Open access • Proceedings Article • DOI:10.1145/3196398.3196407

An evaluation of open-source software microbenchmark suites for continuous performance assessment — Source link

Christoph Laaber, Philipp Leitner

Institutions: University of Zurich, University of Gothenburg

Published on: 28 May 2018 - Mining Software Repositories

Topics: Suite and Benchmark (computing)

An Evaluation of Open-Source Software Microbenchmark Suites for Continuous Performance Assessment

Laaber, Christoph ; Leitner, Philipp

Abstract: Continuous integration (CI) emphasizes quick feedback to devel- opers. This is at odds with current practice of performance testing, which predominantely focuses on long-running tests against entire systems in production-like environments. Alternatively, software microbenchmarking attempts to establish a performance baseline for small code fragments in short time. This paper investigates the quality of microbenchmark suites with a focus on suitability to deliver quick performance feedback and CI integration. We study ten open-source libraries written in Java and Go with benchmark suite sizes ranging from 16 to 983 tests, and runtimes between 11 minutes and 8.75 hours. We show that our study subjects include benchmarks with result variability of 50% or higher, indicating that not all benchmarks are useful for reliable discovery of slow- downs. We further arti cially inject actual slowdowns into public API methods of the study subjects and test whether test suites are able to discover them. We introduce a performance-test quality metric called the API benchmarking score (ABS). ABS represents a benchmark suite's ability to nd slowdowns among a set of de ned core API methods. Resulting benchmarking scores (i.e., fraction of discovered slowdowns) vary between 10% and 100% for the study subjects. This paper's methodology and results can be used to (1) assess the quality of existing microbenchmark suites, (2) select a set of tests to be run as part of CI, and (3) suggest or generate benchmarks for currently untested parts of an API.

DOI: https://doi.org/10.1145/3196398.3196407

An Evaluation of Open-Source Software Microbenchmark Suites for Continuous Performance Assessment

Christoph Laaber
Department of Informatics
University of Zurich
Zurich, Switzerland
laaber@ifi.uzh.ch

Philipp Leitner
Software Engineering Division
Chalmers | University of Gothenburg
Gothenburg, Sweden
philipp.leitner@chalmers.se

ABSTRACT

Continuous integration (CI) emphasizes quick feedback to developers. This is at odds with current practice of performance testing, which predominantely focuses on long-running tests against entire systems in production-like environments. Alternatively, software microbenchmarking attempts to establish a performance baseline for small code fragments in short time. This paper investigates the quality of microbenchmark suites with a focus on suitability to deliver quick performance feedback and CI integration. We study ten open-source libraries written in Java and Go with benchmark suite sizes ranging from 16 to 983 tests, and runtimes between 11 minutes and 8.75 hours. We show that our study subjects include benchmarks with result variability of 50% or higher, indicating that not all benchmarks are useful for reliable discovery of slowdowns. We further artificially inject actual slowdowns into public API methods of the study subjects and test whether test suites are able to discover them. We introduce a performance-test quality metric called the API benchmarking score (*ABS*). *ABS* represents a benchmark suite's ability to find slowdowns among a set of defined core API methods. Resulting benchmarking scores (i.e., fraction of discovered slowdowns) vary between 10% and 100% for the study subjects. This paper's methodology and results can be used to (1) assess the quality of existing microbenchmark suites, (2) select a set of tests to be run as part of CI, and (3) suggest or generate benchmarks for currently untested parts of an API.

KEYWORDS

continuous integration; software performance testing; microbenchmarking; empirical study; Java; Go

1 INTRODUCTION

Continuous integration (CI) [46] and continuous delivery (CD) [9] have become standard development practices in both, open source software (OSS) and commercial software projects. A core tenet of CI and CD is a focus on short cycle times, so as to innovate and deliver value to customers as quickly as possible. One challenge of this "move fast" mentality is ensuring software quality [42]. Regarding functional quality, unit and regression test suites can easily be automatically executed as part of the CI build. Unfortunately, state of the art performance testing practices are harder to align with CI. They usually encompass executing long-running test workloads against entire systems in production-like environments [23], which is both, hard to fully automate and too time-consuming to run for every build [12]. However, if performance tests are only selectively ran outside of the CD pipeline, there is a danger of deploying performance regressions to production, and it becomes harder to identify which code change has actually introduced a problem.

For some projects, one way to approach this issue may be software microbenchmarking, i.e., utilizing small code-level benchmarks which can, at least in theory, be run on every build [7].

Previous research on software microbenchmarks studied root causes [8], raised performance awareness of developers through documentation [19], and studied quantitative and qualitative aspects of microbenchmarking in Java [31, 47].

In this paper, we study the quality of existing software microbenchmark suites of ten OSS projects, five written in Java and five written in Go. The context of our study is to evaluate how useful these benchmarks are for continuous performance assessment, but our results also serve as a general survey of the quality of microbenchmarking in OSS. This work expands on the current state by (1) exploring benchmark-result variability in different environments, i.e., bare-metal and cloud; (2) highlighting similarities and differences of a dynamically compiled/VM-based language (i.e., Java) and a statically compiled language (i.e., Go); and (3) proposing an approach to assess the ability of a benchmark suite to find slowdowns within a defined subset of a project's application programming interface (API), identifying untested parts of it, and providing empirical data from OSS projects.

We thoroughly investigate the following research questions:

***RQ 1**: How extensive are the microbenchmark suites in the study subject projects?*

We investigate how large the microbenchmark suites are, and how long they take to execute. We find that the projects have suites sizing between 16 to 983 individual benchmarks, and take between 11 minutes and 8.75 hours to complete a single complete run in a standard configuration.

***RQ 2**: How stable are the results of microbenchmarks between identical executions?*

We repeatedly execute each suite five times and record the difference in results between runs. As the stability of results will depend on the execution environment, we run all suites in both, an instance rented from Google Compute Engine (GCE) as an example public cloud provider, and on a self-managed bare-metal server. We find that many projects include benchmarks with a maximum result

variability of 50% or higher even on bare-metal, indicating that not all benchmarks are useful for the reliable slowdown discovery.

***RQ 3**: Are the microbenchmark suites able to identify actual slowdowns?*

To evaluate how suitable microbenchmark suites are in finding actual slowdowns, we adopt a methodology inspired by mutation testing [22]. For each project, we identify 20 often-used public API methods by tracing their usage in the unit tests of other OSS projects on GitHub, and artificially introduce slowdowns into those methods. We study which of these artificial slowdowns can be found in our test setup and using the existing microbenchmarks. Using extensive experiments, we show that microbenchmark suites of our study subjects find slowdowns in between 10% and 100% of the selected 20 often-used methods. We further observe that many benchmarks appear redundant, in the sense that they seem to cover regressions in similar core methods.

Our results serve as a baseline for how extensive and variable existing OSS microbenchmark suites are in different environments. The proposed methodology is useful for developers to assess the quality of a software's microbenchmark suite, select a set of benchmarks to be run in fast-paced CD pipelines, and retrieving suggestions or even generating stubs of benchmarks for currently untested parts of a project's core API.

2 BACKGROUND

Performance testing is a widely used umbrella term for many different approaches. Most importantly, we distinguish between small-scale and highly granular performance tests on code level (software microbenchmarks or performance unit tests [19, 47]), and performance tests that target entire components or systems, typically under high load (load or stress tests). Microbenchmarks are executed in a unit-test-like fashion, whereas load tests bring up an entire production-like environment and run a defined workload against this system.

