



# An Empirical Study on Code Coverage of Performance Testing

Muhammad Imran  
University of L'Aquila  
L'Aquila, Italy

[mohammad.imran@graduate.univaq.it](mailto:mohammad.imran@graduate.univaq.it)

Vittorio Cortellessa  
University of L'Aquila  
L'Aquila, Italy

[vittorio.cortellessa@univaq.it](mailto:vittorio.cortellessa@univaq.it)

Davide Di Ruscio  
University of L'Aquila  
L'Aquila, Italy

[davide.diruscio@univaq.it](mailto:davide.diruscio@univaq.it)

Riccardo Rubei  
University of L'Aquila  
L'Aquila, Italy  
[riccardo.rubei@univaq.it](mailto:riccardo.rubei@univaq.it)

Luca Traini  
University of L'Aquila  
L'Aquila, Italy  
[luca.traini@univaq.it](mailto:luca.traini@univaq.it)

## ABSTRACT

Performance testing aims to ensure the operational efficiency of software systems. However, many factors influencing the efficacy and adoption of performance tests in practice are not yet fully understood. For instance, while code coverage is widely regarded as a key quality metric for evaluating the efficacy of functional testing suites, there is limited knowledge about the types and levels of coverage that performance tests specifically achieve. Another important factor, often perceived as a barrier to the broader adoption of performance tests yet remaining relatively unexplored, is their extended execution time. In this paper, we analyze the performance testing suites of 28 open-source systems to study (i) the magnitude of their code coverage, and (ii) their execution time. Our analysis shows that performance tests achieve significantly lower code coverage than functional tests, as expected, and it highlights a significant trade-off between coverage and execution time. Our results also suggest, in perspective, that automated test generation methods might not ensure affordable performance testing due to the associated time cost. This finding poses new challenges in the field of performance test generation.

## CCS CONCEPTS

• Software and its engineering → Empirical software validation; Software testing and debugging; Software performance.

## KEYWORDS

Performance Testing, Code Coverage, JMH, Microbenchmarking

## ACM Reference Format:

Muhammad Imran, Vittorio Cortellessa, Davide Di Ruscio, Riccardo Rubei, and Luca Traini. 2024. An Empirical Study on Code Coverage of Performance Testing. In *28th International Conference on Evaluation and Assessment in Software Engineering (EASE 2024), June 18–21, 2024, Salerno, Italy*. ACM, New York, NY, USA, 10 pages. <https://doi.org/10.1145/3661167.3661196>



This work is licensed under a Creative Commons Attribution International 4.0 License.

EASE 2024, June 18–21, 2024, Salerno, Italy  
© 2024 Copyright held by the owner/author(s).  
ACM ISBN 979-8-4007-1701-7/24/06  
<https://doi.org/10.1145/3661167.3661196>

## 1 INTRODUCTION

Software performance is a critical non-functional aspect of software systems. Deterioration in performance can present significant business challenges, including user dissatisfaction [6] and financial losses [26]. To address these concerns, organizations typically employ performance testing [34], a technique designed to evaluate the software system's performance before its deployment in a production environment. However, the practical implementation of performance testing faces challenges.

One of such challenges is associated with the development and maintainability of performance testing suites. Creating these tests often requires specific technical expertise that may not be readily available to the average developer [5, 30]. Additionally, software development processes tend to prioritize functional development activities, which can lead to limited resource allocation for quality assurance tasks [2, 5], such as the creation and maintenance of performance tests [30]. These factors collectively contribute to the oversight of performance assurance activities, leading to potential implications on the *quality* of performance testing suites.

The traditional way of assessing the *quality* of a testing suite involves using *code coverage*. This metric gauges the extent to which the testing suite executes the software source code, serving as a simple yet effective indicator of the testing suite quality. Although the validity of code coverage as a measure of test quality is still a matter of debate [11, 14], it remains the de-facto standard for evaluating testing suites in practical scenarios. Code coverage has traditionally been used and studied in the context of functional testing [11, 14, 15, 27] (e.g., unit tests); however, there are increasing indications that its relevance extends to software performance testing as well. For instance, researchers have shown that code coverage significantly impacts the test capability of triggering performance bugs [10], and that extending coverage in performance testing suites can enhance their effectiveness in discovering performance issues [16].

Despite these indications, there is still limited knowledge about the code coverage achieved by performance tests. To address this gap, this paper presents the first empirical study to investigate the coverage of performance testing suites. Our methodology involved selecting 28 Java software systems on GitHub that featured JMH benchmarks, a widely used form of performance tests in Java software. We then conducted a comprehensive repository analysis to extract all Java methods within these software systems. Finally, we executed 2,190 JMH benchmarks on these Java software systems

to determine the methods covered by performance testing. Our findings revealed that JMH benchmarks achieve a limited code coverage of about 8.8% on average, and have coverage that is 4 times less than that of JUnit tests.

Another important factor typically at odds with code coverage is the time cost of test execution [35]. Indeed, testing suites that achieve higher code coverage generally require more time-consuming tests and/or a larger number of tests and, therefore, a higher execution time. This can be particularly problematic for performance tests, which are widely known for their time-consuming nature [17], as they often necessitate a certain degree of repetition to deal with the inherent variability of performance measurements [19, 24, 31]. Given the significance of the time cost in performance testing and its inherent relation to code coverage, we conducted an additional analysis to investigate this aspect. Our results indicate that JMH suites are significantly more time-consuming than JUnit suites, with an average execution time of 62 times higher.

The main contributions of this paper are:

- A first comprehensive empirical investigation on performance test coverage, in particular we rely on method-level coverage to evaluate the coverage of testing suites.
- A first evaluation of the execution time cost of performance testing suites across multiple software systems.
- An empirical comparison between performance and functional testing, regarding code coverage and execution time cost.
- A replication package [13] containing the dataset we mined in the study, the scripts we used to perform the data analysis, and the detailed results of our analysis.

## 2 STUDY DESIGN

This study investigates the code coverage achieved by *performance tests* and their time cost, given the relevance of this aspect for the practical usage of performance tests. To aid the interpretation of our results, we also conduct a comparative analysis with *functional tests* to assess the differences both in terms of code coverage and time cost.

We focus on JMH microbenchmarks and JUnit tests due to their extensive use within the Java ecosystem. JMH is the de-facto standard for developing and running microbenchmarks, a widely known form of performance tests in Java software [23] while JUnit stands as one of the most popular libraries for implementing and executing Java functional tests. In this study, we aim to address the following research questions (RQs):

► **RQ<sub>1</sub>:** *To what extent do performance tests cover the source code of software systems?* Our objective is to evaluate the extent of code coverage achieved by performance testing suites in various software systems. We will assess coverage from multiple perspectives, including overall and direct coverage. Additionally, we intend to examine the level of overlap among different tests in their coverage of identical sections of the source code.

► **RQ<sub>2</sub>:** *How does the code coverage achieved by performance testing compare to that of functional testing?* We aim to gain insights into the differences between performance test coverage and functional

test coverage. Our analysis will encompass *suite-level* and *test-level* coverage and the extent of coverage overlap within testing suites.

► **RQ<sub>3</sub>:** *What is the time cost of performance testing?* We assess the time cost incurred during the execution of performance tests. We will measure the overall time consumed at *suite-level* and *test-level*, thus providing insights on the temporal impact of performance testing at different granularity.

► **RQ<sub>4</sub>:** *How does the time cost of performance testing compare to that of functional testing?* We aim to explore the differences in execution time between performance testing suites and functional testing suites. This investigation aims at comparing the time resources required by these two types of testing within the software development context.

### 2.1 Main steps of the performed study

As shown in Fig.1, the executed process consists of three main steps: (i) *preliminary selection* of the software systems to analyze, (ii) *raw data collection*, and (iii) *data wrangling* phase, as described in the following subsections.

