Unit Testing Performance in Java Projects: Are We There Yet?

Petr Stefan[1], Vojtěch Horký[1], Lubomír Bulej[1,2], Petr Tůma[1]

[1] Department of Distributed and Dependable Systems
Faculty of Mathematics and Physics,
Charles University
Prague, Czech Republic
first.last@d3s.mff.cuni.cz

[2] Faculty of Informatics,
Università della Svizzera italiana
Lugano, Switzerland
bulejl@usi.ch

ABSTRACT

Although methods and tools for unit testing of performance exist for over a decade, anecdotal evidence suggests unit testing of performance is not nearly as common as unit testing of functionality. We examine this situation in a study of GitHub projects written in Java, looking for occurrences of performance evaluation code in common performance testing frameworks. We quantify the use of such frameworks, identifying the most relevant performance testing approaches, and describe how we adjust the design of our SPL performance testing framework to follow these conclusions.

Keywords

Performance Unit Testing; Open Source; Survey; JMH; SPL

1. INTRODUCTION

This paper is motivated by two observations, one related to performance testing and one to software testing activities in general:

– Software performance testing is widely recognized as an essential software quality assurance tool. Reports of major companies practicing systematic performance testing [11], as well as analyses of project failures associated with insufficient performance testing [4], make performance testing a common wisdom activity.

– Another common wisdom argument in software testing activities, associated especially with test driven development, is that the earlier a test is done, the cheaper it is to remove the discovered defects [24].

Combining the two points naturally leads to work on early performance testing, such as performance evaluation through architectural performance models [1]. Here, we focus on unit testing of performance, which is situated roughly between architectural performance modeling and system performance testing.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICPE'17, Apr. 22–26, 2017, L'Aquila, Italy

© 2017 ACM. ISBN 978-1-4503-4404-3/17/04...$15.00

DOI: http://dx.doi.org/10.1145/3030207.3030226

Much of the existing work on unit testing of performance is situated in the Java ecosystem. Java is a major software platform for performance sensitive applications, including server side containers such as Glassfish or Tomcat or distributed computing frameworks such as Flink or Hadoop, but also a software platform whose performance related behavior is often more complex than that of traditional languages like C.

The Java ecosystem offers multiple frameworks potentially suitable for unit testing of performance, including Caliper [19], ContiPerf [2], Japex [28], JMH [27], or JUnitPerf [7]. Some of these frameworks exist for over a decade, however, anecdotal evidence suggests that unit testing of performance is still not as established as unit testing of functionality or system performance testing. In this paper, we replace the anecdotal evidence with more rigorous data on the actual practice of unit testing of performance, and propose modifications to our performance testing framework, SPL [6], to reflect this data.

The main contributions of this paper are:

– We analyze 99019 open source software projects on GitHub, totaling nearly 3 TB of data, and provide both current and historical statistics on the use of performance testing frameworks in these projects.

– We collect and summarize positions on unit testing of performance from 111 open source software developers who use the JMH performance testing framework, and use these to supplement the GitHub analysis results.

– We identify those projects that implement performance tests potentially suitable for unit testing of performance, and provide statistics on the test time and measurement accuracy.

– We explain how we adjust the design of our SPL performance testing framework to reflect these results.

The paper is structured as follows. In Section 2, we point to the motivating context and introduce the performance testing frameworks whose use is analyzed in later sections. Section 3 formulates the research questions for our survey and discusses threats to internal validity. Section 4 presents and discusses measurements that answer the research questions. Section 5 focuses on the adjustments of our own performance testing framework. Finally, Section 6 concludes the paper.

2. RELATED WORK

Taken separately, both unit testing and performance testing are well established quality control activities. At the unit testing side, studies such as [12] and [25] report successful defect reduction in industrial settings. At the performance testing side, studies such as [35] provide an early summary of the issues, and papers such as [36] contain a broader overview of the software performance engineering challenges.

Moving beyond the established principles of unit testing, multiple studies summarize developer opinions on the testing practice and identify open research issues. Runeson [29] remarks that multiple companies report issues with the effort of testing automation, Engstrom and Runeson [10] repeat the same concern in the context of regression testing. Greiler et al. [14] examine unit testing in the context of component systems, pointing out both a strong developer preference for testing automation and a need for reasonable setup effort and test time. Our work contributes to the existing unit testing studies by focusing specifically on unit testing of performance – where, interestingly, some of the listed concerns are emphasized due to more demanding test automation and more expensive test execution.

A unit testing survey by Daka and Fraser [8] observes that some developer responses are not necessarily accurate. As an alternative to surveying developers, source code repositories are mined in an open source software testing study by Kochhar et al. [20]. Our work provides more detailed information by combining repository mining with surveying developers for supplementary information. Also, our focus on the Java ecosystem permits us to process the repository content much more accurately – for example we identify tests by searching for specific code constructs rather than for file name patterns.

A related line of research surveys the performance testing practices. Nistor et al. [26] look at the difference between performance bugs and functional bugs. The authors point out that compared to reasoning about functionality, developers have little support for reasoning about performance. From the testing perspective, better oracles for evaluating test conditions are called for.

Jit et al. [17] and Liu et al. [23] focus on characterizing real world performance bugs, as opposed to performance bugs discovered in possibly artificial testing conditions. Their work, one experimenting with Linux and one with Android, identifies typical features of existing performance bugs and applies this knowledge to look for new bugs. Again, the need for automated evaluation of measurement data is listed as one open issue.

Linares-Vásquez et al. [22] investigate the current practices of locating and fixing performance issues in mobile applications. Among other results, the conclusions of the study most related to testing show that 73 % of developers rely on manual testing and 51 % on user feedback to locate performance issues, expressing preference for observation based analysis rather than automation.

Overall, the conclusions of the existing studies reveal definite gap between testing of functionality and performance. In functional testing, automation at unit test level works reasonably well, even if the setup effort and test time are not necessarily trivial. In performance testing, manual approaches appear dominant and automation is asked for. Our study investigates this gap.

Combined together, unit testing and performance testing are also particularly close to the increasingly popular DevOps movement. Acceptance of DevOps principles can not only lead to increased performance awareness [13, 21, 34], but also provide direct software process benefits – for example a retroactive case study by Waller et al. [33] shows how integrating Kieker (performance monitoring tool) with Jenkins (continuous integration tool) can lead to earlier detection of performance regressions.

On the technical side, both unit testing and performance testing rely on established tool support. Among tools relevant to the platform context and testing scale of this work, we have:

- unit testing tools such as JUnit [18] or TestNG [31], which simplify unit test implementation by providing standard constructs for marking test methods, expressing test conditions and managing test fixtures, together with an environment that automatically executes the tests and reports the results, and
- performance testing tools such as JMH [27], which provide constructs for marking measured workload, defining measurement conditions and controlling potentially disruptive optimizations, together with an environment that automatically executes the measurements and collects the results.

For the purpose of this work, we are interested in tools that support unit testing of performance. Table 1 provides a list of the relevant frameworks. While common unit testing tools have no performance testing support, special purpose extensions exist. The table includes ContiPerf [2] and JUnitPerf [7] as two extensions of JUnit, both support specifying absolute limits on test execution time as the test condition. For performance testing tools, the table lists Caliper [19], Japex [28] and JMH [27], three frameworks that target microbenchmark implementation.