In the present work, we focus on performance testing via microbenchmarks. Typical performance counters used in such tests include the average execution time of a method, throughput, heap utilization, lock contention, and I/O operations. This section gives a brief overview of microbenchmarking in Java and Go.

Java Microbenchmarking Harness. Java OSS use a wide variety of microbenchmarking approaches, with Java Microbenchmarking Harness (JMH)[1] being the closest to a standard at the time of writing. JMH is part of OpenJDK since version 1.7. Other microbenchmarking tools, as reported by [31, 47], such as Caliper, Japex, or JUnitPerf hardly receive any OSS attention, are discontinued, or are not executable in an automated way. Hence, we only consider JMH in this study, which allows users to specify benchmarks in a similar notation to JUnit tests through an annotation mechanism. Every public method annotated with `@Benchmark` is executed as part of the performance test suite. Listing 1 shows an example benchmark from one of the study subjects (*protostuff*).

There are further configuration parameters that are either defined through annotations on test-class or test-method level, or through command-line parameters. These parameters include the

```
@Fork(1)
@Warmup(iterations = 5)
@Measurement(iterations = 10)
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.NANOSECONDS)
public class RuntimeSchemaBenchmark {
    ...
    @Benchmark
    public Int1 runtime_deserialize_1_int_field()
     throws Exception {
        Int1 int1 = new Int1();
        ProtobufIOUtil.mergeFrom(data_1_int, int1,
      int1RuntimeSchema);
        return int1;
    }
    ...
}
```

Listing 1: JMH example from the *protostuff* project.

number of warmup iterations, measurement iterations, forks (separate benchmark repetitions in fresh JVMs), and performance counters to measure. A single iteration of a benchmark takes 1*s* by default, and runs as many executions of the benchmark as it can fit into that time period. After each iteration the resulting performance counters are evaluated, aggregated, and reported to the user. Before the actual iterations, a defined number of warmup iterations are executed. Moreover, JMH has the notion of parameterized benchmarks, where the same benchmark implementation is run for each combination of input parameters. A partial sample output for the benchmark defined in Listing 1 is shown in Listing 2.

```
# Run progress: 0.00% complete, ETA 00:00:15
# Fork: 1 of 1
# Warmup Iteration   1: 11.417 ns/op
  ...
# Warmup Iteration   5: 8.955 ns/op
Iteration   1: 8.841 ns/op
Iteration   2: 8.819 ns/op
  ...
Iteration  10: 8.963 ns/op

Result: 8.884 +-(99.9%) 0.169 ns/op [Average]
  Statistics: (min, avg, max) = (8.751, 8.884,
     9.089), stdev = 0.112
  Confidence interval (99.9%): [8.715, 9.053]
```

Listing 2: JMH console output produced by the benchmark defined in Listing 1.

Benchmarking in Go. The Go programming language has a benchmarking framework included in its standard library[2], where benchmarks are defined as functions that have a name starting with `Benchmark` and a single parameter of type `*testing.B` in files with suffix `_test.go`. By default, benchmarks are executed once for at least 1*s* and average execution times are reported. In comparison to JMH, Go does not provide mechanisms for warmup iterations and parameterized tests. Nevertheless, through the command-line flag `-count`, a single benchmark can be repeated multiple times, similar to JMH's iterations. Moreover, forks are not supported, presumably because Go is not executed in a virtual-machine environment, arguably removing the need for warmup iterations and investigation of inter-virtual-machine-variance.

[1] http://openjdk.java.net/projects/code-tools/jmh

[2] https://golang.org/pkg/testing

3 STUDY SUBJECTS

We selected ten study-subject projects, five written in the Java programming language and five written in Go. We chose Java because it is a well established programming language, and consistently ranks highly in "most popular programming languages" rankings for various domains (e.g., Tiobe[3]). Go, on the other hand, is a fairly new programming language at the time of writing, which was introduced in 2009 and comes with a benchmarking library as part of its standard test framework. Microbenchmarking is predominantly used in middleware-style library projects (that is, projects such as *Log4J*, *RxJava*, or *etcd*, as opposed to end-user-facing applications such as Firefox), hence we focus on such projects.

To identify concrete projects, we used the public Google BigQuery dataset `fh-bigquery:github_extracts`[4] to retrieve the projects for both Java (BigQuery table *java_contents_2015*) and Go (BigQuery table *contents_go*). These tables contain all Java files hosted on GitHub that have been commited in the year 2015, and all Go files. We queried for projects that use JMH by filtering for the import statement `import org.openjdk.jmh`, and for projects that use Go's benchmarking library via a query for `*testing.B`. We mapped the identified files back to containing GitHub projects, and ordered the projects by stars and watchers. We then identified suitable study-subject projects for both languages by manually checking out the projects, verifying whether they compile, contain a non-trivial amount of benchmarks (> 10) that largely execute successfully, and whether we could find non-trivial usage of these projects in other OSS (again using Google BigQuery). Table 1 shows summary statistics for the projects we have ultimately chosen. The column "Commit SHA" in the table refers to the project's Git commit we have used for all our experiments. For all projects except *RxJava*, this SHA represents the current version of the master branch at the time of experimentation. For *RxJava*, we used the 1.x branch because version 2 was only released on October 29 2016, hence other projects on GitHub still used *RxJava* version 1 at the time of writing. Note that the low star and watcher count for *Log4j2* is due to the GitHub repository being merely a mirror of an Apache repository for this project.

4 RQ1: EXTENT OF BENCHMARK SUITES

4.1 Context

Improving the build time, i.e., the time it takes for a CI server to check out, compile, and test the build, is a constant concern for CI and CD projects, and the foundation of continuous performance assessment. This is primarily for two reasons: (1) Quick builds allow for fast feedback to the developer, reducing the time that is "wasted" waiting for builds that ultimately fail. (2) State of practice CI platforms restrict builds to a certain maximum runtime. A prominent example is TravisCI, which at the time of writing employs a build job timeout of maximum 120 minutes[5]. Hence, we first study how large (in number of benchmarks) the microbenchmark suites of our study subjects are, and how long they take to execute.

[3]https://www.tiobe.com/tiobe-index
[4]https://bigquery.cloud.google.com/dataset/fh-bigquery:github_extracts
[5]https://docs.travis-ci.com/user/customizing-the-build#Build-Timeouts

4.2 Approach

To establish a baseline of how much time is necessary to execute the microbenchmarking suites of our study subjects, we execute each suite five times in two different environments.

Firstly, similar to other works dealing with performance testing [3, 45], we executed all tests on a dedicated, non-virtualized ("bare-metal") server that we reserved exclusively for our performance testing. This server has a 12-core Intel Xeon X5670 @ 2.93GHz CPU with 70 GiB memory, and runs ArchLinux with a Linux kernel version 4.10.4-1-ARCH. It uses an Hitachi hard-disk (HUA722020ALA330) with 7200rpm and a 32MB cache. We did not execute any other user-space applications except `ssh` during test runs, but did not explicitly disable hardware optimizations. Note that our benchmark experiments do not fully utilize this high-performance test machine.

With our ultimate goal in mind — integrating performance testing with standard CI tooling — we also executed the test suites in virtual machines hosted in GCE. We used GCE's Infrastructure as a Service (IaaS) instances of machine type n1-standard-2[6] running Ubuntu 16.10 a GNU/Linux 4.8.0-46-generic x86_64 kernel. This machine type comes with two virtual CPUs and 7.5 GB of memory. We chose GCE because of its stability compared to other IaaS providers [32]. Experiments were conducted between March and May 2017, and in the `us-central-1a` region of GCE.