**Preliminary Selection** ①: We selected an initial pool of 40 Java software projects hosted on GitHub. This selection was guided by four key considerations: (i) these systems are well-established Java libraries that cover a broad spectrum of domains, (ii) each of these projects includes JMH benchmarks and JUnit tests, (iii) we are familiar with the commands necessary to execute the JMH microbenchmarks for these systems, and (iv) they have been used in prior work [20–22, 31, 32], thus supporting their appropriateness in this context.

**Raw Data Collection** ②: Our data collection combines both static code analysis and dynamic analysis of test executions. In particular, we collected three distinct types of raw data from each Java system, i.e., *srcML XML files*, *JMH/JUnit callstacks*, and *JMH/JUnit reports*.

► *srcML XML files*: We transformed each Java source file into a structured XML format using srcML toolkit [9]. This transformation facilitates a structured and reliable static analysis of the source code, enabling a straightforward extraction of relevant code information, such as the list of Java method signatures that appear in the project. In this process, we encountered a specific issue with two projects, namely *apache hive* and *eclipse jersey*, where the srcML toolkit generated incomplete XML files that omitted the representation of specific Java source files. Thus, we decided to exclude these two projects from our analysis. As a result, we successfully parsed the source code of 38 of our initially selected projects.

► *JMH/JUnit callstacks*: We rely on dynamic analysis to identify the code components covered (or not covered) by JMH/JUnit tests. We prefer dynamic analysis over static analysis due to several limitations of the latter [33], such as its inability to reliably derive method invocations. Specifically, we employed *async-profiler*<sup>1</sup> to profile the execution of JMH benchmarks and JUnit tests, capturing their respective call stacks. The profiler is configured to record call stacks with a 1-nanosecond sampling interval. A call stack reports the currently active methods in the CPU and the sequence of their invocations. For illustration, consider Fig. 2, which presents a call stack from the execution of a JMH benchmark named `skinnyEncodeIntoCompressedByteBuffer`. Through this call

<sup>1</sup><https://github.com/async-profiler/async-profiler>

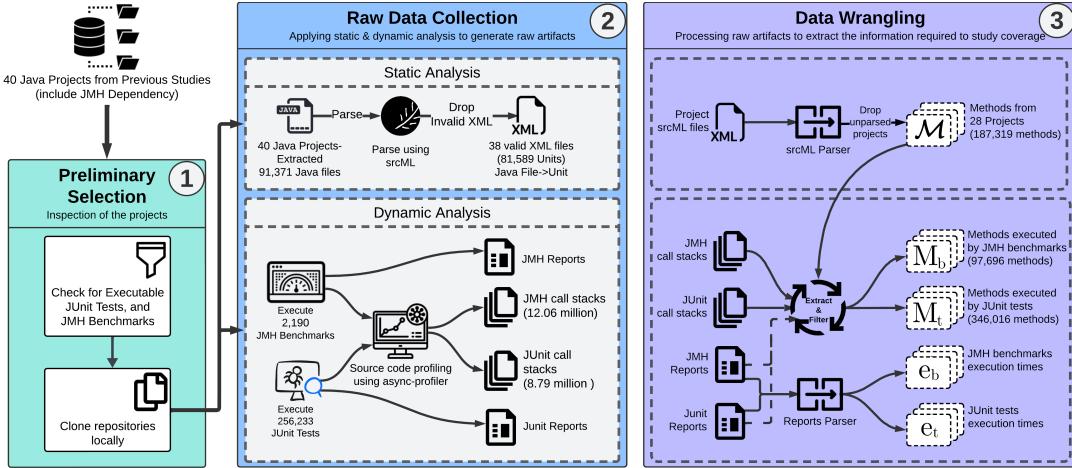


Figure 1: Main steps of the performed study.

stack, we can deduce the sequence of executed methods within the benchmark. From Fig. 2, we can observe that the benchmark first calls `encodeIntoCompressedByteBuffer`, which subsequently invokes `encodeIntoByteBuffer`, and this is followed by `writeCountsDiffs`, and so forth.

After executing each testing suite, we produced a distinct file that catalogs all unique call stacks observed during the execution. However, during this procedure, we faced issues attaching the profiler to the JMH benchmarks of 7 projects. Consequently, we excluded these projects from our analysis. We faced similar issues for JUnit testing in 9 projects. As a result, we collected 12.06 million unique call stacks for JMH benchmarks from 31 systems, and 8.79 million unique call stacks for JUnit tests from 19 projects.

► *JMH/JUnit reports:* We collected JMH and JUnit reports to obtain two primary information: (i) the list of JMH and JUnit tests

```

public class HdrHistogramEncodingBench {
    @Benchmark
    public void skinnyEncodeIntoCompressedByteBuffer() {
        buffer.clear();
        skinnyHistogram.encodeIntoCompressedByteBuffer(buffer)
    }
}

1 java.lang.Thread.run;
2 java.util.concurrent.ThreadPoolExecutor.Worker.run;
3 java.util.concurrent.ThreadPoolExecutor.runWorker;
4 java.util.concurrent.FutureTask.run;
5 java.util.concurrent.Executors$RunnableAdapter.call;
6 java.util.concurrent.FutureTask.run;
7 java.lang.reflect.Method.invoke;
8 sun.reflect.DelegatingMethodAccessorImpl.invoke;
9 sun.reflect.NativeMethodAccessorImpl.invoke;
10 sun.reflect.NativeMethodAccessorImpl.invoke0;
11 bench.HdrHistogramEncodingBench.skinnyEncodeIntoCompressedByteBuffer();
12 org.HdrHistogram.SkinnyHistogram.encodeIntoCompressedByteBuffer();
13 org.HdrHistogram.SkinnyHistogram.encodeIntoByteBuffer();
14 org.HdrHistogram.SkinnyHistogram.writeCountsDiffs();
15 org.HdrHistogram.SkinnyHistogram.putInt();
16 java.nioHeapByteBuffer.putInt();
17 java.nio.Buffer.nextPutIndex();

public class SkinnyHistogram extends Histogram {
    synchronized public int
    encodeIntoCompressedByteBuffer(final ByteBuffer targetBuffer) {
        int uncompressLength =
            this.encodeIntoByteBuffer(intermediateUncompressedByteBuffer);
        benchmarked project method
    }
}

```

(a) Example invocation of a method from a benchmark. (b) Benchmarked method definition and indirect call to another project method. (c) Example of a collected call stack.

Figure 2: (a) Example invocation of a method from a benchmark. (b) Benchmarked method definition and indirect call to another project method. (c) Example of a collected call stack.

and (ii) their corresponding execution times. A naive approach to obtaining the execution time of JMH benchmarks would involve their actual execution. However, this methodology would be extremely time-consuming and, therefore, impractical for our study. Instead, we take advantage of the time-based nature of the JMH test to bypass the need for actual execution. In particular, JMH allows developers to configure the number of repetitions for each JMH test, which eventually defines its final execution time. To obtain the JMH configurations for each JMH test, we applied an approach similar to a prior work [31], exploiting a JMH feature that allows us to overwrite configurations on the fly via CLI arguments. We executed each JMH test while reducing the execution time through JMH CLI arguments<sup>2</sup>, and we store the associated JMH reports, which include the JMH configurations set by developers. To gather execution times of JUnit tests, we utilized the Maven Surefire plugin.<sup>3</sup> This plugin, an established instrument within the Java ecosystem, is tailored explicitly for running JUnit tests through Maven. Executing the tests, it produces XML reports that break down the execution time for each JUnit test.

We conducted all tests on a dedicated machine equipped with Linux Ubuntu 18.04.2 LTS, powered by a dual Intel Xeon CPU E5-2650 v3 at 2.30 GHz, boasting 40 cores and 80 GB of RAM.

**Data Wrangling ③:** To address our research questions, we focus on three key pieces of information for each Java system: (i) the entire set  $\mathcal{M}$  of Java methods appearing in source code, (ii) the set  $M_t$  of methods covered by each (JMH/JUnit) test  $t$ , and (iii) the execution time  $e_t$  of each (JMH/JUnit) test  $t$ . In the following, we describe the process used to derive this information starting from the raw data.