Somewhat separately, Table 1 also lists SPL [6]. SPL is our performance testing framework and formalism for specifying test conditions. Although SPL can be used in real open source projects [16], the framework is still subject to frequent

Framework	Metrics	Asserts	Output	Maintained
SPL	Time	Yes	CSV, charts[a]	2012 –
JMH	`perf`[b]	No	Text[c], JSON	2013 –
Caliper	Time	No	Text[c]	2008 –
JUnitPerf	Time[d]	Yes	Text[c]	2009 – 2010
ContiPerf	Time[d]	Yes	CSV, charts[a]	2010 – 2014
Japex	Time[d]	No	XML, charts[a]	2005 – 2011

Table 1: *Comparison summary of Java benchmarking frameworks. SPL is a research project on performance unit testing; JMH, Caliper and Japex are microbenchmarking frameworks; JUnitPerf and ContiPerf are performance related extensions of the JUnit testing framework.*

[a] Charts embedded in an HTML report.
[b] Linux perf counters and throughput available.
[c] For human reader, machine processing possible.
[d] Throughput also available.

large scale modifications and therefore not a part of our usage survey – instead, we use the survey to collect feedback that motivates modifications described in Section 5.

From the broader context of automated performance testing, we want to mention the performance monitoring frameworks that collect data through instrumentation, such as Kieker [32]. With performance monitoring, unit test execution time can be observed and possibly used for performance testing, however, such practice is complicated by the need for repeated test execution with stable test fixtures and other issues.

Also related is the issue of robust measurement environment. For performance testing to deliver meaningful results, the measurement environment must be sufficiently representative. In performance sensitive applications, unit test execution may benefit from environments such as DataMill [9] to guard against the accidental measurement bias due to platform configuration and to provide platform variability.

3. SURVEY DESIGN

To assess whether unit testing of performance is as well established as unit testing of functionality, we formulate multiple research questions with measurable answers. We aim for answers that can be measured by analyzing source code repositories – such repositories are available for many software projects and we can therefore achieve reasonable project coverage. We start with the key question:

Q1: How much is unit testing of performance used?

An answer to Q1 can be expressed as the share of software projects that use unit testing of performance. To recognize whether a software project uses unit testing of performance, we look for statements that import packages or declare annotations distinctive for particular performance testing frameworks. This approach can be reasonably automated, however, it requires careful discussion of threats to validity on two levels – at the implementation level, it is not necessarily true that recognizing packages or annotations equates to using a performance testing framework, and, at the process level, it is not necessarily true that using a performance testing framework equates to practicing the unit testing of performance.

At the implementation level, we detect usage patterns associated with all the performance testing frameworks surveyed in Section 2. In these frameworks, importing distinct packages or declaring distinct annotations are reliably established usage patterns. Table 2 lists the base packages and the test markers – we consider a reference to the base package anywhere in code as an indicator that the associated framework is used, and we count the uses of test markers to determine the number of tests. Our detection tool uses source code parser to properly resolve imports and distinguish ambiguous textual identifiers.

Although it is technically possible to avoid the listed usage patterns and still invoke particular framework features, such practice would be obscure. We therefore classify the threat of failing to detect a framework that is actually used as low.

Inversely, we can detect a framework that is present but not used. Performance testing frameworks implement packages and annotations that software projects are unlikely to use

Framework	Base package Test marker
Caliper	`com.google.caliper` `@Benchmark`
ContiPerf	`org.databene.contiperf` `@PerfTest`
Japex	`com.sun.japex` `JapexDriverBase`
JMH	`org.openjdk.jmh` `@Benchmark`, `@GenerateMicroBenchmark`
JUnitPerf	`com.clarkware.junitperf` `TimedTest`, `LoadTest`
JUnit	`org.junit` `@Test`
TestNG	`org.testng` `@Test`

Table 2: *Packages and annotations used to detect particular frameworks. Presence of any type from the base package indicates framework use. One use of any test marker counts as one test.*

for purposes other than performance testing, the threat is therefore most likely manifested with abandoned performance testing attempts. Because we execute all the recognized performance tests to answer some of the further research questions, we avoid this threat entirely.

A software project can also implement performance tests in a proprietary manner, not relying on any performance testing framework. This threat needs to be considered especially because our initial premise is that performance testing frameworks are not well established and proprietary test implementations are therefore likely. To identify such software projects, we rely on the assumption that a performance test must query the clock to determine performance. Any code that uses standard system interface to query the clock is identified as a potential performance test, further manual classification is used to determine the purpose for which the clock is queried.

At the process level, an answer to Q1 requires making a distinction between projects whose code contains performance tests and projects whose development process involves unit testing of performance. The existence of performance tests is a necessary but not sufficient condition for unit testing of performance, which also requires that the tests execute with unit granularity and that the test conditions are specified and evaluated.

To address the issue of granularity, we attempt to execute all recognized performance tests. Tests that fail to run due to missing external dependencies (network services to connect to, external data to read, etc.) are not considered unit tests (unit tests should mock or do without dependencies). Similarly, tests that run too long are not considered unit tests (unit tests should be small enough to execute during regular builds). The remaining performance tests, which execute without dependencies and within reasonable time, are considered candidates for unit testing.

Given only project code, it is not possible to reliably check whether the candidate tests specify and evaluate test conditions. Simple assertion statements, typical for unit testing of functionality, are not suitable for unit testing of

performance, where performance measurements can seldom be compared using classical test conditions [5, 6]. More complex evaluation approaches can escape detection when implemented in the build infrastructure rather than the project code [15, 3]. In the answer to Q1, we therefore include all the candidate tests, assuming they could be easily used as unit tests of performance if they are not already. Further research questions evaluate the accuracy of such tests.

To interpret the answer to Q1, we also need a baseline. We therefore measure the share of projects that implement unit tests of functionality, and compare that to the share of projects that implement unit tests of performance. The technical implementation is similar, but we detect the use of general testing frameworks rather than performance testing frameworks.

Q2: How much does unit testing of performance change with time?

Source code repositories permit examining not only the current state of a software project, but also the development history. By measuring the answers to Q1 at different points in time, we therefore provide an answer to Q2. Again, we use measurements for unit testing of functionality as a baseline to interpret the measurements for unit testing of performance.

Q3: What kind of software projects are measuring performance?

Unit testing of performance is likely to differ depending on the kind of software project involved. This has important repercussions – the general project domain may determine what the unit tests need in terms of environment, configuration, mock dependency injection and other features, while parameters such as code size and commit frequency determine requirements on the scalability of the testing infrastructure.

To provide an answer to Q3, we measure the project size in lines of code both now and at the time a performance test first appeared in the project. We also collect the average commit frequency across the project. Finally, we manually classify the general project domain.

Q4: How long does unit testing of performance take?

Unit tests of functionality typically take a fixed amount of time to arrive at the pass or fail decision. Due to the variability inherent to performance measurements, this is not the case with unit tests of performance – instead, the test can collect measurements continuously and the longer it runs, the higher confidence or sensitivity it provides. Furthermore, modern execution environments need some minimum execution time to reach stable performance, making short test runs less representative.

To answer Q4, we give the time needed to execute all the candidate performance tests. However, this time depends on factors such as measurement framework configuration options, which are often set in the build infrastructure rather than the project code. When the real options differ from the defaults, so will the time reported. We therefore also report the accuracy obtained after one hour of measurement, as a metric that is connected with the test time and reflects what results to expect when testing with one hour period (one hour was picked to resemble testing after each commit).

