An exploratory study of the state of practice of performance testing in Java-based open source projects

The usage of open source (OS) software is nowadays wide- spread across many industries and domains. While the functional quality of OS projects is considered to be up to par with that of closed-source software, much is unknown about the quality in terms of non-functional attributes, such as performance. One challenge for OS developers is that, unlike for functional testing, there is a lack of accepted best practices for performance testing. To reveal the state of practice of performance testing in OS projects, we conduct an exploratory study on 111 Java-based OS projects from GitHub. We study the performance tests of these projects from five perspectives: (1) the developers, (2) size, (3) organization and (4) types of performance tests and (5) the tooling used for performance testing. First, in a quantitative study we show that writing performance tests is not a popular task in OS projects: performance tests form only a small portion of the test suite, are rarely updated, and are usually maintained by a small group of core project developers. Second, we show through a qualitative study that even though many projects are aware that they need performance tests, developers appear to struggle implementing them. We argue that future performance testing frameworks should provider better support for low-friction testing, for instance via non-parameterized methods or performance test generation, as well as focus on a tight integration with standard continuous integration tooling.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2496v2 | CC BY 4.0 Open Access | rec: 6 Oct 2016, publ: 6 Oct 2016

An Exploratory Study of the State of Practice of Performance Testing in Java-Based Open Source Projects[*]

Philipp Leitner
Department of Informatics
University of Zurich
Switzerland
leitner@ifi.uzh.ch

Cor-Paul Bezemer
Software Analysis and Intelligence Lab (SAIL)
Queen's University
Canada
bezemer@cs.queensu.ca

ABSTRACT

The usage of open source (OS) software is nowadays widespread across many industries and domains. While the functional quality of OS projects is considered to be up to par with that of closed-source software, much is unknown about the quality in terms of non-functional attributes, such as performance. One challenge for OS developers is that, unlike for functional testing, there is a lack of accepted best practices for performance testing.

To reveal the state of practice of performance testing in OS projects, we conduct an exploratory study on 111 Java-based OS projects from GitHub. We study the performance tests of these projects from five perspectives: (1) the developers, (2) size, (3) organization and (4) types of performance tests and (5) the tooling used for performance testing.

First, in a quantitative study we show that writing performance tests is not a popular task in OS projects: performance tests form only a small portion of the test suite, are rarely updated, and are usually maintained by a small group of core project developers. Second, we show through a qualitative study that even though many projects are aware that they need performance tests, developers appear to struggle implementing them. We argue that future performance testing frameworks should provider better support for low-friction testing, for instance via non-parameterized methods or performance test generation, as well as focus on a tight integration with standard continuous integration tooling.

Keywords

performance testing; performance engineering; open source; mining software repositories; empirical software engineering

1. INTRODUCTION

The usage of open source (OS) software libraries and components is by now widely spread across many, if not all, industries and domains. Studies claim that, from a functional perspective, the quality of OS software is on par with comparable closed-source software [1]. OS projects have largely converged against accepted best practices [5] (e.g., unit testing) and widespread standard tooling for functional testing, such as JUnit in the Java ecosystem.

However, the quality of OS software in terms of non-functional attributes, such as reliability, scalability, or performance, is less well-understood. For example, Heger et al. [10] state that performance bugs in OS software go undiscovered for a longer time than functional bugs, and fixing them takes longer. One reason for the longer fixing time may be that performance bugs are notoriously hard to reproduce [26].

As many OS software libraries (such as `apache/log4j` or the `apache/commons` collection of libraries) are used almost ubiquitously across a large span of other OS or industrial applications, a performance bug in such a library can lead to widespread slowdowns. Hence, it is of utmost importance that the performance of OS software is well-tested.

Despite this importance of performance testing for OS software, our current understanding of how developers are conducting performance, stress, or scalability tests is lacking. There is currently no study that analyzes whether and how real projects conduct performance testing, which tools they use, and what OS software developers struggle with. As such, there exist no guidelines for how OS developers should test the performance of their projects.

In this paper, we conduct an exploratory study on the state of practice of performance testing in Java-based OS software. We study 111 Java-based projects from GitHub that contain performance tests. We focus on five perspectives:

1. **The developers who are involved in performance testing.** In most studied OS projects, a small group of core developers creates and maintains the performance tests. Our findings suggest that in general, there are no developers in the studied OS projects that focus solely on performance testing.
2. **The extend of performance testing.** In most studied OS projects, the performance tests are small in terms of lines of code, and do not change often. We did not observe a statistically significant difference in the size of the performance tests of projects that make claims about their performance (e.g., "fastest implementation of X") and projects that do not make such claims.
3. **The organization of performance tests.** The Java OS software community has not yet converged against

[*] This paper is submitted as a full research paper

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

a common understanding of how to conduct and organize performance tests. Developers freely mix performance tests with unit tests and code comments are used to describe how a performance test should be executed.

4. **The types of performance tests.** Half of the studied OS projects have one or two performance smoke tests for testing the performance of the main functionality of a project. 36% of the studied projects use microbenchmarks for performance testing. Less popular types of performance tests are one-shot performance tests (i.e., tests that focus on one very specific performance issue), performance assertions and implicit performance tests (i.e., measurements that are done during the execution of functional tests).
5. **The tooling and frameworks used for performance tests.** While there exist dedicated tools and frameworks for performance tests, the adoption of these tools and frameworks is not widespread in the Java OS software community. Only 16% of the studied projects uses a dedicated framework such as `JMH` or `Caliper`.

Our findings imply that practitioners who use OS software in their projects must thoroughly test the consequences of doing so on the performance of their own projects, as developers should not assume that OS software necessarily follows stringent performance testing practices. From our exploratory study follows that writing performance tests is not a popular task in OS projects. Performance tests form only a small portion of the test suite, are rarely updated, and are usually maintained by a small group of core project developers. Further, we argue that future performance testing frameworks should provider better support for low-friction testing, for instance via non-parameterized methods or performance test generation, as well as focus on a tight integration with standard continuous integration (CI) tooling.

The remainder of this paper is structured as follows. In Section 2, we summarize related work. In Section 3, we describe our exploratory study setup and research methodology. Section 4 contains our study results, followed by a discussion of the main implications resulting from our work in Section 5. Section 6 discusses the main threats to the validity of our study. Finally, Section 7 concludes the paper.

2. BACKGROUND AND RELATED WORK

In this section, we will discuss work that is related to our exploratory study. The overall body of research on performance testing is too broad to cover here in its entirety. Therefore, we focus on work that has informed our study or is particularly related to it.

Empirical Studies. As a non-functional quality attribute, performance is often equally critical to the perceived value of a program as functional correctness is. For instance, in a cloud computing context, writing inefficient, low-performance code often directly translates to higher operations costs, as more cloud resources need to be used to deliver the same end-user perceived quality (e.g., response time) [7]. Consequently, (at least) two decades of research have led to great insights into how to design and construct performance-optimal systems [23]. A number of empirical studies have recently analyzed the anatomy of reported performance problems [9, 15, 26], arguing that these problems take longer to get reported and fixed than functional problems. Further, recent research on performance regression mining (a research method inspired by software repository mining that repeatedly benchmarks different revisions of a software system to discover historical, unreported, performance bugs) has shown that performance problems can originate from a wide range of code changes, including simple updates of dependencies [2, 22]. A system following a similar basic operating principle is `PerfImpact` [18], which aims to find changes and input combinations that lead to performance regressions. PRA (performance risk analysis) is an approach used to narow down commits that led to a (previously detected) performance regression [13]. Baltes et al. [4] study how developers locate performance problems, and conclude that standard tools do not provide sufficient support for understanding the runtime behavior of software.