► *Java methods:* To extract the fully qualified names of all methods within each project, we parsed the *srcML XML files* using the *lxml* library and employed XPath queries. While executing XPath queries on XML files, facilitated by *lxml*<sup>4</sup>, we encountered a limitation regarding the size capacity. Specifically, there was a size threshold (i.e., 6,800 srcML units) surpassing which *lxml* could not evaluate the given XPath expressions. This constraint necessitated the exclusion of three projects from our analysis. Ultimately, this

<sup>2</sup>We refer the reader to [31] for a detailed explanation of this process.

<sup>3</sup><https://maven.apache.org/surefire/maven-surefire-plugin/>

<sup>4</sup><https://lxml.de/>

Research questions	Employed metrics		
RQ <sub>1</sub> , RQ <sub>2</sub>	Coverage	$C_T = \frac{ \cup_{t \in T} M_t }{ \mathcal{M} }$	$\hat{C}_T = \frac{ \cup_{t \in T} \hat{M}_t }{ \mathcal{M} }$
	Overlap Ratio	$OR_T = \frac{ \cup_{i,j \in T, i \neq j} (M_i \cap M_j) }{ \cup_{k \in T} M_k }$	
RQ <sub>2</sub>	Scope	$S_T = \frac{\sum_{t \in T}  M_t }{ T }$	
RQ <sub>3</sub> , RQ <sub>4</sub>	Total Execution Time	$TET_T = \sum_{t \in T} e_t$	
	Average Execution Time	$AET_T = \frac{\sum_{t \in T} e_t}{ T }$	

Table 1: Employed metrics

process resulted in extracting a set  $\mathcal{M}$  of Java methods for each Java system. In total, we extracted 187,319 methods from 28 distinct systems as reported in Table 2, where detailed information about the amount of *Methods*, *Benchmarks* and *Unit Tests* per project is provided.

► *Test coverage*: We leveraged the JMH and JUnit call stacks to identify the Java methods covered by each JMH benchmark or JUnit test. For each test  $t$ , we extracted the set  $M_t$  of project methods executed within  $t$ . To achieve this, we iterated over all the project methods in  $\mathcal{M}$  and checked if each method appeared after  $t$  in at least one call stack. A method  $m$  is considered covered by test  $t$  if it is invoked either directly or indirectly by  $t$  (i.e.,  $m \in M_t$ ). Additionally, for each test  $t$ , we created a separate set  $\hat{M}_t$  that only contains the methods directly called by  $t$ . In other words, for each test  $t$ , we select the methods in  $\mathcal{M}$  that appear immediately after  $t$  in the call stacks. For instance, in the example shown in Fig. 2, the method directly invoked by the benchmark `skinnyEncodeIntoCompressedByteBuffer` would be `encodeIntoCompressedByteBuffer`, but not `encodeIntoByteBuffer`.

At the end of this process, for each JMH benchmark or JUnit test  $t$ , we obtain two sets:  $M_t$  representing methods covered either directly or indirectly by  $t$ , and  $\hat{M}_t$  representing methods directly covered by  $t$ .

► *Test execution time*: The JMH configuration set by developers determines the execution time for a JMH benchmark. This configuration defines the levels of repetitions (i.e., forks, iterations, and invocations) used during benchmarking to address the inherent variability of performance measurements [19]. Invocations are repeated benchmark executions within a time-bound iteration, while a series of iterations forms a fork. Each fork usually comprises two distinct types of iterations: warmup and measurement iterations. Warmup iterations are intended to bring the fork into a steady state of performance [19, 31], while measurement iterations are the ones that are actually used for performance assessment. For each benchmark  $t$ , we first extract the JMH configuration from the JMH reports, i.e., the warmup iteration time  $w$ , the measurement iteration time  $r$ , the number of warmup iterations  $wi$ , the number of measurement iterations  $i$ , and the number of forks  $f$ . Then, we compute the associated execution time  $e_t$  accordingly:  $e_t = (w \cdot wi + r \cdot i) \cdot f$ .

For JUnit tests, we instead directly extracted the execution time  $e_t$  for each test  $t$  from the Surefire XML reports.

## 2.2 Employed metrics

Our investigation utilized various metrics, each pertinent to specific research questions as shown in Table 1, and detailed below. By answering RQ<sub>1</sub> and RQ<sub>2</sub>, we want to analyze and compare the coverage of performance tests and functional tests. To guide this analysis, we have introduced three key metrics, i.e., *Coverage*, *Overlap Ratio*, and *Scope* as defined below.

**Coverage**: We rely on method-level coverage to assess the coverage of performance/functional testing suites. We chose this coarse-grained metric rather than a fine-grained one, due to the compatibility issues between JMH and traditional statement-level code coverage tools (see Section 5 for details). A method is considered as *covered* by a testing suite if it is executed by at least one test, i.e., if there exists at least one test  $t$  such that  $m \in M_t$ . In Table 1, we formally define the coverage metric  $C_T$ , where  $T$  represents a testing suite,  $M_t$  the set of methods covered by a test  $t \in T$  and  $\mathcal{M}$  the entire set of methods of the project.  $C_T$  represents the proportion of project methods covered by at least one test. This coverage definition includes methods that might be either directly or indirectly called within a test execution, thus, we also introduce a notion of *direct coverage*. A method  $m$  is defined as *directly covered* by a suite  $T$ , if there exists at least one test  $t \in T$  such that  $m$  is directly invoked by  $t$ , i.e.,  $m \in \hat{M}_t$ . We denote *direct coverage* as  $\hat{C}_T$ , as defined in Table 1. To answer RQ<sub>1</sub>, we evaluate performance testing suites considering both *coverage* ( $C_T$ ) and *direct coverage* ( $\hat{C}_T$ ). For RQ<sub>2</sub>, we apply the same metrics to assess the *coverage* of JUnit testing suites.

**Overlap Ratio** ( $OR_T$ ): This metric measures the degree of redundancy within a JMH/JUnit testing suite. In particular,  $OR_T$  quantifies the extent of coverage overlap across different tests, thus providing a measure of redundancy in the testing suite. Table 1 provides a formal definition of this metric, where  $T$  represents a (JMH or JUnit) testing suite,  $i$  and  $j$  denote two distinct tests that belong to  $T$ ,  $M_i$  represents the set of methods covered by test  $i$ , and  $M_j$  represents the set of methods covered by test  $j$ . The numerator in  $OR_T$  denotes the methods covered in more than one test, while the denominator denotes all methods covered by the testing suite.  $OR_T$  values range from 0 to 1, where 0 indicates no overlap, i.e., each test cover distinct methods, and 1 indicates high test redundancy, i.e., all the methods are covered more than one test.

**Scope** ( $S_T$ ): This metric evaluates the coverage of performance and functional tests on an individual test basis. As detailed in Table 1, measures the average number of methods that an individual test covers within a testing suite. We use this metric because previous work has demonstrated that high coverage of tests (i.e., scope) tends to have a positive impact on the capabilities of uncovering performance issues [10].

By answering RQ<sub>3</sub> and RQ<sub>4</sub>, we analyze the time needed to perform tests belonging to a given project and compare the execution time of JMH benchmarks and JUnit tests. To this end, we defined the metrics *Total Execution Time* and *Average Execution Time* as discussed below.

**Total Execution Time** ( $TET_T$ ): It quantifies the total time cost of a (JMH or JUnit) testing suite  $T$ . It is derived by accumulating the execution times  $e_t$  for all tests  $t \in T$  (see Table 1).

*Average Execution Time (AET<sub>T</sub>):* This metric provides insights about the average execution time of tests within a suite. Its definition is outlined in Table 1, where  $|T|$  represents the number of tests in the suite  $T$ .

