed: 2021-07-02.
- [41] Christoph Laaber. Continuous software performance assessment: Detecting performance problems of software libraries on every build. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019*, page 410–414, New York, NY, USA, 2019. Association for Computing Machinery.
- [42] Christoph Laaber. pa - performance (change) analysis using bootstrap, 2021. Accessed: 2021-07-02.
- [43] Christoph Laaber, Harald C Gall, and Philipp Leitner. Applying test case prioritization to software microbenchmarks. *Empirical Software Engineering*, 26(6):133, 2021.
- [44] Christoph Laaber and Philipp Leitner. An evaluation of open-source software microbenchmark suites for continuous performance assessment. In *Proceedings of the 15th International Conference on Mining Software Repositories, MSR '18*, page 119–130, New York, NY, USA, 2018. Association for Computing Machinery.
- [45] Christoph Laaber and Philipp Leitner. Performance change diagnosis in continuous integration systems. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 545–549, 2018.
- [46] Christoph Laaber, Joel Scheuner, and Philipp Leitner. Software microbenchmarking in the cloud. how bad is it really? *Empirical Softw. Engg.*, 24(4):2469–2508, August 2019.

- [47] Christoph Laaber, Stefan Würsten, Harald C. Gall, and Philipp Leitner. Dynamically reconfiguring software microbenchmarks: Reducing execution time without sacrificing result quality. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 989–1001, New York, NY, USA, 2020. Association for Computing Machinery.
- [48] Philipp Leitner and Cor-Paul Bezemer. An exploratory study of the state of practice of performance testing in java-based open source projects. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ICPE '17, page 373–384, New York, NY, USA, 2017. Association for Computing Machinery.
- [49] Philipp Leitner and Jürgen Cito. Patterns in the chaos—a study of performance variation and predictability in public iaas clouds. *ACM Trans. Internet Technol.*, 16(3), April 2016.
- [50] Mario Linares-Vásquez, Gabriele Bavota, Michele Tufano, Kevin Moran, Massimiliano Di Penta, Christopher Vendome, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. Enabling mutation testing for android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 233–244, New York, NY, USA, 2017. Association for Computing Machinery.
- [51] Yepang Liu, Chang Xu, and Shing-Chi Cheung. Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 1013–1024, New York, NY, USA, 2014. Association for Computing Machinery.
- [52] Paulo Martins, Julio Queiroz, and Gustavo Rodrigues. Understanding limitations of using junit for benchmarking. In *Proceedings of the International Conference on Evaluation and Assessment in Software Engineering*. EASE, 2016.
- [53] Daniel A Menasce, Virgílio A Almeida, and Lawrence W Dowdy. *Performance by design: Computer capacity planning by example*. Prentice Hall, 2004.
- [54] Ian Molyneaux. *The Art of Application Performance Testing: Help for Programmers and Quality Assurance*. O'Reilly Media, Inc., 1st edition, 2009.
- [55] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! *SIGPLAN Not.*, 44(3):265–276, March 2009.
- [56] Adrian Nistor, Po-Chun Chang, Cosmin Radoi, and Shan Lu. Caramel: Detecting and fixing performance problems that have non-intrusive fixes. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, pages 902–912, 2015.

- [57] Adrian Nistor, Tian Jiang, and Lin Tan. Discovering, reporting, and fixing performance bugs. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 237–246, 2013.
- [58] Adrian Nistor, Tien-Duy B. Nguyen, Hoan A. Nguyen, and Tien N. Nguyen. Boa: A framework for analyzing the behavior of object-oriented applications. *IEEE Transactions on Software Engineering*, 2017.
- [59] Oswaldo Olivo, Isil Dillig, and Calvin Lin. Static detection of asymptotic performance bugs in collection traversals. *SIGPLAN Not.*, 50(6):369–378, June 2015.
- [60] Oracle Corporation. Japex micro-benchmark framework, 2014. Accessed: 2021-07-02.
- [61] Hristina Palikareva, Magnus O. Myreen, and Gavin Lowe. Mechanising and verifying java jit optimisations in isabelle/hol. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 476–495. Springer, 2016.
- [62] Kai Pan, Sunghun Kim, and E. James Whitehead. Toward an understanding of bug fix patterns. *Empirical Softw. Engg.*, 14(3):286–315, June 2009.
- [63] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. Chapter six - mutation testing advances: An analysis and survey. volume 112 of *Advances in Computers*, pages 275–378. Elsevier, 2019.
- [64] Mike Papadakis and Nikos Malevris. An empirical evaluation of the first and second order mutation testing strategies. In *Proceedings of the 8th International Workshop on Mutation Analysis (Mutation)*, pages 1–10. IEEE, 2013.
- [65] Shravan Pargaonkar. A comprehensive review of performance testing methodologies and best practices: software quality engineering. *International Journal of Science and Research (IJSR)*, 12(8):2008–2014, 2023.
- [66] Goran Petrovic, Marko Ivankovic, Gordon Fraser, and René Just. Does mutation testing improve testing practices? *CoRR*, abs/2103.07189, 2021.
- [67] Aida Radu and Sarah Nadi. A dataset of non-functional bugs. In *Proceedings of the 16th International Conference on Mining Software Repositories, MSR '19*, page 399–403. IEEE Press, 2019.
- [68] Shiquan Ren, Hong Lai, Wenjing Tong, Mostafa Aminzadeh, Xuezhong Hou, and Shenghan Lai. Nonparametric bootstrapping for hierarchical data. *Journal of Applied Statistics*, 37(9):1487–1498, 2010.
- [69] Marcelino Rodriguez-Cancio, Benoit Combemale, and Benoit Baudry. Automatic Microbenchmark Generation to Prevent Dead Code Elimination and Constant Folding.

- In *31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*, ASE 2016:Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, Singapore, Singapore, September 2016.
- [70] Trever Schirmer, Tobias Pfandzelter, and David Bermbach. Elastibench: Scalable continuous benchmarking on cloud faas platforms. *arXiv preprint arXiv:2405.13528*, 2024.
  - [71] Ohad Shacham, Martin Vechev, and Eran Yahav. Chameleon: Adaptive selection of collections. *SIGPLAN Not.*, 44(6):408–418, June 2009.
  - [72] Aleksey Shipilev. Java benchmarking as easy as two timestamps., 2021. Accessed: 2021-07-02.
  - [73] Aleksey Shipilev. Nanotrusting the nanotime., 2021. Accessed: 2021-07-02.
  - [74] Petr Stefan, Vojtech Horky, Lubomir Bulej, and Petr Tuma. Unit testing performance in java projects: Are we there yet? In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering, ICPE '17*, page 401–412, New York, NY, USA, 2017. Association for Computing Machinery.
  - [75] E.J. Weyuker and F.I. Vokolos. Experience with performance testing of software systems: issues, an approach, and case study. *IEEE Transactions on Software Engineering*, 26(12):1147–1156, 2000.
  - [76] Murray Woodside, Greg Franks, and Dorina C Petriu. The future of software performance engineering. In *Proceedings of the 2007 Future of Software Engineering*, pages 171–187. IEEE, 2007.
  - [77] Shahed Zaman, Bram Adams, and Ahmed E. Hassan. A qualitative study on performance bugs. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories, MSR '12*, page 199–208. IEEE Press, 2012.