Problems of Performance Testing. Early on, studies have reported that industrial practice in performance testing is not on the same level as functional testing [25], even for large, performance-sensitive enterprise applications. Historically, performance testing has been made difficult by two peculiarities. Firstly, performance testing of higher-level programming languages, including interpreted languages or those running in a virtual machine as in the case of Java, is difficult due to the high number of confounding factors introduced by features of the program runtime environment. For instance, in the case of Java, just-in-time compilation, hardware platforms, virtual machine implementations, or garbage collection runs can all significantly impact test results [8]. Secondly, performance test results often depend strongly on the used benchmarking workload, such as the load patterns used for testing. Hence, writing expressive performance tests requires careful identification and modeling of representative workloads or production usage patterns [3,14]. Unfortunately, representative workloads tend to not be as stable as functional unit interfaces. Consequently, workloads need continuous validation and maintenance [24].

Industrial Approaches. Besides these more academic research attempts, there are also a number of industrial-strength tools available in the domain of software performance testing. For instance, Caliper, an older Java-based framework from Google, allows developers to write performance tests using a JUnit-like API, and provides some built-in support for metric collection and aggregation. However, many of the problems of benchmarking Java-based software still exist when using Caliper. For this reason, OpenJDK contains since version 7 a proprietary extension called the Java Microbenchmarking Harness (JMH). Not unlike Caliper, JMH allows developers to build performance tests using a simple annotation-based syntax. However, given that JMH is part of the Java virtual machine rather than a library, JMH is able to control for some of the problems of Java benchmarking. For instance, JMH is able to control for Just-in-Time compilation or garbage collection runs. Both, Caliper and JMH, are largely intended for low-level code benchmarking. For performance testing on systems level (especially for Web-based systems), a number of additional tools are available that foster the definition and execution of workloads, such as HTTP sessions. Arguably, the two most prevalent OSS representatives of this class of tools are Apache JMeter[1] and Faban[2]. Both allow the definition and execution of complex workloads, and also contain facilities

[1] http://jmeter.apache.org

[2] http://faban.org

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2496v2 | CC BY 4.0 Open Access | rec: 6 Oct 2016, publ: 6 Oct 2016

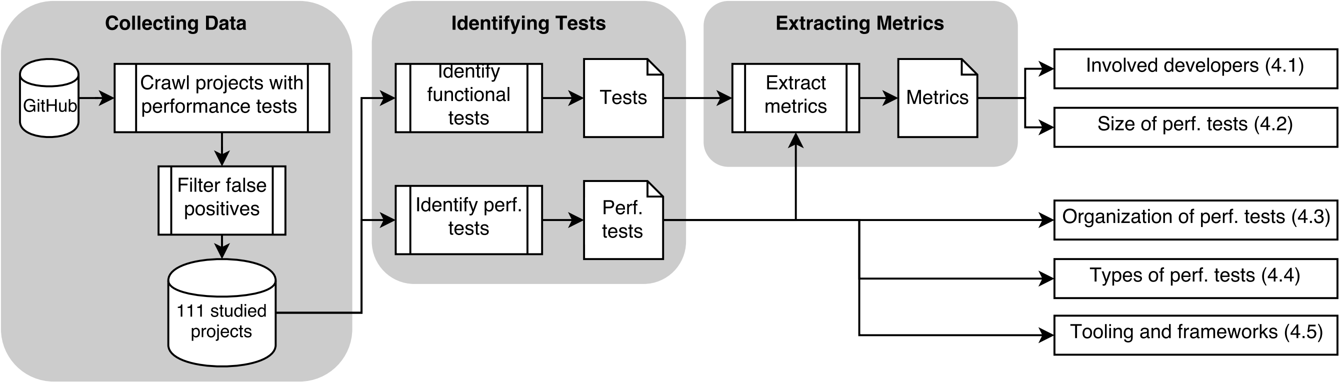
Collecting Data
GitHub
Crawl projects with performance tests
Filter false positives
111 studied projects
Identifying Tests
Identify functional tests
Tests
Identify perf. tests
Perf. tests
Extracting Metrics
Extract metrics
Metrics
Involved developers (4.1)
Size of perf. tests (4.2)
Organization of perf. tests (4.3)
Types of perf. tests (4.4)
Tooling and frameworks (4.5)

Figure 1: Overview of our exploratory study approach.

to collect performance test results. JMeter, for instance, exports detailed performance test outcomes in an XML or CSV file, which can then be used in other tools, such as dashboards (e.g., the Performance plugin for the Jenkins CI system[3] or for statistical analysis.

In our research, we contribute to the state of the art via an exploratory study that establishes to what extend these industrial tools, as well as the concepts and ideas originating from earlier research on performance testing, have been adopted in Java-based OS software. Hence, our study serves as a reality check and can guide future research directions.

3. EXPLORATORY STUDY SETUP

In this section, we describe the setup of our exploratory study and our methodology for collecting data and identifying performance tests. Figure 1 depicts the steps of our methodology, which are detailed below.

3.1 Collecting Data

We aim to generate a data set of OS projects that conduct performance testing. We build a custom crawler for GitHub which uses the GitHub search engine to search for OS projects using a combination of search heuristics. Our crawler searches for Java projects with tests in the path `src/test`, which use one or more terms in the test file name or source code (i.e., "bench" or "perf") or which imports known performance testing frameworks, such as `JMH` or `Caliper`. We have identified these search heuristics to be simple yet reasonably robust ways to identify projects with performance tests as part of a pre-study, where we manually inspected known projects with performance tests, e.g., `apache/log4j` or `ReactiveX/RxJava`. Our crawler extracts an initial data set of 1697 candidate projects. From those candidate projects, we filter forked projects and projects with less than 10 "stars" or less than 50 commits on GitHub, resulting in a data set of 154 projects.

Finally, we manually identify and discard false positives, i.e., projects that match our search heuristics but, upon manual inspection, turn out to not contain performance tests after all. Examples for false positives include projects that have domain objects containing the word "bench", or projects that themselves implement performance testing frameworks (surprisingly, these projects often do not provide performance tests themselves). To foster the exploratory nature of our study, we are inclusive in our manual filtering and use a broad definition of "performance test". For example, we include projects in this step which simply report on runtime performance as part of regular unit testing. After discarding false positives, our data set finally consists of 111 Java-based OS projects that actually contain performance tests.

In addition, we manually study the GitHub description of the projects to classify them as performance-sensitive (PS) or not performance sensitive (not PS). We categorize projects as PS if they self-identify as the fastest implementation of a given functionality, or stress having high performance in their project description. For example, the project `nativelibs4java/BridJ` describes itself as "blazing fast", which would classify the project as PS. Our expectation is that PS projects dedicate more effort to performance testing than non-PS projects to support the claims that PS projects make about their performance. Using this metric, we classify 26 of our projects (14%) as PS, and 85 projects (76%) as "not PS".

Projects in the data set include a healthy mix of well-known large projects (e.g., `apache/hbase`) and smaller, lesser-known ones (e.g., `TridentSDK/Trident`, a Java-based Minecraft server). A summary of statistics about the projects in the data set is provided in Figure 2. Data is reported as available via GitHub at the time of collection, i.e., in July 2016. Furthermore, the entire data set is available an online appendix[4].