### 3 RESULTS DISCUSSION

In this section, we present and discuss the results of our analysis. As illustrated in Table 2, we analysed 2,190 JMH benchmarks from 28 software projects, and 256,233 JUnit tests from 19 software projects. For RQ1 and RQ3, which focus solely on performance testing, we consider all the 28 projects. For RQ2 and RQ4, which compare performance and functional testing suites, we excluded 9 projects due to technical issues encountered during the JUnit data collection (see Section 2.1 for details).

#### 3.1 RQ<sub>1</sub>: To what extent do performance tests cover the source code of software systems?

To answer RQ<sub>1</sub>, we centered our analysis of JMH benchmark coverage around two key metrics: (i) Benchmark Coverage, which measures the extent of method coverage by JMH benchmarks ( $C_T$ ), also including *direct coverage* ( $\hat{C}_T$ ); (ii) Overlap Ratio ( $OR_T$ ), which assesses the degree of redundancy in method coverage across different benchmarks.

*Benchmark Coverage:* Our analysis showed that the extent of coverage by JMH benchmarks in software projects is relatively low compared to the total number of methods. Indeed,  $C_T$  averaged 8.8% with 2.1% of methods directly covered ( $\hat{C}_T$ ) across all 28 projects.

Figure 3 shows the distribution of benchmark coverage ( $C_T$ ) across the 28 projects we examined. The y-axis represents the percentage of methods benchmarked in each project, while the x-axis enumerates the projects. The project with the highest coverage is *panda*, where 48.82% of project methods are covered. On the other

Table 2: Java systems overview

Project	GitHub Stars	Methods (Total)	Benchmarks (Total)	Unit Tests (Total)	Domain
arrow	12600	3789	34	869	Analytics Tools
byte-buddy	5800	5046	39	6357	Code Generation
cantaloupe	259	3475	103	3063	Computer Graphics
client_java	2100	386	33	217	JVM Tools
commons-beel	223	3132	3	137	JVM Tools
crate	3800	24155	39	-	Database Systems
eclipse-collections	2300	15219	515	-	Programming Utility
fastjson	25500	17524	4	4979	Parsing Library
feign	9100	979	8	913	Web Development
HdrHistogram	2100	780	12	147	Analytics Tools
imglib2	278	3432	25	635	Computer Graphics
iri	1200	1554	3	398	Data Structures
jdbi	1800	2379	76	1428	Database Systems
jetty.project	3700	18060	48	-	Web Development
jgraph	2400	3782	51	2416	Programming Utility
jooby	1600	3331	3	485	Web Development
kafka	26000	16157	27	-	Data Streaming
logbook	1600	811	20	564	Web Development
netty	31900	16615	221	-	Network Applications
objenesis	568	207	13	45	Programming Utility
panda	247	1479	4	61	Analytics Tools
protostuff	2000	3312	16	-	Programming Utility
r2dbc-h2	191	291	8	259	Database Systems
rdf4j	331	14041	14	-	Database Systems
RxJava	47300	8282	217	-	Programming Utility
SquidLib	439	5663	236	73	Computer Graphics
tinkerpop	1800	7753	57	-	Database Systems
vert.x	13800	5685	41	4095	JVM Tools

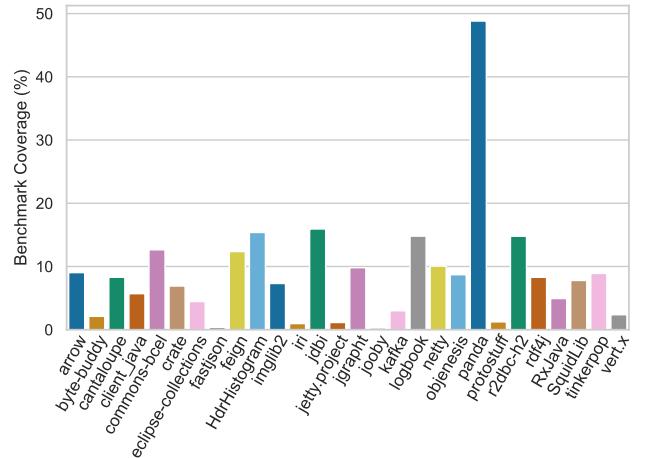


Figure 3: Coverage ( $C_T$ ) of benchmarks across projects.

hand, the *jooby* project has the lowest coverage (0.27%). The analysis indicates diverse levels of coverage among the systems with a standard deviation of 9.2%. By excluding the *panda* project, considered as an outlier, from our analysis, the benchmark coverage's standard deviation was 4.9%. This variation can be attributed to differences in the size of the projects (in terms of their number of methods, benchmarks and unit tests), to the particular development and testing practices adopted, and to the specific performance requirements and constraints.

A detailed look at *direct benchmark coverage*, depicted in Figure 4, reveals a similar trend of varied coverage. The figure shows a sensibly lower variation in the direct coverage compared to the overall coverage. However, the average direct benchmark coverage across many projects is around 2%, substantially lower than the overall benchmark coverage. This indicates that methods directly called by benchmarks often result in a large number of indirect invocations. The standard deviation for direct benchmark coverage is 1.7%, which also indicates a lower spread than the one in overall benchmark coverage. The observed difference between direct and indirect coverage may be explained by developers' tendencies to target high-level methods during performance testing. Such high-level methods often result in a higher number of indirect invocations, which might broaden the overall benchmark coverage.

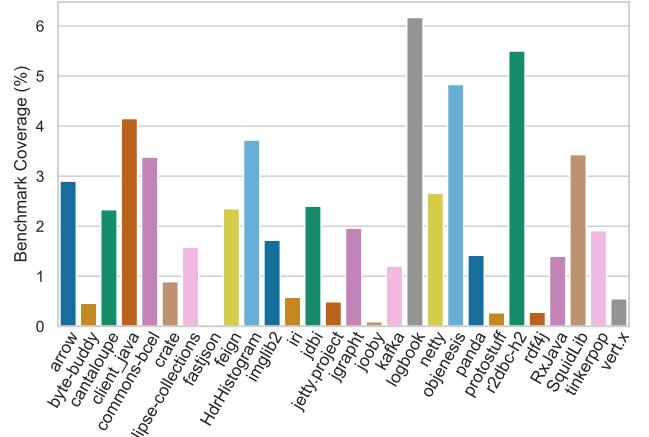
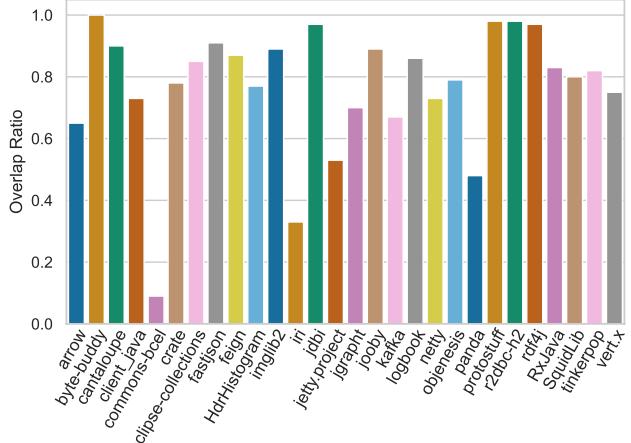


Figure 4: Direct coverage ( $\hat{C}_T$ ) of benchmarks across projects.