One challenge with this approach is that JMH allows to override benchmarking configurations (e.g., number of iterations, number of forks) via commandline parameters. Unfortunately, these customizations are not visible in the software repositories that we have access to. Hence, we use a uniform default JMH configuration for all Java projects (10 warmup iterations, 20 measurement iterations, 1 fork). For Go, we also repeat each benchmark 20 times. For all runs we record both, the duration required for running the entire test suite once with the above configuration, and the results of all benchmarks (this data will be used in Section 5).

4.3 Results and Analysis

Table 2 reports the size of the benchmark suites in number of benchmarks, the mean duration of a single execution of the entire suite, and the standard deviation of the duration between the different runs in hours. For JMH projects, we count every test method annotated with the `@Benchmark` annotation. For parameterized benchmarks, we count every combination of parameters as a different benchmark, as they appear to the developer as individual benchmarks during JMH execution. The Go benchmarking library does not have a comparable parameterization concept, therefore every benchmark is counted exactly once.

We observe that the size of microbenchmark suites among our study subjects varies widely. While Go projects tend to have smaller suites than the Java projects, this is not a strict rule. The two projects with the smallest microbenchmark suites are *gin* (a Go project with only 16 benchmarks) and *protostuff* (a Java project with 31 benchmarks). On the other hand, the two projects with the largest suites are both Java projects (*Log4j2* with 437, and *RxJava* with 983 benchmarks).

[6]https://cloud.google.com/compute/docs/machine-types

	Project	Description	URL	Commit SHA	Stars	Watchers
Java	Caffeine	High performance in-memory caching library	https://github.com/ben-manes/caffeine	64096c0	2461	182
	JCTools	Concurrency tools with focus on message queues	https://github.com/JCTools/JCTools	4a8775b	1062	136
	Log4j2	Well-known logging framework	https://github.com/apache/logging-log4j2	8a10178	44	255
	protostuff	Serialization library that supports schema evolution	https://github.com/protostuff/protostuff	6dfd8fe	551	63
	RxJava	Library for composing asynchronous programs through observable sequences	https://github.com/ReactiveX/RxJava	2162d6d	23770	1770
Go	bleve	Library for text indexing	https://github.com/blevesearch/bleve	0b1034d	3204	191
	etcd	Distributed consistent key/value store	https://github.com/coreos/etcd	e7e7451	13215	818
	fasthttp	High performance HTTP library	https://github.com/valyala/fasthttp	fc109d6	4315	218
	gin	HTTP web framework	https://github.com/gin-gonic/gin	e2212d4	9739	493
	otto	JavaScript parser and interpreter	https://github.com/robertkrimen/otto	1861f24	2880	138

Table 1: Overview of GitHub and benchmark metadata of study-subject projects.

	Project	Test Suite Size (#)	Exec. Time Mean (Hours)		Exec. Time Std.Dev (Hours)	
			Bare-Metal	Cloud	Bare-Metal	Cloud
Java	Caffeine	89	0.79	1.81	0.00	0.08
	JCTools	169	2.28	2.30	0.00	0.00
	Log4j2	437	4.76	4.61	0.00	0.01
	protostuff	31	0.25	0.26	0.00	0.00
	RxJava	983	8.75	8.72	0.02	0.01
Go	bleve	70	1.16	0.92	0.10	0.10
	etcd	41	0.63	0.72	0.17	0.11
	fasthttp	99	1.30	1.20	0.10	0.06
	gin	16	0.19	0.18	0.00	0.00
	otto	49	0.55	0.50	0.00	0.01

Table 2: Size of microbenchmark suites and durations of a single complete run on a bare-metal machine and in a public cloud.

Consequently, the total duration of a single test run also varies substantially between projects. *RxJava* takes close to 9 hours to finish a single run even with the relatively short default JMH configuration that we used. All other Java projects except *protostuff* require (often substantially) longer than a full hour for a single test run, making them infeasible to be executed as part of CI runs. Due to their smaller test suites, the Go projects also require less time for testing. *fasthttp* is the only Go project in our study to take over a full hour in both environments.

Moreover, it is evident that the total duration of a run generally correlates strongly with the size of the microbenchmark suite. This is due to how both, JMH and the Go benchmarking library, work (see also Section 2): they attempt to repeatedly execute a given small benchmark code for a defined duration (e.g., 1 second), and then report summary statistics such as how often the snippet could be executed in this defined duration. Consequently, every benchmark requires close to the same time for execution.

This also explains the generally low standard deviation of durations between our five experiment repetitions, and the often negligible difference between the duration in the bare-metal and public cloud environments (despite the bare-metal machine being much more powerful than the cloud instance that we used). The only exception here is *Caffeine*, which takes about 2.3 times longer in the cloud than in the bare-metal environment. This is due to the project employing time-consuming setups prior to each benchmark, which takes significantly longer on the smaller cloud instance.

RQ1 Summary. We find that the projects have microbenchmarking suites sizing between 16 to 983 tests, and take between 11 minutes and 8.75 hours for a single run. We conclude that many projects have microbenchmark test suites that *per se* take too much time to be used for continuous performance assessment.

5 RQ2: STABILITY OF BENCHMARKS

5.1 Context

A fundamental challenge of performance testing is that results for most performance counters are nondeterministic. A certain baseline level of variability even between identical runs of the same benchmarks is hence to be expected. How large this variability is depends on multiple factors, including the used programming language, the nature of the benchmark, and the stability of the environment [30]. However, establishing this baseline variability gives us a good basis to evaluate the usefulness of the benchmarks in the project's microbenchmark suites in different environments.

The variability of a benchmark affects the stability and reliability of its results. A lower result variability implies a detectability of smaller performance changes by a benchmark without reporting false positives (FPs) due to random fluctuations. Multiple factors, such as concurrent processes, virtualization, or I/O latencies can be the cause of these variations. Therefore it is inevitable to measure a project's benchmark-suite stability to assess how large a performance change needs to be before it can be detected reliably.

5.2 Approach

As a metric for stability of microbenchmarks, we propose the *maximum spread* among the mean performance counter outcomes of n repeated runs. This metric is calculated for benchmarks $b \in B$. In our experiments, we use $n = 5$, and formally define the repeated benchmark executions as the set $R = \{1, 2, 3, 4, 5\}$. We chose $n = 5$ to have a reasonable high number of complete suite executions to catch inter-run result variance. Any concrete run $r \in R$ produces a series of measurements for each $b \in B$, which we refer to as M_r^b, with $\forall b \in B, \forall r \in R, \forall m \in M_r^b : m \in \mathbb{R}^+$. Further, we denote the arithmetic mean of the benchmark measurements of benchmark b in run r as $\overline{M_r^b}$, and the arithmetic mean over all runs as $\overline{M^b}$. The maximum spread is then a function $maxSpread : B \to \mathbb{R}^+$, defined as Equation 1.

$$maxSpread(b) = max\left\{ \frac{\left| \overline{M^b_{r1}} - \overline{M^b_{r2}} \right|}{\overline{M^b}} \;\middle|\; r_1, r_2 \in R \right\} \quad (1)$$

Intuitively, *maxSpread* represents the largest distance in means (from the slowest to the fastest average benchmark execution) between any two runs in our experiment in percent from the overall mean. It gives us a measure about the worst variability *between repeated runs* (as opposed to a measure for variability *within* a run, such as the standard deviation of a series of measurements M^b_r would be). *maxSpread* therefore provides a lower bound for performance changes that are detectable by a benchmark in a particular environment. A *maxSpread* of 0 denotes that there is no variance in the results between runs, whereas, for instance, a value of 1 expresses that in the worst repetition the mean performance counter was twice as high as in the best (e.g., twice the mean execution time).