Q5: Does unit testing of performance reveal actual performance changes?

With unit tests in place, the obvious remaining question is, do their results reveal actual performance changes? To answer Q5, we simply look at the differences in measured performance at chosen points in project lifetime. Such differences can arise due to real change in project performance, but also due to changes in test code. To avoid mixing the two, we compare current performance with performance measured after last test code modification, thus excluding differences due to changes in test code.

Developer Survey

Repository mining readily provides aggregate statistics but does not explain the results. To avoid speculation when discussing the results of the formulated research questions, we have asked the developers who use the dominant performance testing framework to answer a short survey on the following topics:

- reasons for choosing a particular framework,
- degree of integration into the development process,
- degree of automation in processing the results,
- perceived usefulness of performance testing,
- perceived obstacles to performance testing.

We present selected results of the developer survey when discussing answers to the research questions. Exact wording of the survey questions and complete results are available at [30].

4. SURVEY RESULTS

To answer the formulated research questions, we analyze source code repositories from GitHub. GitHub is likely the most popular open source software hosting facility, claiming over 15 million users and 38 million projects[1], compared for example to SourceForge with 3.7 million users and 430 thousand projects[2] (these and other presented numbers on repository sizes and project counts were collected in the period of August to October 2016). Our use of an open source repository can naturally introduce bias, however, it is not likely that we would succeed in getting information on a similar number of closed source software projects and the associated development processes from the software industry.

GitHub provides a code search API with functions essential to our survey – in particular, we can list projects that meet particular language criteria. As a complication, the API is limited to returning at most 1000 entries matching a query, plus an information on the total number of matching entries. Also, answers to queries whose processing exceeds preset timeout are incomplete. To sidestep these limitations, we split each query that returns more than 1000 entries into multiple smaller queries whose answers together form the answer to the original query. This is done by adding interval filters on attributes such as repository size and fork count and recursively splitting the intervals until the queries are satisfied. We note this approach can introduce races because the individual queries observe GitHub at slightly different moments in time (the processing time for the central query that lists all surveyed projects was almost 7 hours), however,

[1] http://github.com/about
[2] http://sourceforge.net/about

we submit the individual queries in the order of growing repository sizes and growing fork counts, making it less likely that we miss a project (it would have to shrink in size or fork count just between the relevant queries).

We use the API to identify Java projects, getting approximately 2.4 million entries excluding forks. Because these contain a high number of obviously invalid projects (for example, we get approximately 314 thousand entries for projects whose repository size is below 16 kB and over 94 thousand of those are empty), we restrict our survey to projects that have been forked at least twice (a non zero fork count is a liberal filter for projects that receive some attention, established projects collect up to thousands of forks). There are approximately 100 thousand such projects, for our analysis we have successfully cloned and processed exactly 99019 repositories totaling almost 3 TB of data.

A1: How much is unit testing of performance used ...

Table 3 gives the share of projects whose source code uses one of the surveyed performance testing frameworks (Caliper, ContiPerf, Japex, JMH, JUnitPerf). As a baseline, the table also gives the share of projects whose source code uses either JUnit 4 or TestNG, two dominant unit testing frameworks.

The table indicates that unit testing of performance using the surveyed performance testing frameworks is extremely rare. The most used framework, JMH, is found on average less than three times in thousand projects, the other frameworks are used an order of magnitude less often. Because subsequent analysis steps further reduce this number, we will only consider projects that use JMH and refrain from drawing conclusions on projects with other frameworks (generalizing from what are basically singular use cases is not possible).

With JMH identified as the dominant framework, we have also sent the developer survey to the developers who use it – specifically, those developers who have updated any performance test in any commit of the 278 relevant projects. We have sent out a total of 483 invitations to fill in the survey and received 111 completed forms (78 with permission to publish results, 26 with permission to publish summary, 7 with no permission to publish).

The developer survey lists trust in results as the top reason for choosing JMH at 72 %. Other reasons include active maintenance at 60 % and good documentation at 40 %. Build system integration is rated comparatively low at 33 %. These results emphasize the difficulty of writing correct performance tests and suggest few frameworks are considered mature.

Framework	Repositories	Relative usage
Caliper	12	0.012 %
ContiPerf	17	0.017 %
Japex	52	0.053 %
JMH	278	0.281 %
JUnitPerf	11	0.011 %
JUnit 4	30871	31.177 %
TestNG	2053	2.073 %
Total	99019	100 %

Table 3: *Java test framework usage on GitHub.*

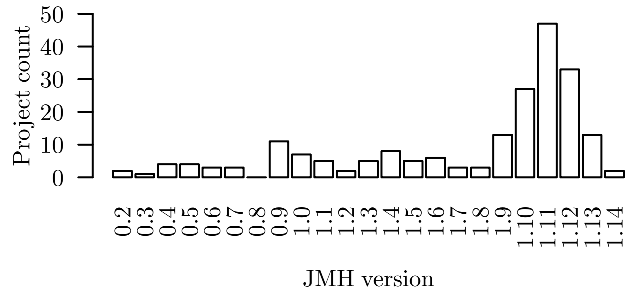

Figure 1: *JMH versions used by projects. Versions 1.11, 1.12, 1.13 and 1.14 were released on January, April, July and September 2016, respectively.*

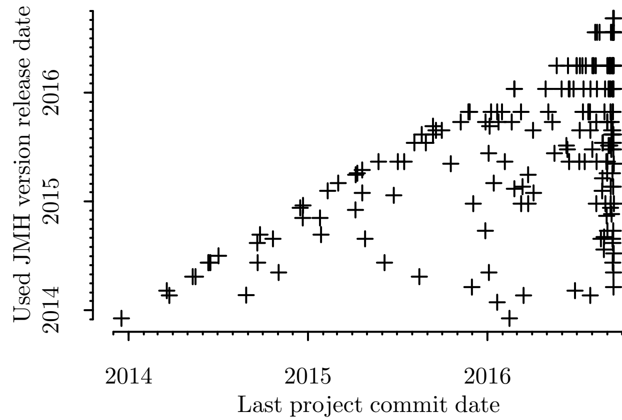

Figure 2: *JMH version adoption delay. Each project is represented by one point, projects near the diagonal use the most current JMH version, the further from the diagonal, the more obsolete the JMH version used.*

Subsequent analysis steps require building and executing all the recognized performance tests. For this, we use Fedora Linux 24 with OpenJDK 1.8.0 in the latest update, software dependencies resolved through build configuration, running on a dual Intel Xeon E5-2660 machine (20 MB cache, 2.2 GHz clock, 8 cores) with 48 GB RAM. For measurements, we disable hardware threads, frequency scaling and boosting, and constrain the execution to single NUMA node for both memory and processor allocation.

As a technical complication, JMH outputs only summary measurement information but for accurate statistical processing we need individual measurements. Before building the projects, we therefore created a modified JMH version and inserted it among project dependencies. For that, we need projects with standard dependency specification – among the projects that use JMH, these are 223 projects built with Maven and 52 projects built with Gradle. Figure 1 lists the JMH versions originally used in the projects, Figure 2 illustrates the JMH adoption speed by plotting the release date of the used JMH version against the last commit date for each project. We note that our modifications are included in standard JMH starting with version 1.14.