**Figure 5: Overlap Ratio ( $OR_T$ ) of benchmarks.**

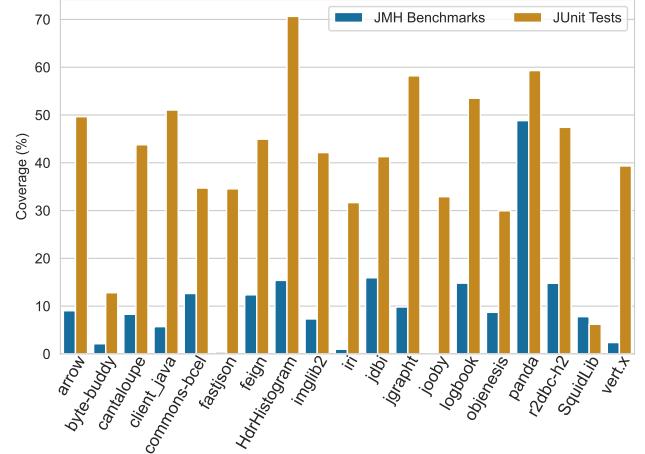
**Overlap Ratio:** Figure 5 illustrates the degree of overlap in method coverage, which represents the redundancy in method coverage across different benchmarks in the project. The overlap ratio for studied projects ranges from a minimum of 0.09 in commons-bcel to a maximum of 1 in byte-buddy. In general we can observe relatively high overlaps, with an average of 74% methods covered by more than one benchmark. While this average suggests a high degree of redundancy within performance testing suites, this may also reflect the attempts of developers to test a method under various workloads [28]. Nonetheless, our analysis suggests room for improving the efficiency of performance testing by reducing unnecessary overlap.

### 3.2 RQ<sub>2</sub>: How does the code coverage achieved by performance testing compare to that of functional testing?

In order to address RQ2, we analyzed the coverage of JUnit tests and subsequently compared it with the coverage of JMH benchmarks within our dataset of software projects.

**Comparison of JMH Benchmarks and JUnit Tests Coverage:** Figure 6 illustrates a comparison between the coverage achieved by performance testing and that of the functional testing. Clearly, the percentage of methods covered by JUnit tests tends to be higher compared to the ones of JMH benchmarks, thus indicating a broader coverage for functional testing in the projects under study. On average the coverage achieved by a performance testing suite is 4 times less than that of a functional testing suite (10.4% *versus* 41.3%). The JUnit tests coverage is significantly higher in many projects (like arrow, cantaloupe, fastjson, iri, jooby, jgraph, and vert.x) as compared to those covered by JMH benchmarks. However, it is also interesting to note an exception, i.e., the SquidLib project, where not only the JUnit Test coverage is relatively low, but also the JMH benchmark coverage exceeds the JUnit tests coverage. Upon closer examination of the project’s code, we attributed this anomaly to the performance-driven focus of the project. Specifically, SquidLib is a Java library crafted to serve as a comprehensive toolkit for the development of various gaming applications. This orientation likely leads to a greater priority being placed on performance testing over unit testing, thereby resulting in a more extensive benchmark coverage.

Similarly, Figure 7 provides a comparison of direct coverage between JMH benchmarks and JUnit tests across the same set of

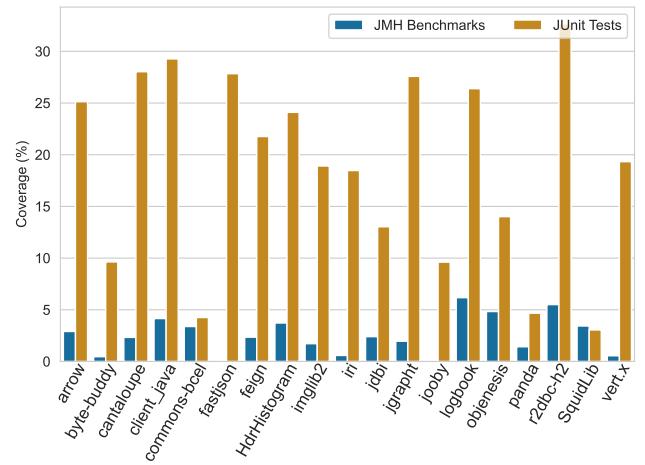


**Figure 6: Comparison of coverage ( $C_T$ ) of JMH Benchmarks and JUnit Tests across projects.**

projects. The trend is similar to the previous one. Few exceptions appear also here, such as (again) SquidLib, where direct benchmark coverage exceeds the direct coverage by unit tests. We also note substantial differences across projects in terms of the direct coverage of methods by both JMH benchmarks and JUnit tests. For instance, in the r2dbc-h2 project, the direct coverage of JUnit tests reaches 30%, whereas the one of JMH benchmarks is nearly 5%. This observation emphasizes the idea that, while functional tests might aim for broad coverage to ensure overall correctness, performance benchmarks often target specific, performance-sensitive parts of the code.

**Overlap Ratio Comparison:** Our analysis reveals that JMH benchmarks exhibit a more significant overlap in coverage compared to that of the JUnit tests. In particular, we found that JMH suites have higher overlap in coverage compared to that of the JUnit suites in 68% of the projects. We do not report the complete results in the paper due to space concerns, however we make them available in our replication package [13].

**Scope Comparison of JMH Benchmarks and JUnit Tests:** Figure 8 compares side by side, the scope ( $S_T$ ) of JMH benchmarks and that of the JUnit tests. While the coverage of performance testing suites is generally lower than that of functional testing suites, the average coverage (i.e., scope) of individual benchmarks is notably



**Figure 7: Comparison of direct coverage ( $C'_T$ ) of JMH Benchmarks and JUnit Tests across projects.**

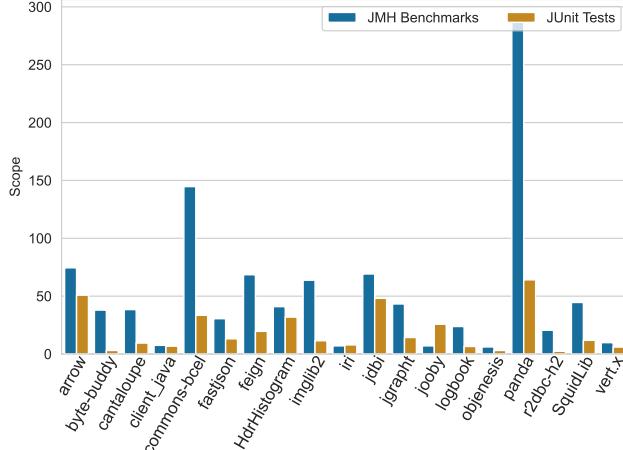
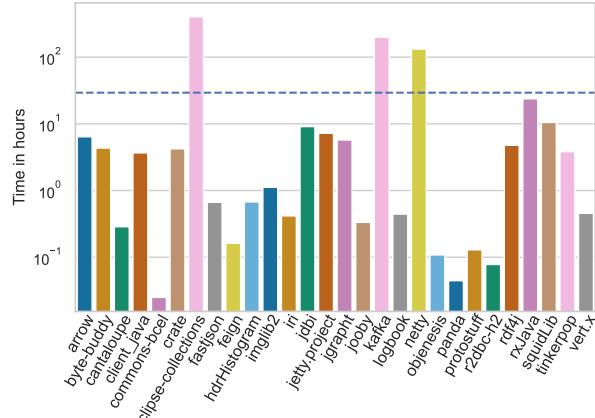
Figure 8: Scope ( $S_T$ ) of JMH Benchmarks vs. JUnit Tests.

Figure 9: JMH total execution time.

larger. Specifically, JMH benchmarks have, on average, approximately three times larger scope than their JUnit counterparts. This significant difference in the scope is more evident in projects like `panda` and `commons-bcel`. These results, once again, suggest that JMH benchmarks tend to target high-level methods, which leads to a larger number of method invocations, whereas JUnit tests tend to target lower-level methods. This finding is in line with our previous observation about the difference between direct and indirect coverage (RQ1).

### 3.3 RQ<sub>3</sub>: What is the time cost of performance testing?

In this subsection, we present our findings concerning the execution time of JMH benchmarks.

**Total Execution Time:** Figure 9 displays the time cost of performance testing suites. On the y-axis, we report the total execution time ( $TET_T$ ) in hours, using a logarithmic scale. The blue dashed line depicts the overall average time which is about 29.3 hours. The  $TET_T$  distribution varies significantly from one project to another, with some testing suites completing in just a few minutes, while others require over 100 hours for execution. The less time-consuming suite is the one of `commons-bcel`, which required only 90 seconds to run the benchmarks. Along with `commons-bcel`, only other two projects kept the  $TET_T$  under 6 minutes (i.e.,  $10^{-1}$  hours)

(namely, `panda` with 160 seconds and `r2dbc-h2` with 280 seconds). About the most time-consuming performance testing suites, few ones exceeded 10 hours. Interestingly, two projects (`squidLib` and `rxJava`) exceeded 10 hours but did not overcome the threshold of 100 hours, which was required by three projects. In particular, `eclipse-collections` required 401 hours, thus representing the most time-intensive project for performance testing.