3.2 Identifying Tests

We manually identify which files contain performance tests in the studied projects. First, we manually identify the test suite of the project, which is `src/test` in most cases by construction. However, in some projects, the `src/test` directory contains resource files that we manually exclude from our study. We search the test files for Java source code files that contain the terms "perf" or "bench" and we manually validate that the tests are indeed performance tests. Further, we sample the remaining test and source code in an exploratory manner for additional performance tests not identified via our heuristics. The set of identified test files and performance test files is available in the online appendix.

[3] https://wiki.jenkins-ci.org/display/JENKINS/Performance+Plugin

[4] https://xleitix.github.io/appendix_perf_tests/

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2496v2 | CC BY 4.0 Open Access | rec: 6 Oct 2016, publ: 6 Oct 2016

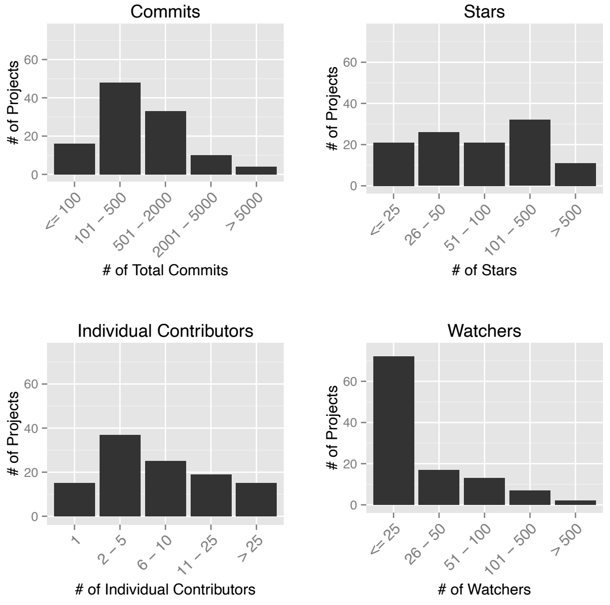

Figure 2: Summary statistics of the 111 OS projects in the data set. The subfigures show the age of the projects in number of total commits, the popularity of the projects as measured by the number of stars on GitHub and the number of subscribers ("watchers"), and the number of individual contributors.

3.3 Extracting Metrics

We extract several metrics that describe the effort that is dedicated to the performance and functional test suites. In the remainder of this section, we explain how we extract these metrics. Our scripts for extracting and analyzing these metrics are available in the online appendix.

3.3.1 Developer Metrics

We extract the names and number of commits of the developers who work on the performance tests, tests and project. In addition, we compute the proportion of performance test developers who are core (test) developers of the project.

- *Names and number of commits of developers:* We run the command `git shortlog -s -n HEAD %FILES%` for each studied project to get a list of developers of files `%FILES%` that is ordered by the number of commits that a developer made to those files. Table 1 shows examples of these lists for the ↗ `SoftInstigate/restheart` project.
- *Proportion of performance test developers who are core (test) developers:* We calculate the number of performance test developers who are in the top-n project developers by the number of commits. We define n as the number of performance test developers in the project. Table 1 shows the performance test developers, test developers and project developers in ↗ `SoftInstigate/restheart`. The three performance test developers are *Andrea di Cesare*, *Maurizio Turatti* and *gokrokvertskhov*. However, only *Andrea di Cesare* and *Maurizio Turatti* are in the top-3 of project developers, hence the proportion of performance test developers who are core developers is 0.67. The proportion of test developers who are core developers is 0.5, as *Stephan Windmuller* and *gokrokvertskhovh* are not in the top-4 of project developers.

3.3.2 Test Size Metrics

We extract the source lines of code (SLOC) of the performance tests and functional tests.

- *Source Lines of code (SLOC):* We run the command line tool `cloc`[5] on the identified (performance) test files and we `grep` the output for the Java SLOC, i.e., lines of code without comments, in the test files.
- *Number of commits:* We calculate the sum of the number of commits per developer as described above to get the number of commits that are made to the (performance) tests.

4. PERFORMANCE TESTING PRACTICES IN OPEN SOURCE SOFTWARE

We report on the results of our exploratory study on performance testing in Java-based OS software from five perspectives: (1) the developers that are involved in performance testing, (2) the extend of performance testing, including the size of the performance test suite, (3) the organization of the performance test suite, (4) the types of performance tests and (5) the tooling and frameworks used. In this section, we discuss the motivation for and the results of studying each of these five perspectives.

4.1 The Involved Developers

Motivation: Large-scale industrial projects may be able to commit one or more dedicated engineers to performance testing, who are experts in software performance engineering [20] (SPE). Contrary, even for many well-known OS projects, such as the Apache web server [19], the team of core developers who contribute the most to the project is small. Hence, it seems unlikely that there are developers who specialize in performance in most OS projects.

Approach: We conduct a quantitative study on the developers of the 111 studied OS projects using the developer metrics that are described in Section 3.3.1 to investigate whether OS projects have developers who specialize in performance.

We use the Wilcoxon signed-rank test to compare distributions of observations. The Wilcoxon signed-ranked test is a non-parametrical statistical test of which the null hypothesis is that two input distributions are identical. If the p-value of the Wilcoxon test is smaller than 0.05, we conclude that the distributions are significantly different.

In addition, we calculate Cliff's delta d [17] effect size to quantify the difference in the distributions of observations. A large value for d implies that the distributions differ at a large scale. We use the following threshold for interpreting d, as suggested by Romano et al. [21]:

$$\text{Effect size} = \begin{cases} \mathit{negligible}, & \text{if } |d| \leq 0.147. \\ \mathit{small}, & \text{if } 0.147 < |d| \leq 0.33. \\ \mathit{medium}, & \text{if } 0.33 < |d| \leq 0.474. \\ \mathit{large}, & \text{if } 0.474 < |d| \leq 1. \end{cases}$$

[5] http://cloc.sourceforge.net/

Table 1: Developers of the `SoftInstigate/restheart` project.

Performance test developers	
Name	**# of Commits**
Andrea di Cesare	27
Maurizio Turatti	9
gokrokvertskhov	1

Test developers	
Name	**# of Commits**
Andrea di Cesare	113
Maurizio Turatti	49
Stephan Windmuller	1
gokrokvertskhovh	1

Project developers	
Name	**# of Commits**
Andrea di Cesare	743
Maurizio Turatti	261
Michal Suchecki	4
Ayman Abdel Ghany	2
Srdjan Grubor	2
Bence Erls	1
Blake Mitchell	1
Stephan Windmuller	1
The Gitter Badger	1
gokrokvertskhov	1

Results: **Performance test developers are usually the core developers of the project.** Figure 3 shows the proportion of (performance) test developers who are in the top-n list of developers based on the number of commits. Figure 3 shows that in 50% of the studied projects, all performance test developers are core developers of the project. Figure 3 shows that the median proportion of test developers who are core developers is slightly lower. However, the Wilcoxon signed-rank test shows that the two distributions are not significantly different.

Performance tests are created and maintained by one or two developers in most of the studied projects. Figure 4 shows the number of performance test developers, test developers and project developers in each of the studied projects. 50% of the studied projects have 2 or less performance test developers, while the median number of developers in a project in our data set is 9. The projects with the highest number of performance test developers are `aseldawy/spatialhadoop` (13 performance test developers), `alibaba/druid` and `franzinc/agraph-java-client` (7 each), `apache/commons-ognl` (6), and finally `jasonrutherglen/HBASE-SEARCH` (5).