5.3 Results and Analysis

Figure 1 shows the distribution of the maximum spread of each benchmark for the study subjects in violin plot notation. To ease visual comprehension, we have limited the y-axis to 1.00 for Java and 0.5 for Go, even though some projects have a small number of outlier benchmarks with even higher *maxSpread*. All data and analysis scripts are provided in a replication package [26]. Further, Table 3 presents the distribution of the *maxSpread* of each benchmark for the study subjects. The table lists how many benchmarks (in absolute numbers and percent of the suite) fall into one of five buckets. The column "Benchs" lists for how many benchmarks we were able to generate results for all 5 runs, and only these are considered for *maxSpread* calculation. This number can be lower than the size of the entire suite as reported in Table 1, as we have experienced transiently failing benchmarks for a subset of projects.

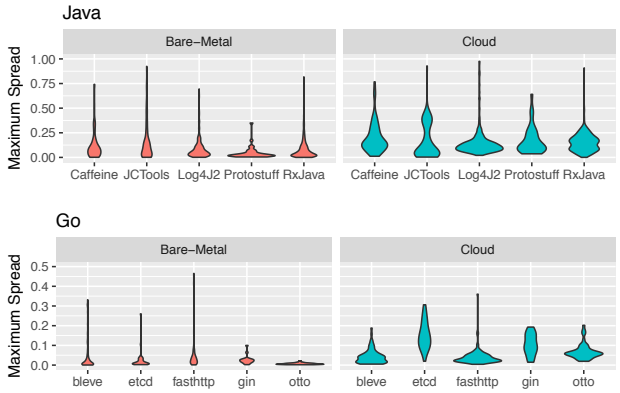

Figure 1: Distribution of each benchmark's *maxSpread* for all study-subject projects in bare-metal and cloud environments in violin plot notation.

Even though there are obvious differences between the projects in terms of benchmark stability, we are able to identify some common trends in the cloud environment. Firstly, even though Go projects have smaller test suites, their benchmarks tend to be substantially more stable than the Java ones. This is largely due to not being executed in a virtual machine. However, even in Go, two of five projects have some outlier benchmarks that are very unreliable. (*maxSpread* > 0.25). For two Go projects (*bleve* and *fasthttp*), more than 70% of benchmarks have a *maxSpread* < 0.05. Only one Go project (*etcd*) has relatively unstable benchmark results, with about two thirds of benchmarks having a *maxSpread* between 0.1 and 0.25.

All JMH projects have a substantial number of benchmarks with a *maxSpread* > 0.10. *JCTools* is interesting, as benchmarks for this project are largely either very stable (*maxSpread* < 0.05) or very unreliable (*maxSpread* > 0.25). We were unable to study in detail how the benchmarks that fall into one of these buckets differ for the present paper. We observe that the Java language runtime appears to introduce a baseline variability of benchmarking results in the order of 10% on GCE, as relatively few JMH benchmarks are more stable than that. This may be due to the well-documented challenges of benchmarking languages that use a virtual machine [14]. Hence we conclude that JMH is not a reliable tool for discovering small performance regressions in our test setup. A potential way to improve the microbenchmarking quality for these projects without improving the benchmarks themselves would be to increase the number of iterations and forks. While increasing these parameters potentially decreases *maxSpread*, it further increases the already long test durations (up to multiple days per run in the case of *RxJava*).

Another important, albeit unsurprising, observation is that both Go and Java projects produce more consistent benchmarking results on bare-metal than in in the cloud. This is due to the performance instability of public cloud instances themselves, as widely reported in literature [15, 21, 32]. For the Go projects except *etcd*, this effect is noticable but not dramatic, as the majority of benchmarks of each project still have *maxSpread* < 0.10, which we consider acceptable for many use cases. For *etcd*, and to a lesser extent *gin*, we observe an interesting effect where benchmarks which exhibit close to no variability in bare-metal become rather unreliable in the cloud (*maxSpread* between 0.05 and 0.3). The generally more unreliable Java benchmarks consistently become even less stable in the cloud, leading to benchmark suites where most benchmarks have a *maxSpread* of 0.1 or higher. Similiar to *etcd*, some Java projects with comparatively stable benchmarks in bare-metal become overproportional more unreliable in the cloud (*protostuff*, *RxJava*, as well as *JCTools* for a significant subset of its benchmarks).

We speculate that certain types of microbenchmarks (e.g., ones that are particularly IO-intensive) are hit harder by the performance instability of public clouds than others. However, a detailed analysis of this is out of scope in the present study. We conclude that microbenchmarking using a public cloud, as commonly used by public CI services, is possible. However, users need to, in some cases drastically, lower their expections with regard to how fine-grained slowdowns can realistically be discovered this way. Based on our data, only slowdowns of more than 50% in most Java projects, and of more than 25% in Go projects, are detectable with reasonable reliability on the used cloud instance.

	Project	Benchs		Bare-Metal								Cloud									
		BM	Cloud	0 - 0.05		0.05 - 0.1		0.1 - 0.25		0.25 - 0.5		>0.5		0 - 0.05		0.05 - 0.1		0.1 - 0.25		0.25 - 0.5		>0.5	
Java	Caffeine	89	86	32%	(28)	29%	(26)	21%	(19)	11%	(10)	7%	(6)	10%	(9)	12%	(10)	45%	(39)	21%	(18)	12%	(10)
	JCTools	168	169	35%	(59)	17%	(29)	27%	(45)	14%	(23)	7%	(12)	39%	(65)	15%	(26)	17%	(28)	27%	(46)	2%	(4)
	Log4j2	437	437	47%	(204)	24%	(104)	22%	(98)	5%	(23)	2%	(8)	7%	(30)	33%	(142)	49%	(216)	7%	(30)	4%	(19)
	protostuff	31	31	74%	(23)	13%	(4)	10%	(3)	3%	(1)	0%	(0)	7%	(2)	39%	(12)	32%	(10)	19%	(6)	3%	(1)
	RxJava	962	977	52%	(500)	20%	(195)	20%	(195)	6%	(57)	2%	(15)	11%	(106)	22%	(213)	55%	(543)	11%	(108)	1%	(7)
Go	bleve	80	70	80%	(64)	5%	(4)	9%	(7)	6%	(5)	0%	(0)	71%	(50)	20%	(14)	9%	(6)	0%	(0)	0%	(0)
	etcd	40	41	93%	(38)	0%	(0)	2%	(1)	2%	(1)	0%	(0)	5%	(2)	17%	(7)	66%	(27)	10%	(4)	2%	(1)
	fasthttp	99	100	72%	(71)	15%	(15)	3%	(3)	7%	(7)	1%	(1)	73%	(73)	21%	(21)	3%	(3)	1%	(1)	2%	(2)
	gin	16	16	88%	(14)	13%	(2)	0%	(0)	0%	(0)	0%	(0)	19%	(3)	37%	(6)	44%	(7)	0%	(0)	0%	(0)
	otto	49	49	100%	(49)	0%	(0)	0%	(0)	0%	(0)	0%	(0)	31%	(15)	53%	(26)	16%	(8)	0%	(0)	0%	(0)

Table 3: Maximum spread in various percentiles.