**Average Execution Time:** In Figure 10, the bar chart depicts the average execution time of benchmarks ( $AET_T$ ) for each performance testing suite. The dashed line shows the average  $AET_T$  across projects, which is 1391.12 seconds (about 23 minutes). A close examination of this chart confirms the diversity across projects observed for  $TET_T$  also holds for  $AET_T$ . In particular, `commons-bcel`, which was the least time consuming in terms of  $TET_T$ , resulted in a relatively low  $AET_T$  of 30 seconds. Similarly, we can see that `panda` and `r2dbc-h2` maintained a low average execution time. Contrariwise, `eclipse-collections` (515 benchmarks and an  $AET_T$  of 2,805 seconds) and `netty` (221 benchmarks and an  $AET_T$  of 2,149 seconds) are very time-consuming projects, equipped with a quite high amount of benchmarks.

The most time-consuming performance testing suite is the one of `kafka`, with an  $AET_T$  of 26,503 seconds (about 7 hours per benchmark), followed by `eclipse-collections`. Interestingly, the performance testing suite of `kafka` consists of a limited number of benchmarks (i.e., 27), each of which is notably time-consuming. We investigated the JMH reports to understand the reasons behind these high  $AET_T$  values, and discovered that the likely reason is the extensive parameterization [28] of `kafka` benchmarks.

### 3.4 RQ<sub>4</sub>: How does the time cost of performance testing compare to that of functional testing?

This subsection compares the execution times of performance and functional testing.

**Total Execution Time Comparison:** In Figure 11, we compare the  $TET_T$  of JMH and JUnit testing suites. As expected, the analysis revealed that, in general, performance testing is significantly more time-consuming than functional testing. For instance, by examining `SquidLib`, the most time-consuming project for performance testing, we observe that the  $TET_T$  for the JMH suite is exponentially greater than that of the JUnit suite (37,735 seconds versus 18.5 seconds). Interestingly, the least time-consuming project in terms of functional

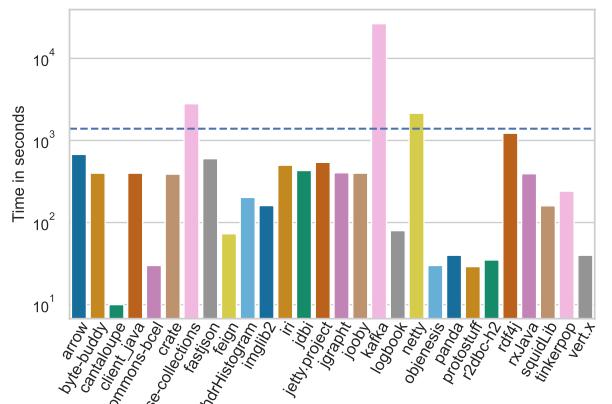


Figure 10: JMH average execution time.

testing is `vert.x` which requires 821 seconds, whereas the  $TET_T$  for the JMH suite is 1,640 seconds, i.e., more than twice the JUnit test suite execution. We can notice comparable results if we analyze less time-consuming projects. For instance, `commons-bcel` requires 90 seconds for the executing the whole JMH suite and reports a  $TET_T$  of 15.27 seconds for the JUnit suite. Another interesting case is `objenesis`, which required half a second for executing the JUnit testing suite, and 390 seconds for executing the JMH suite. Performance testing suites have on average a time cost 62 times higher than that of functional testing suites.

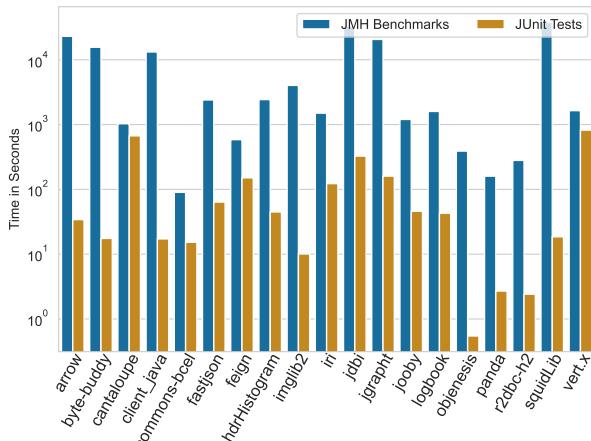


Figure 11: Total execution time comparison.

**Average Execution Time Comparison:** There is a significant difference in test-level execution time between JMH and JUnit tests across all projects. We found that JMH benchmarks exhibit an average  $AET_T$  of approximately 245.9 seconds. On the other side, the average  $AET_T$  of JUnit tests does not exceed 0.098 seconds. On average, an individual JMH benchmark demands about 2,507 times the execution time required for executing one JUnit test. We report the complete results related to  $AET_T$  of JUnit tests in our replication package [13].

## 4 IMPLICATIONS

This section discusses some implications of this study along with some directions for future work.

**For practitioners.** This study gives clear evidence that a significant portion of the codebase of many software systems lacks performance assessment. As software evolves, these code areas may become vulnerable to performance bugs that remain undetected until released. One potential reason for this oversight could be the limited availability of tools for performance test coverage. Indeed, we are unaware of any tools that measure performance test coverage as seamlessly as tools like JaCoCo do for functional testing. We believe that introducing such tools could allow developers to more regularly assess the coverage of their performance testing suites, thus increasing their awareness of the code area that remains unmonitored for performance.

Our results also reveal that distinct performance tests often cover the same code components (i.e., high overlap), even though a significant portion of the codebase remains uncovered by any performance test. Although developers might deliberately target the same

code components with multiple performance tests to assess their behaviour under varying workloads, this highlights an opportunity to broaden test coverage without incurring additional time costs. We hypothesize that, by raising awareness about performance test coverage, developers might be more inclined to prioritize creating tests that target code components currently unassessed for performance. We encourage future work to make it easier for practitioners to measure the coverage of performance testing suites.

**For researchers.** Prior work suggests that the high costs of test development and maintainability often hamper the adoption of performance testing in practice [16, 23, 30]. In response to this issue, researchers have introduced techniques capable of automatically transforming functional testing suites into performance tests [16]. In light of our results, we can formulate educated guesses regarding the potential benefits of these techniques, as well as the challenges that might originate from their adoption. For instance, our results suggest that, by utilizing automated performance test generation, there could be a significant improvement in terms of performance test coverage. Indeed, functional testing suites exhibit significantly higher coverage than that of performance tests (10.4% vs 41.3% on average), and generated performance tests would inherit such high coverage. However, these benefits might come at a cost, particularly regarding execution time. The high coverage of functional test suites is typically a consequence of their extensive sizes, which may not be feasible for a performance testing suite. In fact, performance tests are typically more time-consuming than functional counterparts (on average 2,507 times more). For instance, by using the configuration defined by Jangali *et al.* [16] and a typical number of five forks [22, 31], the time cost of an individual generated performance test would amount to about 400 seconds. For a medium-sized testing suite like `logbook`, which comprises 564 JUnit tests, this translates to a total time cost of roughly two and a half days. This is approximately 141 times longer than the actual `logbook` performance testing suite. Even when considering the smallest functional testing suite in our study, namely `objenesis`, this results in a 5-hour execution time, i.e., 55 times the one of the actual performance testing suite. These findings highlight that automated generation alone might not be sufficient to produce performance test suites that are practically usable, given that developers might be deterred by such a time-consuming test process. This underscores a significant challenge for the research community, i.e., automated performance test generation should take into consideration the associated time cost of the generated testing suite.