In 53 of 111 studied projects (48%) the performance tests are created and maintained by a single developer. This is particularly interesting for 13 of these 53 projects, which have 10 or more developers in total. In 43 of the 53 projects, the performance tests are created by the core developer of the project.

1 of 6 project developers work on the performance tests at least once in 50% of the studied projects. Figure 5 shows the proportion of performance test developers compared to the total number of developers and test developers in the project. We observe that test developers are not necessarily also responsible for performance testing. In 50% of the studied projects, only 44% of the test developers worked on the performance tests as well. The Wilcoxon signed-rank test shows that the distributions in Figure 5 are significantly different with a large effect size.

In most studied OS projects, performance tests are created and maintained by a single person or a small group of core developers.

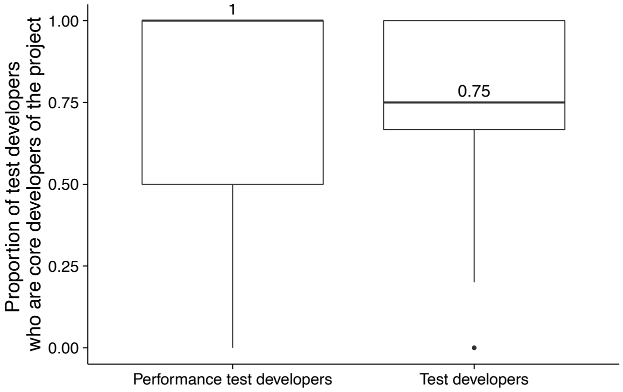

Figure 3: The proportion of (performance) test developers per project who are core developers. The number displayed within/above the boxplot is the median.

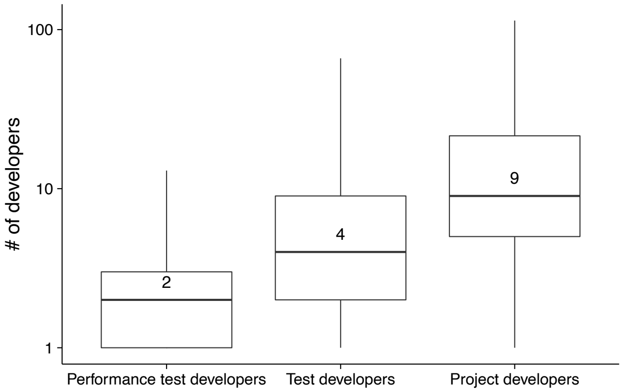

Figure 4: The number of developers per project. The number displayed within the boxplot is the median. Note that the axis is in logarithmic scale.

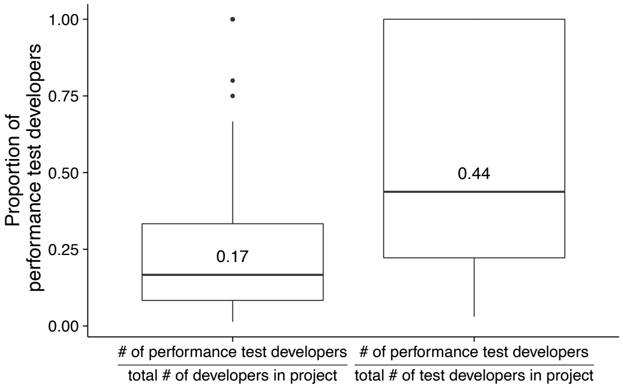

Figure 5: The proportion of performance test developers per project. The number displayed within the boxplot is the median.

4.2 The Extend of Performance Testing

Motivation: The size of the performance test suite of a project is an indication of the effort that the project dedicates to performance testing. While there exist no guidelines for the ideal size of a performance test suite, intuitively, we expect that projects that care about their performance have a larger performance test suite than other projects.

Approach: We conduct a quantitative study on the source lines of code (SLOC) of the performance tests and functional tests in the studied 111 OS projects using the SLOC metric that is described in Section 3.3.2.

Note that we decide against using the cyclomatic complexity as an indication of development effort. Over the last years, the contribution of complexity metrics has been under discussion in software engineering research [11, 27]. Therefore, we decide to use SLOC as an indication for the effort that a project dedicates to performance testing and additionally, we conduct a qualitative study on the contents of the performance tests to study their complexity in Section 4.4.

Results: **Performance tests are small in most projects.** Figure 6 shows the SLOC for the performance tests and functional tests. The two outliers in the boxplot are the ↗ `DeuceSTM/DeuceSTM` and ↗ `mediator/HadoopUSC` projects, which have 9460 and 6802 SLOC in their performance tests. The ↗ `DeuceSTM/DeuceSTM` description on GitHub[6] mentions that the included performance tests are known benchmarks, and not specific to this project.

There is no significant difference between the performance test suite size of PS and non-PS projects. Figure 6 shows the SLOC for the (performance) tests in 26 PS projects and 85 non-PS projects. The Wilcoxon rank-sum test shows that the number of SLOC in the performance tests in PS and non-PS projects are not significantly different. Our data does not show evidence that projects that claim superior performance take extra measures in their performance testing suites to justify these claims.

Performance tests form a small portion of the functional tests in most projects. Figure 6 shows that the median SLOC of performance tests is 246, which is a small part of the median SLOC used for functional tests (3980). We calculate that the median percentage of performance test SLOC is only around 8%.

16 of 111 studied projects (14%) have a single performance test commit. Figure 7 shows the number of performance and functional test commits for each project. In 16 projects, the performance tests are committed once and never maintained after, which suggests that the performance tests do not evolve together with the code in these projects. The median number of commits for the performance tests is 7, which indicates that many projects do not actively maintain performance tests. The outlier in our study is ↗ `h2oai/h2o-2` with 242 performance test commits, which is 18% of the total commits of this project at the time of study.

> *In most studied OS projects, the performance test suite is small and does not change often. While the mean SLOC of performance tests is larger in projects that claim high performance (PS), these results are not statisically significant.*

4.3 The Organization of Performance Tests

Motivation: By construction, all 111 projects have at least one performance test in place. However, we observe during our quantitative study on the developers involved and the size of performance tests that the extent and rigor of performance testing varies immensely between projects. To investigate whether OS software developers follow specific guidelines for organizing their performance tests, for example, as is the case with unit tests for functional testing, we study the organization of performance tests in OS projects.

Approach: We conduct a qualitative, exploratory study on the organization of the performance tests of the 111 studied OS projects. The first author of this paper studies the performance tests with regards to their file structure and infrastructure. The results are independently verified by the second author. Both authors have more than five years of experience in doing research on software performance. The exploratory methodology that is used to conduct the qualitative study is as follows:

1. Scan the performance tests of a project for remarkable observations, or for characteristics that are observed in other projects.
2. For each observation, count the number of projects for which this observation holds.