After inserting the modified JMH version, we attempt to build two versions of each project. One is the HEAD commit, one is the last commit which modified any recognized performance test. This yields 29 projects whose tests produce

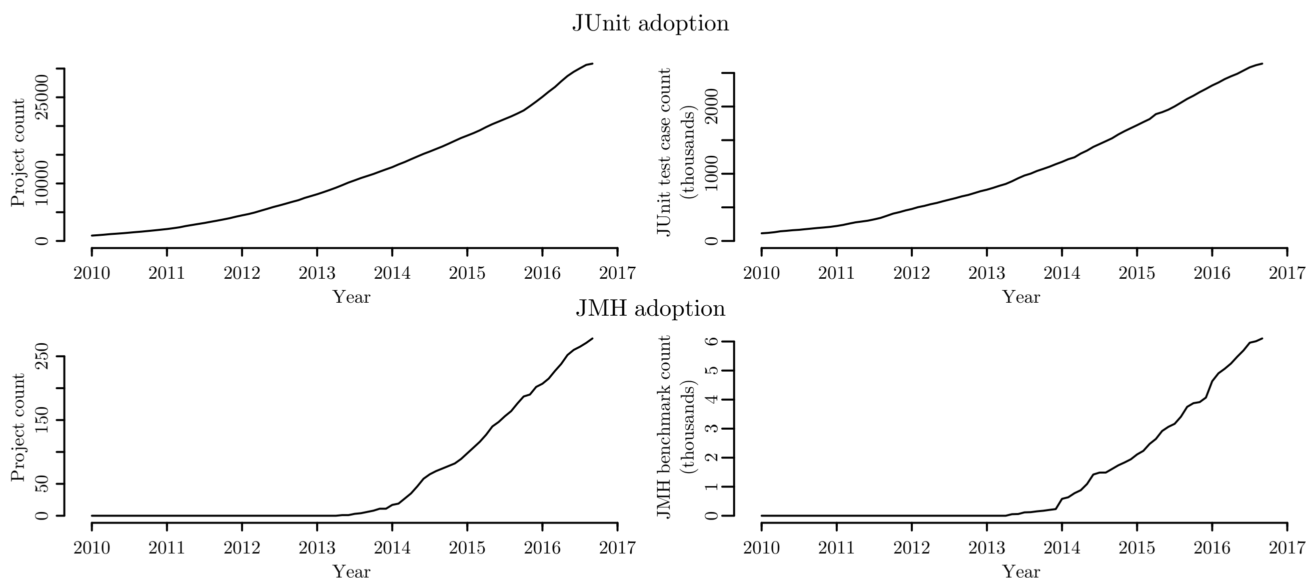

Figure 3: *JMH and JUnit adoption rate comparison. The plots show how the number of projects that use JUnit 4 or JMH (left) and the total number of test cases or benchmark methods in these projects (right) changes with time. The data was collected for each day in the plot range.*

measurements within 4 hours of execution, 9 projects whose tests execute but do not produce measurements within 4 hours, 9 projects whose tests crash without output, and 235 projects that do not build in the two chosen versions. In the following, the 29 projects whose tests produce measurements within 4 hours are referred to as the measured projects and the two versions as the measured versions.

To summarize the answer to Q1 where the use of performance testing frameworks is concerned: use of the surveyed performance testing frameworks is extremely rare, at most 0.37 % of the analyzed projects use any performance testing framework and at least 62 % of those projects whose tests readily build and execute produce measurements within 4 hours. Although the analysis suffers from high attrition rate, these numbers should represent reasonable upper and lower bounds.

To analyze projects that implement performance tests without any performance testing framework, we look for the use of three clock query functions available in Java – `System.nanoTime()`, `System.currentTimeMillis()` and `ThreadMXBean.get*Time()`. Because manual classification of all 99019 projects is not practical, we pick 1000 projects at random. Of those, 332 projects did query the clock, their classification is in Table 4.

The classification is based on manual exploration of the call site and the immediately surrounding code. To reduce classification error, the exploration was done independently by two researchers and keywords were assigned to each project. Afterwards, similar keywords were gradually merged into clusters as in hierarchical clustering. To provide some estimate of the classification error involved, we note that 70 % of the time there was overlap between keywords assigned by the two researchers to a project.

The classification contains several categories where unit test use is not likely – timeout handling in caches and sockets, calendar, scheduling, randomization. In some categories, unit

Usage	Count	99 % CI
Timeout handling	125	99 – 154
Logging of durations	133	107 – 163
Querying calendar time	122	97 – 151
Event scheduling, GUI	89	67 – 115
Randomization, unique naming	65	47 – 88
Proprietary benchmarking	34	21 – 52
Custom timer infrastructure	29	17 – 46

Table 4: *Usage of System.nanoTime and System.currentTimeMillis. Projects that span multiple categories are counted multiple times.*

test use is possible – information from logs or benchmarks can be used for testing, but the free form of the output and the lack of established benchmark culture (warmup, repetitions, randomization) often suggests otherwise. The custom timer infrastructure category groups situations where the collected time samples are wrapped and propagated beyond reach of the immediately surrounding code with no hint on use.

To summarize the answer to Q1 with proprietary performance tests: no more than 33.2 % of the randomly selected projects query clock using the most common function calls, and no less than 3.4 % of the randomly selected projects query clock in direct relation to testing or benchmarking. The confidence intervals in Table 4 help extrapolate the ratios to general project population.

A2: How much does unit testing of performance change with time ...

Figure 3 shows how the number of projects that use JMH and the number of JMH benchmark methods in those projects changed over time, contrasted with the number of projects that use JUnit 4 and the number of JUnit 4 test cases in those projects. The gradually rising shape is similar in both cases,

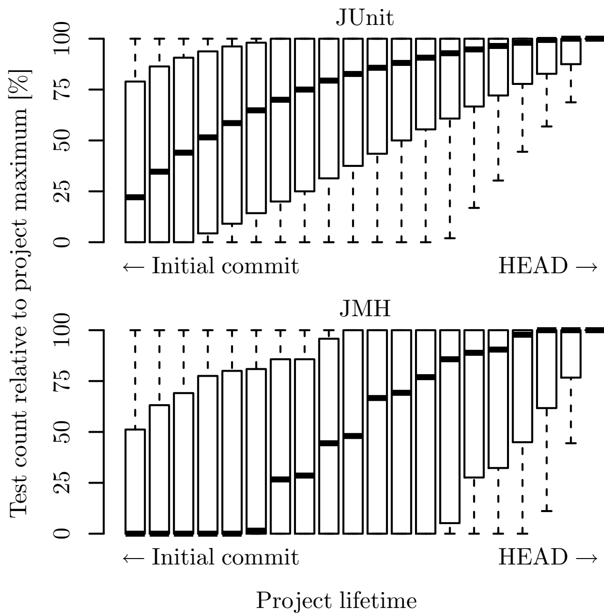

Figure 4: *Test count during project lifetime. Both axes are normalized, the project lifetime (X axis) stretches from the initial commit to the HEAD commit of each project, the test count (Y axis) ranges from zero to maximum number of test cases across the history of each project. Outliers are not shown, whiskers are at 1.5 IQR.*