A potential research avenue is the adoption of “smart” test selection strategies that aim to maximize coverage while mitigating time costs. For instance, one could exploit data on the functional test coverage to reduce the number of redundant performance tests targeting the same code component. Future work should be directed to address this challenge.

## 5 THREATS TO VALIDITY

**Construct validity.** We focused solely on *Java software systems*. Our results may not generalize to systems developed in other programming languages. Nevertheless, Java is still among the most used programming languages<sup>5</sup>. We restricted the coverage analysis

<sup>5</sup>Stack Overflow Developer Survey, <https://survey.stackoverflow.co/2023>.

to JMH microbenchmarks and JUnit tests since they are mature and widely adopted frameworks for developing performance and functional testing, respectively. Moreover, both these frameworks operate at the fine-grained level, as they are both used to test individual methods within a codebase. This commonality provides a fair basis for comparing their test coverage.

Using method-level coverage may have limitations, since this metric does not account for cases where performance/functional tests only partially cover the method statements. Our study results may change when employing a statement-level coverage metric. The decision to use method-level coverage stems from the significant technical challenges encountered in integrating JMH with traditional statement-level coverage tools, such as JaCoCo and Cobertura. To obtain statement-level coverage information, these tools modify the Java bytecode, which we observed could interfere with the execution of JMH microbenchmarks. Given these challenges, method-level coverage was deemed a reasonable compromise between the practicality of the study and the representativeness of the results. Furthermore, method-level coverage has been extensively employed in software performance research [7, 8, 32].

**External validity.** The presented analysis is limited to 28 open-source software systems. The findings may not be broadly generalizable; nonetheless, the selected systems are all well-known Java systems encompassing different domains (e.g., database systems, logging frameworks, and web servers). This limited number of subject systems is also motivated by an effort-intensive data collection, which required months of work (in multiple iterations) to get reliable results. This is a known issue in performance engineering that typically restricts the number of subject systems in empirical studies. Nevertheless, the number of subject systems used in our study is larger than most of the recent empirical studies on performance (e.g., see [8, 10, 16, 21, 22]).

**Internal validity.** We used a sampling-based CPU profiler to identify the methods covered by tests. These profilers operate by periodically capturing a program’s call stack during its execution. A limitation of this approach is the potential omission of call stack information. Since sampling is done at discrete intervals, short-lived function calls or those that fall between sampling points might not be captured. To mitigate this threat, we used async-profiler, which (to our knowledge) provides the lowest sampling rate (i.e., 1 nanosecond) for profiling Java software.

## 6 RELATED WORK

**Performance Testing.** The study most closely related to our work is that of Laaber and Leitner [21], which proposes a performance test quality metric inspired by mutation testing score, namely API benchmarking score (ABS). ABS is related to the concept of test coverage, as it represents the capability of the performance testing suite to find slowdowns. While Laaber and Leitner focus on defining a novel metric for test quality, our research evaluates the quality of existing performance testing suites using traditional code coverage metrics. Traini *et al.* [32] show that code components covered by performance tests tend to be less susceptible to refactoring. The time cost of performance testing is also related to this work. Researchers proposed approaches to reduce the time cost of performance testing without sacrificing results quality [1, 12, 19, 22].

**Test Coverage.** In [15], the authors describe Google’s code coverage infrastructure and how the computed code coverage information is visualized and used. The study demonstrates that most of the projects contain few unit tests, despite the opposite perception of the developers. The authors in [37] analyzed test coverage data on several widely used Python projects. The main finding is that the coverage strongly depends on the control flow structure. Moreover, the authors found that error-handling code is also neglected. In [11], the authors examine the question of coverage criteria as suite quality predictors from the perspective of the non-researcher audience. Alves *et al.* [3] conceived an approach for estimating code coverage through static analysis, particularly slicing of call graphs.

**Performance Bugs.** Jin *et al.* [18] empirically studied 110 real-world performance bugs collected from 5 open-source software repositories. A more extensive study was recently conducted by Zhao *et al.* [38], which investigated 570 performance issues from 13 open-source projects. Other empirical studies have focused on more specific domains, such as internet browsers [36], mobile applications [25], and JavaScript applications [29]. While our empirical findings may not directly correlate with the ability to uncover performance issues, there are strong indications of the relevance of code coverage for the efficacy of performance testing. Batch *et al.* [4] found that source code covered by functional/performance tests is less prone to bugs. Ding *et al.* [10] showed that the code coverage of tests (i.e., scope) influences their capability to uncover performance bugs. The study of Jangali *et al.* [16] suggests that the extension of code coverage in performance testing improves the capability of detecting performance issues. These works indicate that incorporating code coverage analysis into performance testing can be beneficial for software performance assurance.

## 7 CONCLUSION AND FUTURE WORK

This paper presented a comprehensive empirical study focused on performance testing coverage. Our findings revealed the limited coverage of current performance testing suites and the significant time cost associated with them. The results of this work suggest opportunities to enhance the coverage of performance testing suites, by emphasizing the necessity to enlighten practitioners about these prevalent limitations.

We have intentionally considered in this paper the concept of code coverage that usually relates to functional testing. Additional metrics should be considered for a sharper concept of performance test coverage, like workload and operational profile. However, these metrics are quite difficult to collect and may sensibly vary for the same application in different contexts. Therefore, we have intended to explore the extent at which performance testing can be solely based on code coverage.

As suggested by our findings, the real-world adoption of performance testing techniques might be hampered by their substantial time costs. The evidence provided in this paper sustains the idea that the limited coverage of performance tests (as compared to functional ones) only stems from technical issues (e.g., time limits, problems to collect dynamic metrics). Indeed, a wider coverage is desirable as it would allow to identify performance issues in code sections that, for the above reasons, are usually not considered.

Therefore, we encourage further research to address this challenge, possibly leveraging the automated generation of performance tests. One potential direction, indeed, could be the development of “smart” test selection strategies that can reduce the execution time of a performance testing suite without compromising its effectiveness, thus facilitating a smooth transition of automated performance test generation to practice.

## ACKNOWLEDGMENTS

This work is partially supported by Italian Government (Ministero dell’Università e della Ricerca, PRIN 2022 PNRR): “RECHARGE: monitoRing, tEsting, and CHaracterization of performAne Regressions” (cod.P2022SEL A7), and by “ICSC – Centro Nazionale di Ricerca in High Performance Computing, Big Data and Quantum Computing”, funded by European Union – NextGenerationEU.