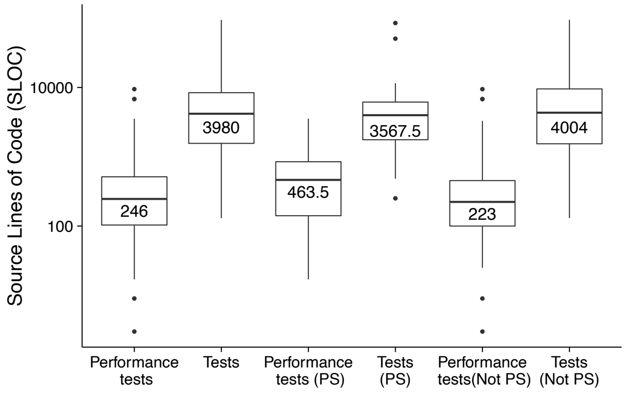

Figure 6: The number of (performance) test source lines of code (SLOC) per project. PS stands for performance-sensitive. The number displayed within the boxplot is the median. Note that the axis is in logarithmic scale.

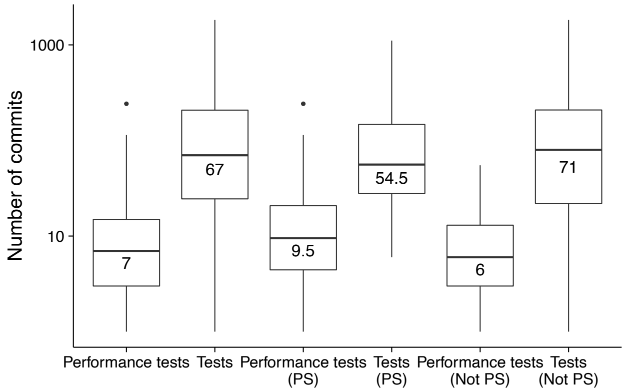

Figure 7: The number of (performance) test commits per project. PS stands for performance-sensitive. The number displayed within the boxplot is the median. Note that the axis is in logarithmic scale.

[6] https://github.com/DeuceSTM/DeuceSTM

3. Continue with the next project until all projects are studied.

Results: **The Java open source community has not yet converged against a common understanding of how to conduct performance tests.** While there are projects that appear to have a well-defined strategy for conducting performance tests, most projects in our study lack an evident performance testing vision, instead selecting what to write performance tests for and how to write those tests on a case-by-base basis. An example is `crosswire/jsword`, which has a single performance test for an isolated implementation issue mixed with the project's functional test suite, and two more end-to-end benchmarks that are implemented as stand-alone Java programs.

Generally, the vast majority of projects in our study are not treating performance testing as a matter of similar importance to functional testing, which is confirmed by the results of our quantitative study in Section 4.1 and 4.2. While we observe that 103 projects in our study include unit testing, usually following rather standardized best practices, the performance testing approach of the same projects often appears less extensive and less standardized.

58 of 111 (52%) studied projects scatter performance tests throughout their functional test suite. Another problem that projects seem to struggle with is how to organize and classify performance tests. 58 of the studied 111 (52%) projects, e.g., `alibaba/simpleel`, mix performance tests freely with their functional test suite (i.e., performance tests are in the same package, or even the same test file, as functional tests). 48 of 111 (43%) projects have separate test packages or categories that are related to performance, load, or stress testing. Finally, in 6 projects, performance tests are implemented as usage examples, and are delivered along with other example applications. This is the case for instance in the `apache/commons-math` project.

10 of 111 (9%) studied projects use code comments to communicate how a performance test should be executed. We also observe that some developers struggle with how to communicate to other project stakeholders how to run performance tests, what good or healthy results for specific tests are, and what environment these tests have been run in previously. 10 projects, including the `nbronson/snaptree` project, use code comments to communicate run configurations, as illustrated in Listing 1.

```
/*
 * This is not a regression test, but a micro-benchmark.
 *
 * I have run this as follows:
 *
 * repeat 5 for f in -client -server;
 * do mergeBench dolphin . jr -dsa\
 *    -da f RangeCheckMicroBenchmark.java;
 * done
 */
public class RangeCheckMicroBenchmark {
  ...
}
```

Listing 1: `nbronson/snaptree/../RangeCheckMicroBenchmark.java`

In 4 projects, we even observe that they use code comments to communicate previous benchmarking results. For instance, performance tests in `maxcom/lorsource` list a selection of previous run results in the footer of the source code files, as shown in Listing 2.

```
public class ImageUtilBenchTest {
  ...
}

/**
 * #1
 Running ru.org.linux.util.image.ImageInfoBenchTest
 Tests run: 1, Failures: 0, Errors: 0, Skipped: 0,
   Time elapsed: 0.228 sec
 Running ru.org.linux.util.image.ImageUtilBenchTest
 Tests run: 1, Failures: 0, Errors: 0, Skipped: 0,
   Time elapsed: 0.637 sec
 * #2
 Running ru.org.linux.util.image.ImageInfoBenchTest
 Tests run: 2, Failures: 0, Errors: 0, Skipped: 0,
   Time elapsed: 0.374 sec
 Running ru.org.linux.util.image.ImageUtilBenchTest
 Tests run: 2, Failures: 0, Errors: 0, Skipped: 0,
   Time elapsed: 1.152 sec
 * #3
 */
```

Listing 2: `maxcom/lorsource/../ImageUtilBenchTest.java`

5 of the 111 (5%) studied projects include empty stubs for performance testing. These projects, including the `graphaware/neo4j-reco` project, apparently planned to write performance tests, but never actually finished implementing them, leaving stubs in the project source code, as in Listing 3.

```
package com.graphaware.reco.perf;

public class EnginePerfTest {
}
```

Listing 3: `graphaware/neo4j-reco/../EnginePerfTest.java`

> *There appears to be no common understanding of how to conduct performance tests in Java OS software. Developers have no standardized way of organizing, executing, and reporting on the results of performance tests.*

4.4 Types of Performance Tests

Motivation: The types of performance tests in a project represent different mind sets and goals that developers have when writing performance tests. Hence, by studying the types of performance tests that a project uses, we get an insight into the developer's mind regarding performance testing.

Approach: We conduct a qualitative study on the types of performance tests used in each of the studied project following the same approach as described in Section 4.3. We extract five types of performance tests, which are discussed below. Note that none of the types represent "optimal performance testing" per se. Instead, all types exhibit their own advantages and disadvantages, upon which we comment in our discussion.

Results: **Type 1 – Performance Smoke Tests.** We use the term *performance smoke tests* for tests that are written with a high level of abstraction, and typically measure the end-to-end execution time of the most important end-user features for a limited number of example workloads. This type of test was the most common performance testing approach that we observed in our study, and was used by 55 of 111 (50%) projects. These projects typically implement a rather limited number of performance smoke tests, most commonly one or two. For example, `lbehnke/hierarchical-clustering-java` implements a single performance test case with a randomized clustering workload, as in Listing 4.

```
public class ClusterPerfTest {
    ...
  private Long timeN(int n) {
    Long t0 = System.currentTimeMillis();
    Cluster cluster = randomCluster(n);
    return System.currentTimeMillis() - t0;
  }

  @Test
  public void testn() throws Exception {
    for (int n = 2; n < 513; n = n * 2) {
      Long t = timeN(n);
      System.out.println("" + n + "\t" + t);
     }
  }
    ...
}
```

Listing 4: `lbehnke/hierarchical-clustering-java/../ClusterPerfTest.java`

A similar example is `kennycason/kumo`, a library to produce word clouds, which implements just two performance tests, one per major rendering strategy implemented in the library. These smoke tests allow developers to quickly identify large, end-user visible performance regressions. Another advantage of performance smoke tests is that they require little maintenance, as the end-user facing interfaces that these tests are written against change rarely. This is consistent with our quantitative results discussed in Section 4.2, where we have shown that many projects rarely or never change their performance tests. However, performance smoke tests provide little support for detailed debugging and metering of an application. Further, given that these high-level tests can be expected to exhibit substantial natural variability [16], small regressions cannot be detected with statistical confidence.