RQ2 Summary. Study subjects implemented in Go largely have very reliable benchmarks, with a $maxSpread$ below 0.05 in bare-metal. Conversely, the benchmark stability in most Java projects is more varied, with at least a quarter of benchmarks having a $maxSpread > 0.10$. In the cloud, benchmarks of all projects become substantially less stable, often leading to a $maxSpread$ greater than 0.25 (Java) and greater than 0.1 (Go).

6 RQ3: DETECTABILITY OF SLOWDOWNS

6.1 Context

Benchmark variability, as studied in Section 5, is only half the story when evaluating the quality of a microbenchmark suite, as even a microbenchmark suite with very stable benchmarks may still suffer from not-benchmarking a sufficiently large part of the project. With the approach proposed in this section, we can analyze the ability of existing microbenchmark suites in finding regressions, identify parts of the API that are not covered by microbenchmarks, and discovery methods which are tested for performance by multiple benchmarks.

6.2 Approach

Test coverage is a common quality metric for functional test suites [53]. To the best of our knowledge, there is currently no similar metric for microbenchmarks. We propose an API benchmarking score $ABS(K', B, v)$ that represents what fraction of (a subset of) the public API of a project is "covered" by the test suite B, i.e., in how many public methods $K' \subseteq K$ an actual slowdown of severity v can be found. Note that this metric is geared towards library or middleware-type projects, such as *Log4j2* or *RxJava*. We evaluate *ABS* for all study subject projects using the methodology outlined in Figure 2. This methodology is inspired by mutation testing [22], and relies on systematically introducing artificial slowdowns to observe whether the benchmark suite is able to catch these artificial regressions. Unfortunately, this methodology is very time-consuming. For two repeated runs, it requires $2 \cdot (|K'| + 1)$ executions of the entire microbenchmark suite, so we decided to use a small subset of the public API of only 20 often-used methods for our experiments (i.e., $|K'| = 20$). Even this fairly low number of considered methods our experiment still requires between ~7.5 hours (*gin*) and ~15.3 days (*RxJava*) to execute (cp. the execution time durations for a single run as reported in Table 1).

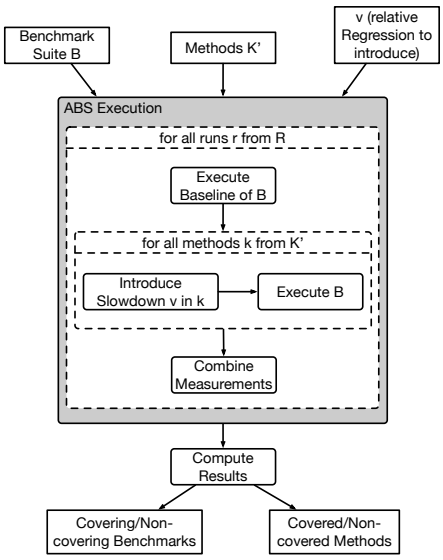

Figure 2: *ABS* calculation approach.

Finding Often-Used API Methods. This paper's approach is concerned with testing software performance of a performance-relevant part of a project's API. We denote this subset as $K' \subseteq K$, where K is the entire public API of a project. Generally, we expect project developers to be able to identify this subset using their domain knowledge. However, as we are basing our experiments on OSS projects where this kind of information is not publicly available, we chose to trace invocations of the study subjects' public API from unit tests of *other* OSS projects on GitHub. Recent work by [43] has proposed a sophisticated approach to extract API usage data for OSS projects. However, this work was not yet available when we conducted our research, so we elected to use a simpler approach to finding often-used methods.

Concretely, we identified additional GitHub projects per subject by running yet another BigQuery query searching for projects with `import` statements that indicated usage of our subjects. We again ordered the resulting projects by stars and watchers, manually iterated over them, executing the unit test suites of the found projects and tracing the number of invocations per public method of our subjects.

For Java projects, we used AspectJ to weave a custom-written simple tracer at load-time, while for Go projects we transformed

Java Project	Caffeine	JCTools	Log4j2	protostuff	RxJava
# Traces	$1638 * 10^6$	$100 * 10^6$	$4.5 * 10^6$	$126 * 10^6$	$1.2 * 10^6$
Go Project	bleve	etcd	fasthttp	gin	otto
# Traces	$972 * 10^3$	789	$25 * 10^3$	$21 * 10^3$	$6.4 * 10^6$

Table 4: Number of project-usage traces.

the abstract syntax trees (ASTs) of the case study projects before executing the unit test suites. The tool chain for tracing API usage is available on GitHub for both, Java [29] and Go [27]. Table 4 shows the total number of traces for each project that we generated using this tooling. Trace information is again collected in the replication package [26]. Instead of tracing a fixed number of projects per study subject, we have chosen a flexible number of projects (between four and six) that led to a sufficient number of traces for each subject. For all Go projects except *etcd*, we have in this way collected more than 10^3 unique API execution traces, and more than 10^6 for all Java projects. Only for *etcd*, we were unable to identify more than 768 traces using our methodology. We speculate that the reason for this is that *etcd* is not a traditional library project, but data store where we expect access is through network protocols rather than API invocations.

Determining Slowdowns (v) per Subject. Based on the assessment described in Section 5, we determine the slowdown that is potentially detectable by the 95$^{\text{th}}$ percentile of most instable benchmark of the subject's performance test suite. We chose the 95$^{\text{th}}$ percentile in order to remove the worst 5% benchmark results in terms of variability.

We define the slowdown to introduce v for a study subject project as $v = 2 \cdot max\{MS_i \mid 0 < i \leq \lfloor 0.95 \cdot |MS| \rfloor\}$, where MS is the ordered set (ascending) containing the maximum spreads of B such that $MS = \{maxSpread(b) \mid b \in B\}$. That is, we introduce a slowdown that is large enough that 95% of a project's benchmarks still have a *maxSpread* that is half as large as the slowdown, or lower.

To introduce slowdowns into the projects, we transform the ASTs (using javaparser[7] for Java and Go's standard ast package[8]) of the projects to measure the execution time $t(k)$ of the body of each method $k \in K'$ and append a relative slowdown $t'(k) = t(k) \cdot v$ at the end of the body such that the overall new execution time is $t(k) + t'(k)$. In Java we used `Thread.sleep` and in Go `time.Sleep` to implement the actual slowdown. Note that our tooling is unable to implement very small slowdowns (in the nano-second range), as the overhead of measuring the execution time and invoking the programming language's `sleep` method already delays by more than the intended slowdown. Hence, for very short-running methods, v should be understood as a lower bound, with actual slowdowns potentially being higher.

We have developed tool chains for both, JMH [28] and Go [27] projects, using the Groovy and Go programming languages respectively. These tools implement the approach outlined in Figure 2, and can be used by practitioners to evaluate *ABS* for their projects and based on the core API methods that they deem essential using their domain knowledge. The tools expect a configuration in JSON notation, which designates the project to evaluate, a list of methods forming K', and a slowdown to introduce v.

[7] https://github.com/javaparser/javaparser
[8] https://golang.org/pkg/go/ast

Calculating *ABS*. For every $k \in K'$ and every run $r \in \{1, 2\}$, our experiments produce a set of metrics M_r^{bk} for each benchmark $b \in B$. Based on this and the results from the baseline execution M_r^b, where no slowdown was introduced, we then define a function $reg : M \times M \rightarrow \{true, false\}$ which captures whether a difference between the sets of measurements can be determined with sufficient confidence (i.e., the benchmark detects a difference).