## REFERENCES

- [1] Hammam M. Alghmadi, Mark D. Syer, Weiyi Shang, and Ahmed E. Hassan. 2016. An Automated Approach for Recommending When to Stop Performance Tests. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 279–289. <https://doi.org/10.1109/ICSME.2016.46>
- [2] Wasim Alsqaqf, Maya Daneva, and Roel Wieringa. 2019. Quality requirements challenges in the context of large-scale distributed agile: An empirical study. *Information and Software Technology* 110 (2019), 39 – 55.
- [3] Tiago L Alves and Joost Visser. 2009. Static estimation of test coverage. In *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*. IEEE, 55–64.
- [4] Thomas Bach, Artur Andrzejak, Ralf Pannemanns, and David Lo. 2017. The impact of coverage on bug density in a large industrial software project. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 307–313.
- [5] Woubshet Behutiye, Pertti Karhápää, Lidia López, Xavier Burgués, Silverio Martínez-Fernández, Anna Maria Vollmer, Pilar Rodríguez, Xavier Franch, and Markku Oivo. 2020. Management of quality requirements in agile and rapid software development: A systematic mapping study. *Information and Software Technology* 123 (2020), 106225.
- [6] Jake Brutlag. 2009. Google AI Blog: Speed matters. <https://ai.googleblog.com/2009/06/speed-matters.html>
- [7] Jinfu Chen, Zishuo Ding, Yiming Tang, Mohammed Sayagh, Heng Li, Bram Adams, and Weiyi Shang. 2023. IoPV: On Inconsistent Option Performance Variations. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2023)*. ACM, 845–857. <https://doi.org/10.1145/3611643.3616319>
- [8] Jinfu Chen, Weiyi Shang, and Emad Shihab. 2022. PerfJIT: Test-Level Just-in-Time Prediction for Performance Regression Introducing Commits. *IEEE Transactions on Software Engineering* 48, 5 (2022), 1529–1544.
- [9] Michael L. Collard, Michael J. Decker, and Jonathan I. Maletic. 2011. Lightweight Transformation and Fact Extraction with the srcML Toolkit. In *2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation*. 173–184. <https://doi.org/10.1109/SCAM.2011.19>
- [10] Zishuo Ding, Jinfu Chen, and Weiyi Shang. 2020. Towards the use of the readily available tests from the release pipeline as performance tests: are we there yet?. In *ICSE ’20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 1435–1446. <https://doi.org/10.1145/3377811.3380351>
- [11] Rahul Gopinath, Carlos Jensen, and Alex Groce. 2014. Code Coverage for Suite Evaluation by Developers. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, 72–82.
- [12] Sen He, Glenna Manns, John Saunders, Wei Wang, Lori Pollock, and Mary Lou Soffa. 2019. A Statistics-Based Performance Testing Methodology for Cloud Applications. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)*. ACM, 188–199.
- [13] Muhammad Imran, Vittorio Cortellessa, Davide Di Ruscio, Riccardo Rubei, and Luca Traini. 2023. An Empirical Study on Performance Test Coverage-Replication Package. [https://github.com/SpencerLabAQ/replication-package\\_performance-test-coverage](https://github.com/SpencerLabAQ/replication-package_performance-test-coverage)
- [14] Laura Inozemtseva and Reid Holmes. 2014. Coverage is Not Strongly Correlated with Test Suite Effectiveness. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, 435–445.
- [15] Marko Ivanković, Goran Petrović, René Just, and Gordon Fraser. 2019. Code Coverage at Google. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)*. ACM, 955–963.
- [16] Mostafa Jangali, Yiming Tang, Niclas Alexandersson, Philipp Leitner, Jinqiu Yang, and Weiyi Shang. 2023. Automated Generation and Evaluation of JMH Microbenchmark Suites From Unit Tests. *IEEE Transactions on Software Engineering* 49, 4 (2023), 1704–1725. <https://doi.org/10.1109/TSE.2022.3188005>
- [17] Zhen Ming Jiang and Ahmed E. Hassan. 2015. A Survey on Load Testing of Large-Scale Software Systems. *IEEE Transactions on Software Engineering* 41, 11 (2015), 1091–1118. <https://doi.org/10.1109/TSE.2015.2445340>
- [18] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and Detecting Real-World Performance Bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI ’12)*. ACM, 77–88. <https://doi.org/10.1145/2254064.2254075>
- [19] Tomas Kalibera and Richard Jones. 2013. Rigorous Benchmarking in Reasonable Time. In *Proceedings of the 2013 International Symposium on Memory Management (ISMM ’13)*. ACM, 63–74. <https://doi.org/10.1145/2491894.2464160>
- [20] Christoph Laaber, Harald C. Gall, and Philipp Leitner. 2021. Applying test case prioritization to software microbenchmarks. *Empirical Software Engineering* 26, 6 (2021), 133. <https://doi.org/10.1007/s10664-021-10037-x>
- [21] Christoph Laaber and Philipp Leitner. 2018. An Evaluation of Open-Source Software Microbenchmark Suites for Continuous Performance Assessment. ACM.
- [22] Christoph Laaber, Stefan Würsten, Harald C. Gall, and Philipp Leitner. 2020. Dynamically Reconfiguring Software Microbenchmarks: Reducing Execution Time without Sacrificing Result Quality. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*. ACM, 989–1001.
- [23] Philipp Leitner and Cor-Paul Bezemer. 2017. An Exploratory Study of the State of Practice of Performance Testing in Java-Based Open Source Projects. In *Proceedings of the 8th ACM/SPEC International Conference on Performance Engineering (ICPE ’17)*. ACM, 373–384. <https://doi.org/10.1145/3030207.3030213>
- [24] Aleksander Maricq, Dmitry Duplyakin, Ivo Jimenez, Carlos Maltzahn, Ryan Stutsman, and Robert Ricci. 2018. Taming Performance Variability. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 409–425.
- [25] Alejandra Mazuera-Rozo, Catia Trubiani, Mario Linares-Vásquez, and Gabriele Bavota. 2020. Investigating types and survivability of performance bugs in mobile apps. *Empirical Software Engineering* 25 (2020), 1644–1686.
- [26] Steve Olenki. 2016. Why Brands Are Fighting Over Milliseconds. <https://www.forbes.com/sites/steveolenki/2016/11/10/why-brands-are-fighting-over-milliseconds/>
- [27] Paul Piwowarski, Mitsuhiro Ohba, and Joe Caruso. 1993. Coverage Measurement Experience during Function Test. In *Proceedings of the 15th International Conference on Software Engineering (ICSE ’93)*. IEEE Computer Society Press, 287–301.
- [28] Hazem Samoaa and Philipp Leitner. 2021. An Exploratory Study of the Impact of Parameterization on JMH Measurement Results in Open-Source Projects. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE ’21)*. ACM, 213–224. <https://doi.org/10.1145/3427921.3450243>
- [29] Marija Selakovic and Michael Pradel. 2016. Performance issues and optimizations in javascript: an empirical study. In *Proceedings of the 38th International Conference on Software Engineering*. 61–72.
- [30] Luca Traini. 2022. Exploring Performance Assurance Practices and Challenges in Agile Software Development: An Ethnographic Study. *Empirical Software Engineering* 27, 3 (2022), 74. <https://doi.org/10.1007/s10664-021-10069-3>
- [31] Luca Traini, Vittorio Cortellessa, Daniela Di Pompeo, and Michele Tucci. 2022. Towards effective assessment of steady state performance in Java software: are we there yet? *Empirical Software Engineering* 28, 1 (2022), 13. <https://doi.org/10.1007/s10664-022-10247-x>
- [32] Luca Traini, Daniela Di Pompeo, Michele Tucci, Bin Lin, Simone Scalabrin, Gabriele Bavota, Michele Lanza, Rocco Oliveto, and Vittorio Cortellessa. 2021. How Software Refactoring Impacts Execution Time. *ACM Trans. Softw. Eng. Methodol.* 31, 2, Article 25 (dec 2021), 23 pages. <https://doi.org/10.1145/3485136>
- [33] Andrew Walker, Michael Coffey, Pavel Tisnovsky, and Tomas Cerny. 2020. On Limitations of Modern Static Analysis Tools. In *Information Science and Applications*, Kuinam J. Kim and Hye-Young Kim (Eds.). Springer Singapore, 577–586.
- [34] Elaine J. Weyuker. 2000. Experience with performance testing of software systems: issues, an approach, and case study. *IEEE Transactions on Software Engineering* 26, 12 (2000), 1147 – 1156. <https://doi.org/10.1109/32.888628>
- [35] S. Yoo and M. Harman. 2012. Regression Testing Minimization, Selection and Prioritization: A Survey. *Softw. Test. Verif. Reliab.* 22, 2 (mar 2012), 67–120.
- [36] Shahed Zaman, Bram Adams, and Ahmed E. Hassan. 2012. A Qualitative Study on Performance Bugs. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories (MSR ’12)*. IEEE Press, 199–208.
- [37] Hongyu Zhai, Casey Casalnuovo, and Prem Devanbu. 2019. Test coverage in python programs. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 116–120.
- [38] Yutong Zhao, Lu Xiao, Andre B. Bondi, Bihuan Chen, and Yang Liu. 2023. A Large-Scale Empirical Study of Real-Life Performance Issues in Open Source Projects. *IEEE Transactions on Software Engineering* 49, 2 (2023), 924–946.