Type 2 – Microbenchmarks. 36 of 111 (32%) studied projects use performance testing that is more in line with unit testing, and strive to measure performance on a detailed level for smaller units of program code, i.e., using *microbenchmarks*. The microbenchmarking approach naturally requires more extensive performance test suites. An example of the microbenchmarking approach is the `TridentSDK/Trident` project, as shown in Listing 5.

In this project, performance testing is implemented in around 2500 SLOC using `JMH` and encompasses 49 separate performance tests, which are each configured with more than 10 different workloads. Unlike performance smoke tests, microbenchmarks allow developers to identify less severe performance regressions, before they have a noticeable impact on end-users. However, because microbenchmark suites are often larger than performance smoke tests, they require more time to develop and maintain, and typically take longer to execute.

Type 3 – One-Shot Performance Tests. 17 (15%) studied projects use what we dub *one-shot performance tests*. These tests usually benchmark very detailed performance issues, and appear to have been written primarily to support development-time decision making or to debug a specific issue. Such tests often compare different implementation alternatives or multiple external libraries with similar functionality, and appear to not be intended to be part of the regression test suite of the project. Consequently, such tests are often either implemented as stand-alone tests with their own main-method, or, if they are implemented as unit tests, set to `@Ignore`. An example snippet from the `jenkinsci/remoting` project that compares different regular expression pattern matching strategies is shown in Listing 6.

```
@State(Scope.Benchmark)
public class LatchTest {
  private static final HeldValueLatch<HeldValueLatch<?>> LATCH
          = HeldValueLatch.create();
  @Param({ "1", "2", "4", "8", "16", "32", "64",
          "128", "256", "512", "1024" })
  private int cpuTokens;

    ...

  @Benchmark
  public void down() {
    Blackhole.consumeCPU(cpuTokens);
    LATCH.countDown(LATCH);
  }

  @Benchmark
  public void wait(Blackhole blackhole) {
    Blackhole.consumeCPU(cpuTokens);
    try {
      blackhole.consume(LATCH.await());
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
}
```

Listing 5: `TridentSDK/Trident/../LatchTest.java`

```
@Ignore("This is not a test just a benchmark"+
  "and is here for ease of running")
public class RegExpBenchmark {

  final Pattern p1 =
    Pattern.compile("^org\\.codehaus\\.groovy\\.runtime\\..*");
  final Pattern p2 =
    Pattern.compile(
      "^org\\.apache\\.commons\\.collections\\.functors\\..*");
  final Pattern p3 =
    Pattern.compile("^.*org\\.apache\\.xalan\\..*");

    ...
}
```

Listing 6: `jenkinsci/remoting/../RegExpBenchmark.java`

Many examples of one-shot performance tests in our study appear to be source code artifacts of previous debugging sessions or discussions, rather than tests that provide continuing value to the project.

Type 4 – Performance Assertions. A fourth variation of performance test that is used by 6 (5%) projects in our study is the performance assertion. These tests are the result of developers striving to align performance testing with their functional test suite. Consequently, such tests are exclusively written in conjunction with unit testing frameworks such as `JUnit`, and often mixed with functional tests. An example of this approach comes from `anthonyu/KeptCollections` in Listing 7. This example compares the performance of two bytecode deserializers, and asserts that one implementation is at least 4 times as fast as the alternative. The same test file also contains similar tests for converters of other data types, with assertions that are different but seem similarly arbitrary

Presumably, the appeal of such performance assertions is that they integrate naturally with the unit testing frame-

```
@Test
public void testSerDeserIntPerf() throws Exception {
  long startNanos = System.nanoTime();
  for (int i = 0; i < 10000; i++)
    Assert.assertEquals(i, Transformer.bytesToObject(
      Transformer.objectToBytes(i, int.class), int.class));
  final long elapsed1 = System.nanoTime() - startNanos;

  Transformer.STRINGIFIABLE_PRIMITIVES.remove(int.class);

  startNanos = System.nanoTime();
  for (int i = 0; i < 10000; i++)
    Assert.assertEquals(i, Transformer.bytesToObject(
      Transformer.objectToBytes(i, int.class), int.class));
  final long elapsed2 = System.nanoTime() - startNanos;
  Assert.assertTrue(elapsed2 > elapsed1);
  Assert.assertTrue(elapsed2 > 4 * elapsed1);
}
```

Listing 7: anthonyu/KeptCollectionsTransformerTest.java

works that most OS projects already use (and with which developers are comfortable). Unfortunately, performance assertions are also relatively inflexible, require constant maintenance to ensure assertions remain useful, and are in danger of producing volatile failures due to external factors. Further, parameterizing performance assertions (e.g., deciding that the "right" speedup factor in the above example is 4) is fickle, as defining assertions conservatively means that smaller regressions can easily slip through, while aggressive assertions lead to many arbitrary test failures.

Type 5 – Implicit Performance Tests. 5 (5%) projects in our study choose to not have dedicated performance testing at all, but instead provide execution-time metrics as part of primarily functional tests. Such metrics are often as simple as writing performance data to `System.out` within a subset or all unit tests. This low-friction approach to performance testing has the advantage of exceedingly low additional development effort. However, systematically using the resulting data to track the performance of a project over multiple builds or versions is difficult. Further, functional tests are not necessarily optimized for performance measurement (e.g., they often do not use representative workloads, but rather workloads that implement the edge cases that are most important for functional testing). Hence, the expressiveness of such implicit performance tests is arguably low as compared to well-written dedicated performance tests.

Most studied OS projects use one or two performance smoke tests, to test the performance of the main functionality of the project, or a suite of microbenchmarks, to test the performance of smaller parts of the project.

4.5 Tooling and Frameworks

Motivation: There exist several widely-spread tools and frameworks, such as `JUnit`, for functional testing. In this section, we investigate whether there are performance testing tools that are similarly dominating in OS projects.

Approach: We conduct a qualitative study using the approach described in Section 4.3. We extract three general approaches that Java-based OS software uses for performance testing.

Results: **Approach 1 – Performance Unit Testing.** 57 (51%) projects in our study use their unit testing framework, typically `JUnit`, to write and run performance tests. These tests are unusual in that they typically do not actually "assert" anything (except in the case of performance assertions, as discussed above). Instead, such performance-related unit tests often just run an example workload and print metrics, either to `System.out` or to a file. An example from `fasseg/exp4j`, a simple Math calculator, is shown in Listing 8.

```
public class PerformanceTest {

  private static final long BENCH_TIME = 2l;
  private static final String EXPRESSION =
    "log(x) - y * (sqrt(x^cos(y)))";

  @Test
  public void testBenches() throws Exception {
    StringBuffer sb = new StringBuffer();
      ...
    int math = benchJavaMath();
    double mathRate = (double) math / (double) BENCH_TIME;
    fmt.format(
      "| %-22s | %25.2f | %22.2f %% |%n",
      "Java Math", mathRate, 100f
    );
    System.out.print(sb.toString());
      ...
    }
}
```

Listing 8: `fasseg/exp4j/../PerformanceTest.java`

The main advantage of using a unit testing framework to run performance tests is that this approach requires little to no changes to the project build configuration, including configuration of the CI server. Furthermore, developers do not need to learn additional tools such as `JMH`.