We choose the following implementation for reg. We conduct a Wilcoxon rank-sum test between M_r^{bk} and M_r^b. H_0 is that the sets of measurements do not differ, which we conservatively reject *iff* $p < 0.01$. However, preliminary cloud-reliability experiments and A/A testing have shown that relying only on statistical testing leads to many FPs, as changes in cloud performance often lead to statistically different measurements even if no slowdown was introduced [30]. Hence, we add another necessary condition, which is that the relative difference of the means needs to be at least half as large as the relative slowdown (Equation 2). $reg = true$ *iff* both the statistical test reports a significant difference and Equation 2 holds for *both* runs.

$$\frac{\left| \overline{M_r^{bk}} - \overline{M_r^b} \right|}{\overline{M_r^b}} \geq \frac{v}{2} \tag{2}$$

We consider a method k covered *iff* at least one benchmark registers a difference, i.e., $covered(k, B, v) = 1 \iff \exists b \in B : reg(M_r^{bk}, M_r^b) = true$. The *ABS* of a project is then the percentage of $k \in K'$ that are covered (Equation 3).

$$ABS(K', B, v) = 100 \cdot \frac{\sum_{k \in K'} covered(k, B, v)}{|K'|} \tag{3}$$

6.3 Results and Analysis

API Benchmarking Score. We present all resulting scores in Table 5. Further, all data is again available online [26]. To identify how sensitive scores are to individual unreliable benchmarks, we present three variants: "total" (includes all benchmarks), "relative", and "absolute". "Relative" and "absolute" scores describe two strategies where most unreliable benchmarks, as per the results discussed in Section 5, are excluded. We explore how scores change if (1) "relative" score, the worst 5% of benchmarks with respect to result stability (as per *maxSpread*), or (2) "absolute", benchmarks with a *maxSpread* > 0.5, are excluded. Intuitively these values should not be as prone to reported FPs as the full set of benchmarks ("total" score) is. Column "Excluded" lists how many benchmarks have been skipped in that way. Column "v" lists how large the introduced relative slowdown was per project.

Our results show that the Go projects, despite testing for substantially smaller slowdowns, have higher scores than the Java projects. Two of five Go projects have a scores of or close to 100% (*bleve* and *fasthttp*). *Gin* and *otto* have comparable scores to *Caffeine* and *JCTools*. Interestingly, *etcd* has a very low score at 10%. This could be due to *etcd* being a database project, and the public API may not reflect the most performance critical part of the system.

Given that many benchmarks in the Java projects have a high *maxSpread* and are hence prone to lead to FPs, excluding the most unstable benchmarks often leads to significantly lower scores. This is most evident for *Log4j2*, where all tests together lead to 80%

	Project	v	*ABS*			Excluded	
			Total	Rel.	Abs.	Rel.	Abs.
Java	Caffeine	2.6	65%	60%	60%	4	13
	JCTools	1.0	55%	55%	55%	8	4
	Log4J2	1.0	80%	40%	40%	22	19
	protostuff	1.0	40%	35%	35%	2	1
	RxJava	0.6	95%	95%	95%	50	22
Go	bleve	0.1	100%	100%	100%	4	0
	etcd	0.3	10%	10%	10%	2	1
	fasthttp	0.2	95%	95%	95%	5	2
	gin	0.2	65%	65%	65%	1	0
	otto	0.2	60%	60%	60%	2	0

Table 5: API benchmarking scores (*ABS*) of all subjects.

ABS. However, once we exclude unstable tests, the score drops to 40% (relative and absolute). We conclude that in *Log4j2*, many FPs during our test runs give a too optimistic view of the test suite's ability to find regressions. *RxJava* is the standout Java project of our case studies on multiple levels. As highlighted in Table 1, *RxJava's* benchmark suite is the most extensive in both runtime and size. Further, its benchmarking score is the best among Java projects with 95% which is similar to the best Go projects (*bleve* and *fasthttp*). Generally, the fact that benchmarks are more unreliable in the Java than in the Go projects also manifests in the difference of the three *ABS* results. All Java projects except *JCTools* and *RxJava* show a decreased score when filtering the most unreliable benchmarks. Go projects on the other hand have stable scores across all three types. Experiments with different combinations of *ABS* calculation showed, that the combination of difference in *maxSpread* and a hypothesis test produced the best results among the different benchmarking-score types.

Redundancy of Benchmarks. Another question is whether there typically is a one-to-one mapping between benchmarks and covered methods, or whether different benchmarks tend to redundantly find similar slowdowns. Figure 3 shows for each study subject how many methods from K' are covered by what number of benchmarks as a density plot. The higher the curve is in the right part of a figure, the more benchmarks find slowdowns in the same API methods.

A common pattern across multiple projects (*JCTools*, *Log4j2*, *protostuff*, *etcd*, *gin*, and *otto*) is that most methods are either not covered at all, or only covered by a small number of benchmarks, but a small subset of methods are covered by many benchmarks. The remaining projects *Caffeine*, *RxJava*, *bleve*, and *fasthttp* have more redundant benchmarks in comparison, indicating that benchmarks in these projects tend to be more coarse-grained. *Caffeine* has an evenly distributed set of benchmarks in terms of redundancy which shows that the number of benchmarks covering many methods steadily decreases. *RxJava* shows a high degree of redundancy with most methods being covered by around 35 benchmarks with an substantial amount covered by up to 160 benchmarks. Contrarily, *bleve* and *fasthttp* show a similar redundancy trend, where a relatively low number of benchmarks cover a few methods and the majority are benchmarks detecting around a third of the slowdowns in the selected API methods.

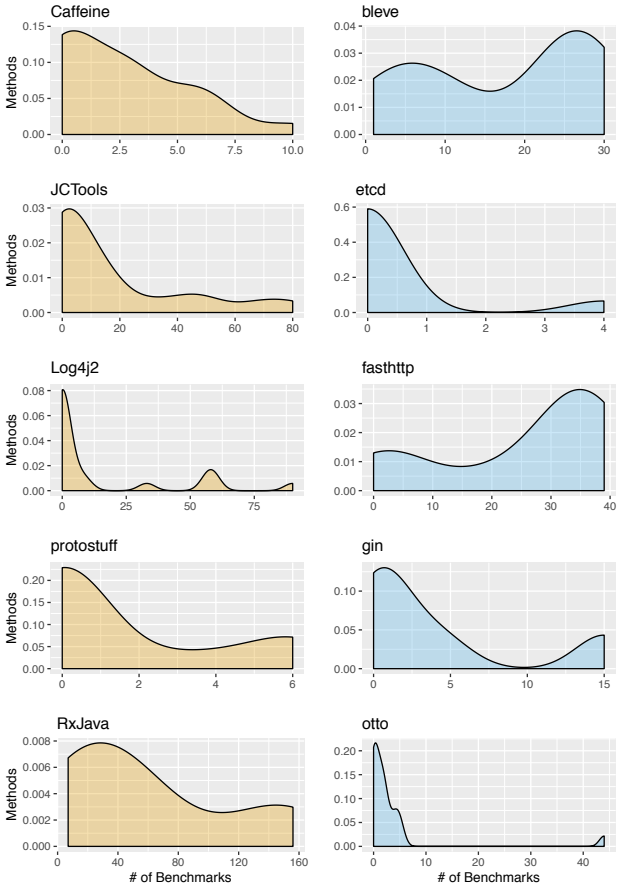

Figure 3: Mapping of benchmarks to covered API methods as density plot. Plots that lean more to the right indicate that a larger number of benchmarks find slowdowns in the same API methods, indicating redundancy.