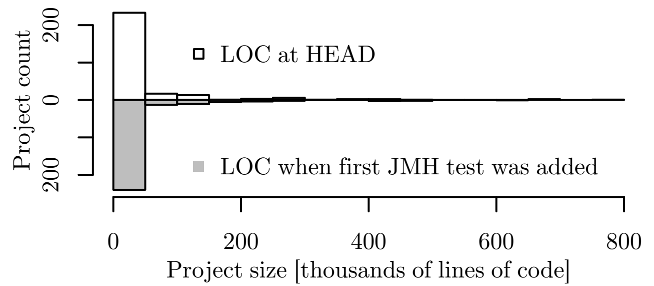

Figure 5: *Source code size for projects that use JMH.*

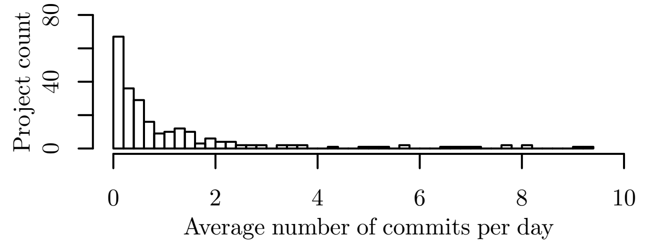

Figure 6: *Commit frequency for projects that use JMH. We show only projects with at least five commits and history longer than two weeks. One project with over 30 commits per day was omitted to preserve readable scale.*

the absolute numbers are naturally very different. Results for other performance testing frameworks are not given due to low use count, which makes it impossible to distinguish stagnating trends reliably.

To provide individual project perspective, Figure 4 shows how the number of benchmark methods or test cases changes over time within each project. Project data is aggregated by normalizing the time (X axis) to stretch from the initial commit to the HEAD commit, and the count (Y axis) from 0 % to 100 % of the maximum count observed. Again, results for other performance testing frameworks are not given due to low use count.

Figure 4 suggests that although both the benchmark method count and the test case count tend to increase across project lifetime, the first benchmark methods tend to appear later than the first test cases. There are several possible explanations. For one, the test first approach to implementation may be easier to do with functional testing than performance testing, because the test conditions are more obvious. Performance tests may also require more complete implementation than functional tests. The developer survey provides another insight – 64 % of developers touch performance test implementation only when addressing performance issues, only 28 % of developers maintains performance tests as often as other code.

A3: What kind of software projects are measuring performance ...

Figures 5 and 6 show the size and commit frequency of projects that use JMH, measured between the first commit that introduced any recognized performance test and the HEAD commit. We note that although some large projects are included, most projects that use JMH have less than 50 kLOC.

Table 5 provides rough classification for projects that use JMH. The table was constructed by labeling each project with information from the project documentation and then clustering similar labels until reasonable granularity was reached (each project is counted only once). Not listed are 66 analyzed projects whose clusters were smaller than 10 projects and 30 analyzed projects whose only purpose was to compare other projects against each other. Complete information is available in [30].

The developer survey provides additional information on the use of performance tests across categories – 80 % of developers report measuring their own code, 37 % of developers report measuring external code such as libraries. Also, 64 % of developers measure performance to compare alternatives. This can help explain the prominent position of tutorials and examples – obviously, explorative performance measurements provide important information during development.

Category	Count
Database (ORM, SQL ...)	33
Tutorials and examples	30
Networking and distributed systems	29
Algorithms	27
Data structures	22
Object serialization, parsers (XML, JSON, ...)	22
Web frameworks or plugins	18

Table 5: *JMH benchmark classification. Categories created through hierarchical clustering, small categories not shown, complete information in [30].*

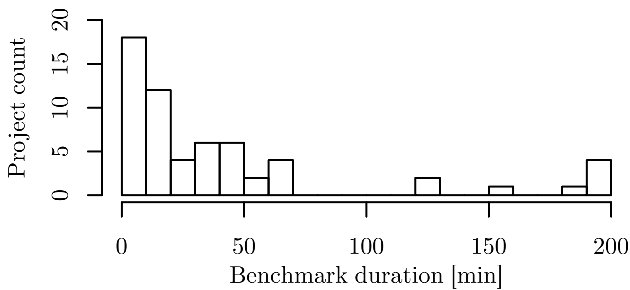

Figure 7: *Benchmark durations. Distribution of total benchmark execution time for all projects, measured with default settings.*

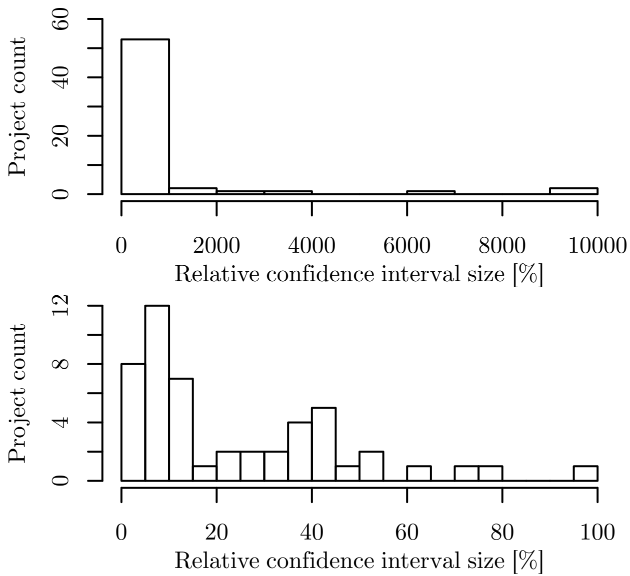

Figure 8: *Benchmark accuracy. Distribution of the worst relative confidence interval sizes of the mean for all projects. Relative confidence intervals of the mean were computed for each benchmark from one hour of measurements, each project is represented by the widest interval. The bottom graph is a zoom of the top one.*

A4: How long does unit testing of performance take ...

Figure 7 shows the distribution of the time needed to execute the performance tests with the default configuration for the measured projects, with each project counted in two measured versions. We note that the default configuration uses a small number of test executions (forks in JMH parlance), here median 3 and maximum 10, but still only about one third of the tests finishes within one hour, suggesting that collecting enough measurements for sensitive performance change detection can be an issue.

Figure 8 shows the accuracy obtained after one hour of measurement, expressed as the distribution of the relative confidence interval sizes for the mean. Specifically, we compute the relative 99 % confidence interval width (99 % confidence interval width divided by mean) for each benchmark method in each project using a bootstrap procedure, and then select the least accurate width to represent the project in the figure. The mean computation in the bootstrap procedure uses only as many samples as there are test executions in one hour, to properly address variation between executions (for rationale and exact computation we refer to [6]). We note that only about half of the projects have enough measurements to estimate the mean time of all benchmark methods with at most 20 % relative width, and about 20 % of the projects provides the least accurate estimate with over 100 % relative width.

The importance of the time needed to execute the performance tests becomes clear when considering the results of the developer survey, where 47 % of developers want to run performance tests on each commit, as opposed to 37 % of developers who are content with testing only each release. Additional comments also express the opinion that running tests on every commit may be too expensive.

Other observations in the developer survey also support the conclusion that regular performance testing on each commit remains an open goal. Overall, only 42 % of developers report running regular performance tests (although not necessarily on each commit). The processing of the measurements is also an issue – whole 77 % of developers report mostly manual processing, only 13 % mention automated plotting and only 6 % further automated evaluation.

A5: Does unit testing of performance reveal actual performance changes ...

Figure 9 shows the distribution of performance changes across individual benchmark methods, expressed as the ratio of the mean times from the two measured versions (new over old). Due to the large relative confidence interval widths in one hour of measurement, almost no change is statistically significant (99 % confidence intervals overlap).