However, unlike dedicated frameworks, unit testing frameworks are not optimized for performance testing. Hence, the intricacies of benchmarking modern virtual machine code [6] (e.g., just-in-time compilation and garbage collection) can easily influence performance testing results. Another problem is that performance tests often take substantial time to execute. As a result, performance tests slow down the build considerably if these tests are part of the regular unit test suite of the project. Solutions to this issue include putting performance unit tests into a different test category that is excluded from standard builds, or setting performance tests to `@Ignore` by default and only running them on-demand.

Given the apparent popularity of performance unit testing in practice, it is unsurprising that some performance testing frameworks actually extend `JUnit` to ease integration. One example of such a tool is the `JUnitBenchmarks` framework[7]. However, the authors have by now discontinued the project and suggest using `JMH` instead.

Approach 2 – Stand-alone Performance Testing. In 55 (50%) projects in our study, performance tests are written as stand-alone Java programs. These projects eschew using any support framework for writing and executing tests. Instead, benchmark workloads, performance tests, and data reporting are implemented from scratch and customized for the project. These customizations can take the form of a large number of programs intended to be run independently (i.e., each with their own main method), simple bash scripts that launch a benchmark run, or sophisticated testing frameworks that appear to have required non-trivial development effort. An example is `rabbitmq/rabbitmq-java-client`, which built a dedicated command-line tool for launching performance tests, implementing various scenar-

[7] https://labs.carrotsearch.com/junit-benchmarks.html

ios and workloads, and supporting more than 25 command line options.

Writing performance tests as stand-alone Java programs offers the advantage of great flexibility, and naturally avoids "polluting" the project's unit tests. However, stand-alone performance tests need specific tooling or more complex setup should they be integrated into the project build, e.g., on the CI server. Furthermore, building and maintaining a sophisticated custom performance testing framework on a per-project basis arguably leads to considerable additional development effort.

It should be noted that many developers building such stand-alone performance tests seem to be at least vaguely aware of the problems of naïvely benchmarking Java code. Attempted workarounds that these developers employ include excessively invoking `System.gc()`, or finding workloads using trial-and-error that avoid just-in-time compilation for their specific system and used JVM. Furthermore, some developers use code comments or strings of test data to indicate doubt about the expressiveness of their performance tests. The example in Listing 9 shows this from the `dustin/java-memcached-client` project.

```
String object =
  "This is a test of an object blah blah es, "
  + "serialization does not seem to slow things down so much.  "
  + "The gzip compression is horrible horrible performance, "
  + "so we only use it for very large objects.  "
  + "I have not done any heavy benchmarking recently";
```

Listing 9:
`dustin/java-memcached-client/../MemcachedThreadBench.java`

Approach 3 – Dedicated Performance Testing Frameworks. Finally, only 18 (16%) projects in the study use dedicated performance testing frameworks. More detailedly, `JMH` is used by 10 projects, and Google `Caliper` by 4. An additional 4 projects are using lesser-known or more specialized frameworks, such as `JUnitBenchmarks`, `Contiperf`, or `Faban`. However, not all of these projects use dedicated performance testing frameworks for all their performance tests. Some projects, for instance `apache/commons-csv`, have some newer performance tests written in `JMH` in addition to their `JUnit`-based legacy tests.

An interesting question is why, even among projects that are per se interested in performance testing, so few elect to use a dedicated performance testing framework (even going through the pain of manually building a custom performance testing framework from scratch instead, as discussed above). One potential explanation is that performance testing frameworks may not be widely-known among developers. Another explanation is that dedicated frameworks may be perceived as too difficult to set up or to write tests with, or that the benefits of a dedicated framework do no outweigh the disadvantages of writing tests in `JUnit`. Finally, a possible explanation is that dedicated frameworks do not integrate easily with standard build systems, such as Jenkins. More research is needed to address the question of why OS software developers do not use a dedicated performance testing framework.

> *Most studied OS projects use either JUnit or simple standalone programs for performance testing in lieu of a dedicated performance testing framework.*

5. IMPLICATIONS

In this section, we give an overview of the lessons that we learned from our study, and we provide an interpretation of what our results imply for practitioners, OS developers, and academics who are working on topics that are related to performance testing.

There is a lack of a "killer app" for performance testing. First and foremost, a recurring observation in our study is that performance testing in Java-based OS software is neither as extensive nor as standardized as it is the case for functional testing. While the lack of standardization was already observed by Weyuker et al. in 2000 [25], our study results indicate that even at the time of writing there is still a long way to go before performance engineering truly becomes an integral part of the software development lifecycle for OS software. We speculate that one reason for this evident lack of standardization is that there currently is no "killer app" to conduct performance tests with in an easy, yet powerful, way, similar to how `JUnit` has standardised unit testing for Java.

Writing performance tests is not a popular task in OS projects. In 48% of the studied projects, performance tests are written by a single developer, who usually is a core developer of the project. In addition, performance tests are often written once and rarely or never updated after. These observations suggest that performance testing is a daunting and unpopular task in open source projects that does not attract many external contributors. Potential reasons for the small number of external contributors may include a perceived difficulty of writing performance tests or a lack of awareness in OS projects that performance tests are a potential contribution to the project. However, our data does not allow us to provide an answer to this question, requiring further research.

OS developers want support for quick-and-dirty performance testing. In performance testing research, emphasis is typically put on providing the most accurate and statistically rigorous performance results possible (e.g., controlling for as many confounding factors as possible). Contrarily, in our study we observe that developers are often willing to write "quick-and-dirty" performance tests, trading off accuracy of measurement for lower implementation effort. For instance, many projects are either not aware of or knowingly ignore the influence of the Java virtual machine on their performance tests, or report only arithmetic mean values, ignoring standard deviations or outliers. The apparent mismatch between what OS developers want and what performance testing research is delivering may be an explanation for the low adoption of existing performance testing tools. We feel that there is unexplored potential for future research on performance testing approaches that particularly focus on being low-friction for developers, for instance, through easy-to-use and non-parameterized methods, reusing existing code (e.g., existing unit tests) for performance testing, or through automated performance test generation.

Performance testing is multi-faceted. One potential reason why no universal performance testing framework has emerged yet is that performance testing is inherently multifaceted. We identify five different types of performance tests, which are usually conducted with vastly different goals in mind, consequently leading to different implementation and tool selection choices. However, note that this is not dif-

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2496v2 | CC BY 4.0 Open Access | rec: 6 Oct 2016, publ: 6 Oct 2016

ferent to functional testing, where the difference between unit testing, integration testing, or user acceptance testing is well-understood in research and practice. We hope that our study will serve as a first step towards a similar understanding of performance testing, ultimately leading to more targeted tool support.

Integration into standard CI frameworks is key. 51% of the projects in our study are piggy-backing on their unit testing frameworks to also conduct performance testing, despite not actually using any assertions. The usage of unit testing frameworks for performance testing indicates that a straight-forward integration into standard build and CI tooling, as provided by unit testing frameworks, is a key real-world feature which current industrial performance testing tools as well as research approaches do not sufficiently value. Recent work by Horký et al. [12] on declarative performance testing that integrates well with unit testing is a promising direction for addressing this integration challenge.

6. THREATS TO VALIDITY

In this section, we discuss the most relevant threats to the validity of our study.

External Validity. One of the external threats to our results is generalization. While the goal of our exploratory study is not to be exhaustive, we aimed at studying a wide range of representative Java-based OS projects that conduct performance testing. However, it is possible that our results do not extend to projects that are written in different languages or industrial projects. In order to address this threat, additional research on industrial projects and projects that are written in different languages is necessary.