RQ3 Summary. The API benchmarking score *ABS* is a quality metric to assess how well-tested a subset of a software's API is. *ABS* is determined similarly to mutation testing, where artificial regressions of project-dependent sizes are introduced into the API's subset, and the detection-scores of these is recorded by the microbenchmark suite. Resulting scores of the study subjects vary between 10% and 100%., and benchmarks often detect regressions in the same core method.

7 DISCUSSION

We presented an approach and metric to study the microbenchmark suites of ten OSS projects written in Java and Go. We showed that, depending on the environment, benchmark suites have different variability, which needs to be taken into consideration when evaluating performance. Through the proposed approach, we can assess the ability of a set of performance tests to detect actual performance changes in fine-granular software components. We now propose

three concrete usage scenarios for our approach and results, and discuss threats to the validity.

7.1 Usage Scenarios

Quality Assessment of Microbenchmark Suites. Software developers, both of OSS and commercial projects, can use our approach and the *ABS* to assess the quality of the benchmark suite of their own projects. Firstly, developers can find the best-fitting environment executing benchmarks by applying the methodology from Section 5. We showed that benchmark-result variability depends on the type and properties of the software. Hence, finding an environment that produces the most reliable results is key for continuous performance assessment. Secondly, by using the approach from Section 6, developers can determine "how good" their software's benchmark suite is in actually finding slowdowns. They can either use a similar process to ours for establishing K' (Section 6.2), or use their domain knowledge. In this way, developers can increase confidence in their performance testing. Moreover, the results that we have presented for ten OSS projects can serve as a point of reference for other OSS developers and their projects. Using our tool chain [26–29], setting up the environment, and executing the necessary steps to gather and analyze the data is straight-forward. As with any statistical test, especially in the context of performance evaluation and uncertain cloud environments, there will be FPs. The approach and results presented in Section 6 mitigates false-positives through multiple runs r where only a slowdown is reported if in *all* runs slowdowns are discovered. If users of our approach still experience too many FPs, an increase in runs should lead to more stable results.

Performance Test Case Selection. We argue that the approach taken in Section 6 can be used to steer the selection of a subset of benchmarks for CI integration. In future work, we will explore this angle and propose approaches to select an optimal subset of benchmarks for continuous performance assessment. These approaches build on the results of *ABS* in combination with an assessment of the available time budget or other constraints. We foresee that search-based methods [17] could be used to optimize benchmark selection. However, our approach is highly dependent on the identified performance-relevant set K'. Based on the selection of this subset, identified benchmarks with *ABS* potentially do not cover all possible performance problems detectable with a full microbenchmark suite or systematic, large-scale load testing. This subset is intended to act as a smoke test that should be seen as a sanity check rather than a full-fledged performance evaluation.

Generating Stubs for Additional Benchmarks. Using our methodology, developers can identify important parts of the API which are not covered by the an existing microbenchmark suite. Consequently, they can use this information directly to improve their performance tests. Additionally, we envision future tooling that will be able to generate new benchmark stubs for such uncovered core methods. In future research, we plan to explore these possibilities in more detail. For instance, we envision future developer tooling that analyzes all existing benchmarks, and then uses code generation to propose new benchmarks that follow the same style as the existing tests, but which are able to identify slowdowns in the so-far uncovered parts of the API.

7.2 Threats to Validity

There are certain threats and limitations to our approach and the paper's empirical results, that a reader should take into account. We discuss the most important ones in the following.

In terms of **external validity**, readers should consider the characteristics of our study subjects, and not generalize our results to other programming languages (with potentially different performance characteristics), application models, or industrial software. However, we have chosen a common programming language running on top of a virtual machine (Java), as well as a statically compiled language (Go) to improve representativeness. Future work should address this by extending our *ABS* results to other programming languages and paradigms (e.g., interpreted or functional). Given the time-consuming nature of the proposed approach in Section 6, i.e., between ~7.5 hours and ~15.3 days for a *single* project, scaling up to more than ten projects was considered infeasible. Results regarding performance testing in public clouds may vary if a different cloud provider (e.g., Microsoft Azure or Amazon AWS) is used instead of GCE. We have specifically investigated microbenchmarking suites, which evaluate software performance on fine-granular level. To detect full-system-performance degradations (e.g., network latencies, database access, load balancing), one might have to consider load tests or performance monitoring with release-engineering/live-testing strategies. Further, even though slowdowns are only one part of software performance, other performance counters (e.g., memory consumption, I/O operations, lock contention) were considered out of scope. We argue that the overall approach would not change drastically when considering these. Nonetheless, future research needs to investigate performance along these other dimensions.

In terms of **construct validity**, the most relevant threat is the selection of methods to test for in Section 6.2. We trace API usage in the unit tests of other open-source projects to get a simple, data-driven selection of often-used methods. However, there are two limitations to this method. Firstly, usage in unit tests does not necessarily correspond to usage in production. Secondly, absolute invocation counts are not necessarily a good proxy for importance. However, as the main contribution of the present study is the *ABS* methodology, hence a more comprehensive and time-consuming methodology to generate a more robust list of important methods, such as [43], was considered out of scope. Further, 20 methods may be too little to comprehensively evaluate a large project. Projects attempting to apply our methodology in practice may wish to calculate the *ABS* for a larger part of the public API, but need to keep in mind that the time necessary for running the experiments increases linearly with the number of considered methods. Moreover, *maxSpread* bases its variability on average statistics, whereas performance distributions are often long-tailed [11]. In future research, we want to investigate the effect performance measurements have on *maxSpread* and compare it to other measures such as averages with confidence-interval widths [47] or medians. The overridden JMH execution settings are another threat to the benchmark results. We decided to use our defaults as they reflect performance engineering best practices [14], and limit the execution time to a lower bound. This is especially necessary due to the expensive nature of our experiments (see Section 6). Finally, we were unable

to determine how sensitive the calculated scores are to the extent of the introduced slowdown. A follow-up sensibility study is required to investigate whether the benchmark scores change significantly when decreasing or increasing the slowdown.

8 RELATED WORK

Historically, research on software performance was often conducted in the context of system-scale load and stress testing [4, 23, 35, 51]. More recent research focuses on industrial applicability [13, 37], or how to reduce the necessary time for load testing [16]. Studies of microbenchmarking are scarcer, but have recently gotten more attention from research. [47] and [31] study quantitatively and qualitatively microbenchmarking practices in Java OSS. [47] indicate that microbenchmarks hardly find useful regressions and are often used for design decisions, which might not be the case for larger, industry-driven projects like *RxJava*. In their subject-selection approach projects with long-running benchmark suites were not considered. Compared to this work, we extend their findings by studying result variability on cloud infrastructure, which is arguable the way to go for integrating performance testing in CI. Further, we examine a different type of programming language that translates directly to machine code (i.e., Go). Studies from [19] and [8] are auxiliary to ours, focusing on raising performance-awareness of developers through generated API documentation and studying root causes of performance degradations reported by microbenchmarks respectively. None of these microbenchmarking works conducted a detailed analysis of the performance test suites' ability to find actual slowdowns, as is the focus of Section 6.