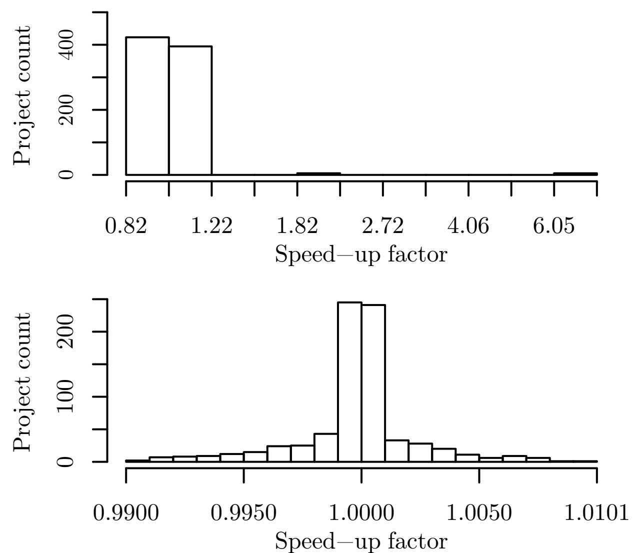

Figure 9: *Distribution of performance changes. A performance change is a ratio of mean execution times of each benchmark in the two measured versions. Logarithmic scale used for symmetry around 1. The bottom graph is a zoom of the top one.*

The developer survey offers additional explanation to the small changes in performance on Figure 9. Overall, 80 % of developers report acting on performance testing results in the past, however, less than one third of those report finding performance improvements or regressions – the remaining majority used the results to guide design decisions. Also, 13 % of developers mention seeing interesting results only rarely. This may suggest that performance testing is perceived as more useful during initial development stages, where it helps guide design.

To conclude, we summarize the remaining results of the developer survey, which concerns perceived obstacles to performance testing. These are in line with the previous findings – 61 % of developers believe automated evaluation would encourage more performance testing, 50 % of developers also want better build integration. Whole 31 % of developers feel the environment for implementing performance tests should be simpler, and 27 % mention budget issues.

4.1 Threats To Validity

Most threats to internal validity of our analysis were discussed together with the research questions. Where the recognition of performance testing frameworks is concerned, we classify the threat of false positives as low (we use source code parsing rather than mere text pattern searching), the threat of false negatives is limited to obscure usage patterns and unknown performance testing frameworks. For proprietary performance tests, the threats include possible classification mistakes due to manual processing and no support for native code implementation outside Java.

For building and executing the recognized performance tests, we avoid attributing specific causes to failures. Some projects may be broken from the very start, some projects may be correct but fail due to mismatch in platform requirements or dependencies. While some failures may be amenable to manual correction, some may be beyond fixing. Our survey does not have data to distinguish these cases.

Threats to external validity are tied to the high attrition rate. While we believe it is connected mostly to the general shape of open source projects and common backward compatibility issues, it still prevents most generalizations.

5. SPL INTEGRATION DESIGN

We want to reflect the results of the survey from Section 4 in the design of our SPL performance testing framework. In general, we strive to integrate SPL-based performance testing into the software development process in a manner similar to that of commonly used unit testing frameworks, with the goal of supporting three basic use cases: identifying performance regressions, capturing performance assumptions about (third-party) code, and providing performance documentation [6].

Briefly, in the case of a functional unit test, the test result depends on evaluating program state. A test is responsible for setting up the initial state, executing the test operations, and checking whether the resulting program state meets the expectations. Each test is the sole arbiter of the correctness of the tested behavior, and the testing framework just orchestrates test execution and provides the tests with means to indicate test results.

In contrast, in the case of a performance unit test, the test results depends on performance data collected during program execution. A test needs to provide a way to induce (representative) workload on a performance sensitive part of the program code, and a test condition in form of a hypothesis over performance measurements collected during test execution. The test itself does not determine the test result, because the workload may need to be executed many times to obtain data suitable for statistical analysis, and because the test condition may actually involve more than one version of the program code. Here the role of a testing framework is much more involved, because it needs to direct the collection of performance data from test executions (possibly for different versions), and then evaluate the test conditions on the collected data.

The specific requirements of performance unit tests have led our work on integrating SPL into development process towards a complex framework that supports different version control systems and build systems, performs measurements on demand and controls the amount of measured data to evaluate SPL-based test conditions, keeps a history of the measured data, and integrates with IDEs such as Eclipse and continuous integration tools such as Hudson. However, the results of the survey from Section 4 suggest that a different design direction may be needed to make SPL-based performance unit testing attractive to developers.

What we find interesting is that practically the only framework that is being used to conduct some form of performance evaluation or testing is JMH. Apparently, JMH is considered more useful or easier to use by the developers who have adopted some kind of performance testing into their project. Consequently, when it comes to integrating SPL-based performance testing into project development, we should consider the needs of these developers. We therefore turn to JMH for usability cues, and adapt the design of our SPL evaluation framework so that we can provide loosely-coupled building blocks that the developers can integrate into their projects as they see fit, without having to face the steep learning curve of a huge integrated framework. Before discussing the adjustments to the design of our performance testing framework, we first briefly review the typical usage of JMH in projects analyzed in our survey.

5.1 Overview of Typical JMH Usage

JMH provides developers with a simple way to implement and execute microbenchmarks correctly. Microbenchmarks typically execute operations that take very short time, but because they may be executed very often in many programs, they have the potential to influence the overall performance. Unsurprisingly, there are many pitfalls related to microbenchmarking – especially on a managed platform with a just-in-time compiler such as Java. JMH goes to great lengths to avoid them.

With JMH, the developer is responsible for providing the workload (i.e. the operation to be measured), while JMH takes care of benchmark execution and data collection. When finished, JMH typically reports aggregate statistics concerning the durations of the measured operations. From the developer's perspective, the usage of JMH is rather simple, as shown in Listing 1. Similar to using a testing framework such as JUnit, the developer only needs to create a class containing the operations to be measured in form of methods annotated with the `@Benchmark` annotation. Unlike JUnit, JMH does not evaluate the results in any way (apart from providing aggregate statistics), and leaves the responsibility for interpreting the results to the developer.

```java
public class SimpleBenchmark {
    @Benchmark
    public void measuredOperation() {
        // Implementation of the
        // operation to be measured.
    }
}
```

Listing 1: *Minimal JMH-based benchmark*

JMH supports additional annotations that provide more fine-grained control over microbenchmark execution, including benchmarking mode, output time units, output format, the number of warmup iterations, and many others. While they may require deeper understanding on the side of the developer, this does not really change the fact that a basic microbenchmark is very simple to write.

The microbenchmarks are also executed using JMH, which locates all the benchmark methods, generates benchmarking harness around those methods and executes the microbenchmarks in separate virtual machines configured to disable inlining of the benchmark method. While JMH can run the benchmarks directly (from an IDE or a command line), it is typically used to build a self-contained JAR file with the benchmark, which can then be simply executed on a dedicated machine. In contrast to functional unit testing, this is an important prerequisite for obtaining measurements that are not influenced by interference that may be present on the developer's machine.