Our project selection may be biased by assuming that the test code of a project is stored inside the `src/test` directory, following the conventions introduced by Apache `Maven`. While this assumption seems reasonable for Java-based OS projects, more research is needed for identifying an exhaustive list of projects that have performance tests.

Construct Validity. There are several threats to the construct validity of our study. First, we assume that the performance tests of a project are stored as part of the GitHub repository. We are aware that some OS projects have externalized their benchmarking and performance testing code to external projects. These projects are not considered by our research design.

Further, we use a manual identification process to identify the performance tests in a project, which may in some cases have led us to miss performance tests, especially if those are stored in a non-standard location. To mitigate this threat, the process was conducted by the two authors of the paper, who both have more than five years of experience in performance testing research.

In addition, in several projects, developers appear to be using multiple GitHub profiles. As it is difficult to decide whether these profiles are actually owned by the same developer, we treat them as owned by different developers. As a result, our observations may be slightly overestimating the number of developers that are engaged in performance testing.

Finally, we count only the SLOC in Java files. If a project uses a different language for its performance tests, the code is not included in our study. We observed that only one of the projects uses Scala for its performance tests.

7. CONCLUSION

In this paper, we have presented an exploratory study of performance testing in OS Java projects. We have extracted a data set of 111 projects from GitHub which conduct performance testing. We used a combination of quantitative and qualitative research methods to investigate the performance tests used by these projects. We find that that the current state of performance testing in Java-based OS projects is not overly reassuring. Most projects contain few performance tests, which are also hardly maintained. Projects do not appear to receive many source code contributions related to performance testing. Instead, performance tests are often written as a one-time activity by a core developer. Further, we note that projects approach performance testing from different angles – while most projects focus particularly on high-level performance smoke tests, others conduct more extensive microbenchmarking or even define a small number of performance assertions using `JUnit`. While performance testing frameworks exist, they are not used by most projects. Instead, projects often either build more or less sophisticated standalone test suites from ground up, or repurpose their unit testing framework for performance testing.

Our study results imply that developers are currently missing a “killer app” for performance testing, which would likely standarize how performance tests are conducted similar to how `JUnit` has standardized unit testing. An ubiquitious performance testing tool will need to support performance tests on different levels of abstraction (smoke tests versus detailed microbenchmarking), provide strong integration into existing build and CI tools, and support both, extensive testing with rigorous methods as well as quick-and-dirty tests that pair reasonable expressiveness with being fast to write and maintain even by developers who are not experts in software performance engineering.

The main limitation of our work is that we have focused exclusively on OS projects written in the Java programming language. Hence, it is unclear to what extend our results generalize to other communities. This question should be answered in a follow-up study. Further, more research will be needed to investigate whether existing performance testing tools such as JMH are indeed missing important functionality as indicated above, or simply not well-known enough in the OS community.

8. ACKNOWLEDGEMENTS

The research leading to these results has received funding from the Swiss National Science Foundation (SNF) under project name “MINCA”.

9. REFERENCES

[1] M. Aberdour. Achieving quality in open-source software. *IEEE Software*, 24(1):58–64, Jan 2007.

[2] J. P. S. Alcocer and A. Bergel. Tracking down performance variation against source code evolution. In *Proceedings of the 11th Symposium on Dynamic Languages*, DLS 2015, pages 129–139, New York, NY, USA, 2015. ACM.

[3] A. Avritzer, J. Kondek, D. Liu, and E. J. Weyuker. Software performance testing based on workload characterization. In *Proceedings of the 3rd International Workshop on Software and Performance (WOSP)*, pages 17–24. ACM, 2002.

[4] S. Baltes, O. Moseler, F. Beck, and S. Diehl. Navigate, understand, communicate: How developers locate performance bugs. In *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–10, Oct 2015.

[5] M. Beller, G. Gousios, A. Panichella, and A. Zaidman. When, how, and why developers (do not) test in their IDEs. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 179–190. ACM, 2015.

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

[7] J. Cito, P. Leitner, T. Fritz, and H. C. Gall. The making of cloud applications: An empirical study on software development for the cloud. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 393–403. ACM, 2015.

[8] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous java performance evaluation. In *Proceedings of the 22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications (OOPSLA)*, pages 57–76. ACM, 2007.

[9] X. Han and T. Yu. An empirical study on performance bugs for highly configurable software systems. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '16, pages 23:1–23:10, New York, NY, USA, 2016. ACM.

[10] C. Heger, J. Happe, and R. Farahbod. Automated Root Cause Isolation of Performance Regressions During Software Development. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 27–38. ACM, 2013.

[11] I. Herraiz and A. E. Hassan. Beyond lines of code: Do we need more complexity metrics? *Making software: what really works, and why we believe it*, pages 125–141, 2010.

[12] V. Horký, P. Libič, L. Marek, A. Steinhauser, and P. Tůma. Utilizing performance unit tests to increase performance awareness. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 289–300. ACM, 2015.

[13] P. Huang, X. Ma, D. Shen, and Y. Zhou. Performance regression testing target prioritization via performance risk analysis. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 60–71, New York, NY, USA, 2014. ACM.

[14] Z. M. Jiang and A. E. Hassan. A survey on load testing of large-scale software systems. *IEEE Transactions on Software Engineering (TSE)*, 41(11):1091–1118, 2015.

[15] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and detecting real-world performance bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 77–88. ACM, 2012.

[16] J. Larres, A. Potanin, and Y. Hirose. A study of performance variations in the mozilla firefox web browser. In *Proceedings of the Thirty-Sixth Australasian Computer Science Conference (ACSC)*, pages 3–12. Australian Computer Society, Inc., 2013.

[17] J. D. Long, D. Feng, and N. Cliff. Ordinal analysis of behavioral data. *Handbook of psychology*, 2003.

[18] Q. Luo, D. Poshyvanyk, and M. Grechanik. Mining performance regression inducing code changes in evolving software. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 25–36, New York, NY, USA, 2016. ACM.

[19] A. Mockus, R. T. Fielding, and J. Herbsleb. A case study of open source software development: The apache server. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE)*, pages 263–272. ACM, 2000.

[20] R. Pooley. Software engineering and performance: A roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 189–199, New York, NY, USA, 2000. ACM.

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

[22] J. P. Sandoval Alcocer, A. Bergel, and M. T. Valente. Learning from source code history to identify performance failures. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, ICPE '16, pages 37–48, New York, NY, USA, 2016. ACM.

[23] C. U. Smith and L. G. Williams. Software performance engineering: A case study including performance comparison with design alternatives. *IEEE Transactions on Software Engineering (TSE)*, 19(7):720–741, July 1993.

[24] M. D. Syer, Z. M. Jiang, M. Nagappan, A. E. Hassan, M. Nasser, and P. Flora. Continuous validation of load test suites. In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 259–270. ACM, 2014.

[25] 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 (TSE)*, 26(12):1147–1156, Dec. 2000.

[26] S. Zaman, B. Adams, and A. E. Hassan. A Qualitative Study on Performance Bugs. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 199–208. IEEE Press, 2012.

[27] F. Zhang, A. E. Hassan, S. McIntosh, and Y. Zou. The use of summation to aggregate software metrics hinders the performance of defect prediction models. *IEEE Transactions on Software Engineering*, (1):1–16, 2016.