With respect to build system integration, JMH officially supports Maven through a plugin which provides a JMH benchmark project archetype. Community-supported bindings enable integration with other build systems and IDEs. The typical (and endorsed) way using JMH is to create a standalone Maven-based project which contains benchmark code and depends on the application JAR files. The artifact produced by the project is the benchmark JAR file, which can then be used to execute the benchmarks as needed.

5.2 Catering to the Use Case that Matters

Given that performance unit testing is not yet a common practice and its benefits are not as immediate as in the case of functional unit testing, we need to support the adoption of SPL-based performance unit testing in a lightweight and gradual fashion. Considering that the role of JMH is to produce performance data which the developers have to process and analyze themselves to determine whether there has been a performance regression, we believe that the next useful step would be to automate the detection of performance regressions based on the data collected by JMH.

Recall that SPL is a simple language that allows capturing performance assumptions as formulas evaluated using statistical tests applied to performance data. An SPL formula that captures the essence of performance regression testing can be actually as simple as $last \leq 1.05 \times base$. Attached to something that can produce performance data (e.g. a microbenchmark), this formula captures the assertion that the measured operation in the last software version is not more than 5% slower than in some base (e.g. previous) version. In a project using JMH, performance regression testing would mean subjecting the results of all microbenchmarks to such tests, with the corresponding SPL formulas adjusted to the nature of the tested operation.

While the SPL formula representing the performance assumption for this particular use case is trivial, evaluating the formula is not. To remove this burden from the developer, we provide a tool for evaluating given SPL formulas with given data. Just like JMH allows the developer to implement microbenchmarks correctly, the SPL evaluator allows the same developer to compare microbenchmark results between versions correctly and with confidence. Also like JMH, the SPL evaluator is a standalone tool, leaving the developer free to integrate it into a project in any way desired.

To automate performance unit testing, the developer has to ensure that benchmarks will be automatically executed to collect performance data for new software versions, and that the SPL evaluator will be executed with data representing the performance of the last and the base versions of the software. The specifics of the automation and performance data storage remain at the discretion of the developer.

Because JMH officially supports integration with the Maven build system, we also provide a Maven plugin intended to ease adoption of SPL and to simplify integration. We present the SPL evaluator and the Maven plugin in more detail.

5.3 SPL Evaluator Tool

The SPL evaluator provides a command-line interface to the SPL evaluation engine originally developed for our much heavier-weight SPL-based performance unit testing framework. Decoupling the evaluator from the framework provides more opportunities for reuse and integration in existing build systems.

To illustrate the operation of the evaluator, let us again consider the simple SPL formula that can be used for performance regression testing: $last \leq 1.05 \times base$. An SPL formula is one of the evaluator inputs, and the evaluator expects to find it in a text file within the `META-INF` directory of a JAR file, in an external text file, or on the command line, in that order.

In the formula, the relation operator ($\leq$) represents a statistical test, the symbols *last* and *base* identify the data sets on which to perform the test, and the constant represents a scaling factor for the values from the *base* data set.

Conceptually, the symbols *last* and *base* denote the software versions that are the subjects of the performance test. However, the SPL evaluator is not tied to any particular version control system, and has no notion of the concept of a version – all it cares about is whether it can find data associated with the names used in an SPL formula. The evaluator looks for the data sets in directories corresponding to the identifiers used in the formula, relative to a base directory that can be specified on the command line.

In addition, the evaluator can read external files providing custom mapping between the names used in SPL formulas and the names to look for in the base directory of benchmark results. Using this mechanism, the developer can associate the symbols used in SPL formulas with version identifiers specific to a particular version control system, such as commit identifiers in Git. It should be possible to configure most version control systems to update the mapping file automatically, thus keeping a name such as *last* always mapped to the latest commit in a particular branch.

Finally, provided with all the required information, the evaluator performs the statistical tests necessary to evaluate

the given formula, and returns the result. If the result cannot be computed, for example because there is not enough data, the evaluator will return an alternative result specifying the reason. The developer is then responsible for providing more data for the software versions being tested.

5.4 JMH SPL Maven Plugin

It is important to note that the SPL evaluator presented in the previous section is not tied to JMH in any way, except that it can use data produced by JMH.[3] The actual integration of SPL and JMH occurs at the level of the build system and only affects the project containing JMH benchmarks.

To make adoption of SPL-based performance unit testing as non-intrusive as possible, we have created a Maven plugin to aid with basic integration tasks. The plugin provides three execution goals bound to different phases of the JMH benchmark project's build cycle.

The first goal, `spl_annotation`, is bound to the `generate-sources` build phase, and its purpose is to generate a single source file with the definition of the `@SPLFormula` annotation. The annotation can then be used to attach SPL formulas to JMH benchmark methods. This goal serves to avoid introducing additional compile dependencies into the JMH benchmark project – instead, the necessary file is simply injected into it.

The second goal, `formula_extractor`, is bound to the `compile` build phase, and its purpose is to scan the compiled benchmark classes for annotations containing SPL formulas. All SPL formulas are then stored into a text file within the `META-INF` directory of the benchmark JAR file produced by JMH in the subsequent `package` build phase.

The third goal, `data_saver`, is bound to the `verify` build phase, and its purpose is to execute the JMH benchmarks and store the collected performance data for later use with the SPL evaluator. This goal is important because the JMH benchmarks need to be run with specific options to store the results into a revision-specific directory, and to output raw data needed for SPL formula evaluation. By providing this goal, we avoid requiring the developer to change the JMH execution options when adopting SPL-based performance unit testing.

In summary, through the plugin we provide a thin, non-intrusive layer on top of a JMH benchmark project. This layer enables annotating JMH benchmarks with SPL-formulas and using JMH as a provider of performance data. To facilitate SPL-based performance unit testing based on the collected performance data, we provide a standalone SPL formula evaluator. As a result, developers can introduce SPL-based performance unit testing into their software project gradually, without having to adopt a heavy-weight approach imposed by a fully integrated testing framework. However, they are left with the responsibility for automating the performance testing process and integrating it into their development process.

6. CONCLUSION

Coming back to the question posed in the title of this paper, we have to conclude that as far as Java open source projects from the GitHub repository are concerned, we are not there yet.

[3]It requires JMH with support for reporting raw measurement data, which was introduced in version 1.14.

Of the 99019 projects that we have analyzed, only 0.37 % (370) actually use any performance testing framework. Of those projects whose performance tests we have executed, only 62 % produce performance measurements within 4 hours. Considering a broader class of projects that query clock using functions available in Java, only 3.4 % of sampled projects (using a random sample of 1000 projects) obviously implement performance tests or benchmarks.

Focusing on the usage of the JMH framework, performance tests are usually introduced later in the lifetime of a project (compared to functional unit tests), and can be mostly found in projects with less than 50 kLOC. Using the default settings, only about one third of the tests finishes within one hour, but more importantly, only half of the projects using JMH has enough measurements to estimate the mean time of all benchmark methods at 99 % confidence level with at most 20 % relative confidence interval width. About 20 % of projects using JMH can only estimate the mean execution time with a relative confidence interval width exceeding 100 %. Consequently, in one hour of execution, almost none of the tests can provide enough data to detect other than very obvious changes in performance. Collecting enough data to achieve higher sensitivity would prevent such tests from being executed on every commit. Observations on the relationship between accuracy and execution time suggest that careful measurement scheduling, rather than simple execution of all tests on all commits, is needed to achieve reasonable sensitivity at reasonable cost.

Survey responses from 111 developers who use the JMH performance testing framework indicate automation of performance testing is a major issue – 77 % of developers report processing results manually, automated evaluation and build integration are listed by 61 % and 50 % of developers, respectively, as needed features. The developers also recognize that correct test implementation is important – 72 % report trust in results as their reason for choosing the JMH framework, and 31 % mention need for simpler test implementation.

The issue of test execution time is further highlighted by 47 % of developers preferring to run performance tests at commit time, contrasted with 37 % of developers preferring to run performance tests only before each release.

Also interesting is the role of performance tests in enabling reasoning about performance. Only 23 % of developers report regularly acting on performance improvements or regressions, as opposed to 57 % of developers who report using performance tests for design decisions.

Our own effort in the area of performance unit testing is centered around SPL, a performance testing framework and formalism to express performance assumptions that can be validated through statistical testing. The results of our survey indicate that, contrary to our previous development, SPL should consider a light and flexible design to make it easier to introduce performance testing into a software project. Inspired by JMH as the most popular framework among projects that do any performance testing, we have focused on providing the essential building blocks of SPL while reusing tools such as JMH that the developers have already adopted, and otherwise leaving much of the responsibility for the actual implementation of the performance testing process with the developers.

Additional resources to complement our submission are available at [30].

Acknowledgments

We gratefully acknowledge the help of developers who have completed our survey. This work was partially supported by Charles University Institutional Funding (SVV) and by the Research Group of the Standard Performance Evaluation Corporation (SPEC).

7. REFERENCES

[1] S. Balsamo, A. D. Marco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: a survey. *IEEE Transactions on Software Engineering*, 30(5), 2004.

[2] V. Bergmann. ContiPerf. http://databene.org/contiperf, 2012.

[3] C. Bezemer, E. Milon, A. Zaidman, and J. Pouwelse. Detecting and analyzing I/O performance regressions. *Journal of Software: Evolution and Process*, 26(12), 2014.

[4] P. Brebner. Thoughts on the ABS Census website crash on census night (9 August 2016). http://www.performance-assurance.com.au/thoughts-on-the-abs-census-website-crash-on-census-night-9-august-2016/, 2016.

[5] L. Bulej et al. Capturing Performance Assumptions Using Stochastic Performance Logic. In *Proc. of ICPE '12*. ACM, 2012.

[6] L. Bulej et al. Unit testing performance with Stochastic Performance Logic. *Accepted for Automated Software Engineering*, 2016.

[7] Clarkware Consulting, Inc. JUnitPerf. http://clarkware.com/software/JUnitPerf.html, 2009.

[8] E. Daka and G. Fraser. A Survey on Unit Testing Practices and Problems. In *Proc. of ISSRE '14*. IEEE Computer Society, 2014.

[9] A. B. de Oliveira, J.-C. Petkovich, T. Reidemeister, and S. Fischmeister. DataMill: Rigorous Performance Evaluation Made Easy. In *Proce. of ICPE '13*. ACM, 2013.

[10] E. Engström and P. Runeson. A Qualitative Survey of Regression Testing Practices. In *Product-Focused Software Process Improvement*. Springer, Berlin, Heidelberg, 2010.

[11] D. G. Feitelson, E. Frachtenberg, and K. L. Beck. Development and Deployment at Facebook. *IEEE Internet Computing*, 17(4), 2013.

[12] B. George and L. Williams. An Initial Investigation of Test Driven Development in Industry. In *Proc. of SAC '03*. ACM, 2003.

[13] W. Gottesheim. Challenges, Benefits and Best Practices of Performance Focused DevOps. In *Proc. of LT '15*. ACM, 2015.

[14] M. Greiler, A. v. Deursen, and M.-A. Storey. Test Confessions: A Study of Testing Practices for Plug-in Systems. In *Proc. of ICSE '12*. IEEE Press, 2012.

[15] C. Heger, J. Happe, and R. Farahbod. Automated Root Cause Isolation of Performance Regressions During Software Development. In *Proc. of ICPE '13*. ACM, 2013.

[16] V. Horký et al. Performance Regression Unit Testing: A Case Study. In *Computer Performance Engineering*, 8168 in LNCS. Springer Berlin Heidelberg, 2013.

[17] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and Detecting Real-world Performance Bugs. In *Proc. of PLDI '12*. ACM, 2012.

[18] JUnit. JUnit. http://junit.org/, 2006.

[19] G. Kick, C. Decker, P. Duffin, et al. Caliper. https://github.com/google/caliper, 2015.

[20] P. S. Kochhar, T. F. Bissyandé, D. Lo, and L. Jiang. Adoption of Software Testing in Open Source Projects–A Preliminary Study on 50,000 Projects. In *Proc. of CSMR '13*, 2013.

[21] J. Kroß, F. Willnecker, T. Zwickl, and H. Krcmar. PET: Continuous Performance Evaluation Tool. In *Proc. of QUDOS '16*. ACM, 2016.

[22] M. Linares-Vásquez, C. Vendome, Q. Luo, and D. Poshyvanyk. How developers detect and fix performance bottlenecks in Android apps. In *Proc. of ICSME '15*, 2015.

[23] Y. Liu, C. Xu, and S.-C. Cheung. Characterizing and Detecting Performance Bugs for Smartphone Applications. In *Proc. of ICSE '14*. ACM, 2014.

[24] J. D. McGregor. Test early, test often. *The Journal of Object Technology*, 6(4), 2007.

[25] N. Nagappan, E. M. Maximilien, T. Bhat, and L. Williams. Realizing quality improvement through test driven development: results and experiences of four industrial teams. *Empirical Software Engineering*, 13(3), 2008.

[26] A. Nistor, T. Jiang, and L. Tan. Discovering, Reporting, and Fixing Performance Bugs. In *Proc. of MSR '13*. IEEE Press, 2013.

[27] Oracle Corporation. Java Microbenchmarking Harness (JMH). http://openjdk.java.net/projects/code-tools/jmh/, 2016.

[28] Oracle Corporation, Project Kenai, and Cognisync. Japex Micro-benchmark Framework. https://japex.java.net/, 2014.

[29] P. Runeson. A Survey of Unit Testing Practices. *IEEE Softw.*, 23(4), 2006.

[30] P. Stefan, V. Horký, L. Bulej, and P. Tůma. Supplementary Material. http://d3s.mff.cuni.cz/resources/icpe2017, 2017.

[31] TestNG. TestNG. http://testng.org/, 2006.

[32] A. van Hoorn, J. Waller, and W. Hasselbring. Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis. In *Proc. of ICPE '12*. ACM, 2012.

[33] J. Waller, N. C. Ehmke, and W. Hasselbring. Including Performance Benchmarks into Continuous Integration to Enable DevOps. *SIGSOFT Softw. Eng. Notes*, 40(2), 2015.

[34] J. Walter, A. van Hoorn, H. Koziolek, D. Okanovic, and S. Kounev. Asking "What"?, Automating the "How"?: The Vision of Declarative Performance Engineering. In *Proc. of ICPE '16*. ACM, 2016.

[35] E. J. Weyuker and F. I. Vokolos. Experience with Performance Testing of Software Systems: Issues, an Approach, and Case Study. *IEEE Trans. Softw. Eng.*, 26(12), 2000.

[36] M. Woodside, G. Franks, and D. C. Petriu. The Future of Software Performance Engineering. In *Proc. of FOSE '07*. IEEE Computer Society, 2007.