Studies on the nature of performance problems have found that these bugs tend to be particularly problematic. They take more time to be fixed than functional bugs [18], and require more experienced developers to do so [52]. Performance bugs are often dormant, and only become visible after a time when the system's usage changes [24]. Further studies have investigated performance regressions in specific domains, for instance in browser-based JavaScript [45] or Android mobile applications [33]. Another recent line of research related to the study of software performance is mining regressions from version histories [2, 34]. There has also been work on the automated fixing of (special types of) performance problems [4, 18, 38, 39]. Performance test regression selection research so far explored testing only performance-critical commits [2, 20], or focused on particular types such as collection-intensive software [36] and concurrent classes [40]. [10] propose selection of individual benchmarks based on static and dynamic data that assess whether a code change affects the performance of each benchmark. [7] tackle performance-regression testing through stochastic performance logic (SPL) which lets developers describe performance assertions in hypothesis-test-style logical equations. In the present work, we do not propose methods to actually fix issues, but suggest that our *ABS* methodology can be used to reduce existing test suites to a minimal required set, and propose stubs for valuable new benchmarks that should be added to a project's benchmark suite.

Further, there has recently been an increase of interest in the study of CI and CD builds, as well as in the types of build faults and failures that may occur. For instance, [41] categorized build failures on TravisCI and found no evidence that performance testing is currently a relevant part of CI builds. [5] specifically study testing on TravisCI, but neither mention performance testing. [49] study the CI builds of OSS and industrial applications, but again find little evidence of structured performance testing in the pipeline. Integrating performance testing into CI is only starting to gain momentum, with an initial publication by [6] on CI load testing. As an alternative to invoke a full load test run with every build (which may be infeasible for many systems), existing techniques to judge the performance sensitivity of a commit could be utilized [20]. In industrial practice, and especially in the context of Software as a Service (SaaS) applications, application performance management [1, 25] and partial rollouts (or canary releases) [44, 48, 50] are common approaches to unify CI with performance management. Fundamentally, these approaches eschew performance testing entirely, and instead rely on post-hoc identification of performance issues after deployment. However, these approaches are fundamentally targeted at SaaS rather than library and middleware projects, such as the ones studied in the present paper.

9 CONCLUSIONS

In this paper we studied the quality of software microbenchmark suites of ten open source projects written in Java and Go, and evaluated the suitability of these for continuously assessing software performance in CI environments. Firstly, we showed that benchmark suite sizes vary from 16 benchmarks to 983 with a mean execution time for a single run between 11 minutes and 8.75 hours. Secondly, we studied the result variability of the study subject's benchmark suites over identical repetitions, which we defined as *maxSpread*. Most Go benchmarks have a *maxSpread* below 5% in bare-metal and below 25% in cloud, whereas most Java benchmarks have one below 25% in bare-metal and below 50% in cloud. Thirdly, we evaluated how well the benchmark suites find actual slowdowns. We introduced a performance-test quality metric *ABS*, and showed that the study subject's score varies between 10% and 100%. Moreover, we found significant redundancy in the microbenchmark suites, indicating that it may be possible to execute only a subset of the suite as a smoke test within CI. The methodology and results of our study can be used to assess the microbenchmarking quality of other projects, identify missing tests that would improve the quality of existing benchmark suites, and select benchmarks to be run as part of CI.

ACKNOWLEDGMENTS

The research leading to these results has received funding from the Swiss National Science Foundation (SNSF) under project MINCA - Models to Increase the Cost Awareness of Cloud Developers (no. 165546). Conference attendance is partially supported by CHOOSE (Swiss Group for Software Engineering).

REFERENCES

[1] Tarek M. Ahmed, Cor-Paul Bezemer, Tse-Hsun Chen, Ahmed E. Hassan, and Weiyi Shang. 2016. Studying the Effectiveness of Application Performance Management (APM) Tools for Detecting Performance Regressions for Web Applications: An Experience Report. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR '16)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/2901739.2901774

[2] Juan Pablo Sandoval Alcocer and Alexandre Bergel. 2015. Tracking Down Performance Variation Against Source Code Evolution. In *Proceedings of the 11th Symposium on Dynamic Languages (DLS 2015)*. ACM, New York, NY, USA, 129–139. https://doi.org/10.1145/2816707.2816718

[3] Eytan Bakshy and Eitan Frachtenberg. 2015. Design and Analysis of Benchmarking Experiments for Distributed Internet Services. In *Proceedings of the 24th International Conference on World Wide Web (WWW '15)*. 108–118. https://doi.org/10.1145/2736277.2741082

[4] Cornel Barna, Marin Litoiu, and Hamoun Ghanbari. 2011. Autonomic Load-testing Framework. In *Proceedings of the 8th ACM International Conference on Autonomic Computing (ICAC '11)*. ACM, New York, NY, USA, 91–100. https://doi.org/10.1145/1998582.1998598

[5] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. Oops, My Tests Broke the Build: An Explorative Analysis of Travis CI with GitHub. In *Proceedings of the 14th International Conference on Mining Software Repositories (MSR)*.

[6] Andreas Brunnert and Helmut Krcmar. 2017. Continuous Performance Evaluation and Capacity Planning Using Resource Profiles for Enterprise Applications. *Journal of Systems and Software* 123 (2017), 239 – 262. https://doi.org/10.1016/j.jss.2015.08.030

[7] Lubomír Bulej, Tomáš Bureš, Vojtěch Horký, Jaroslav Kotrč, Lukáš Marek, Tomáš Trojánek, and Petr Tůma. 2017. Unit testing performance with Stochastic Performance Logic. *Automated Software Engineering* 24, 1 (01 Mar 2017), 139–187. https://doi.org/10.1007/s10515-015-0188-0

[8] Jinfu Chen and Weiyi Shang. 2017. An Exploratory Study of Performance Regression Introducing Code Changes. In *Proceedings of the 33rd International Conference on Software Maintenance and Evolution (ICSME '17)*. New York, NY, USA, 12.

[9] L. Chen. 2015. Continuous Delivery: Huge Benefits, but Challenges Too. *IEEE Software* 32, 2 (Mar 2015), 50–54. https://doi.org/10.1109/MS.2015.27

[10] Augusto Born de Oliveira, Sebastian Fischmeister, Amer Diwan, Matthias Hauswirth, and Peter Sweeney. 2017. Perphecy: Performance Regression Test Selection Made Simple but Effective. In *Proceedings of the 10th IEEE International Conference on Software Testing, Verification and Validation (ICST)*. Tokyo, Japan.

[11] Augusto Born de Oliveira, Sebastian Fischmeister, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. 2013. Why You Should Care About Quantile Regression. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. ACM, New York, NY, USA, 207–218. https://doi.org/10.1145/2451116.2451140

[12] Mikael Fagerström, Emre Emir Ismail, Grischa Liebel, Rohit Guliani, Fredrik Larsson, Karin Nordling, Eric Knauss, and Patrizio Pelliccione. 2016. Verdict Machinery: on the Need to Automatically Make Sense of Test Results. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*. 225–234. https://doi.org/10.1145/2931037.2931064

[13] King Chun Foo, Zhen Ming (Jack) Jiang, Bram Adams, Ahmed E. Hassan, Ying Zou, and Parminder Flora. 2015. An Industrial Case Study on the Automated Detection of Performance Regressions in Heterogeneous Environments. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 159–168. http://dl.acm.org/citation.cfm?id=2819009.2819034

[14] Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically Rigorous Java Performance Evaluation. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications (OOPSLA '07)*. ACM, New York, NY, USA, 57–76. https://doi.org/10.1145/1297027.1297033

[15] Lee Gillam, Bin Li, John O'Loughlin, and Anuz Pratap Singh Tomar. 2013. Fair Benchmarking for Cloud Computing Systems. *Journal of Cloud Computing: Advances, Systems and Applications* 2, 1 (2013), 6. https://doi.org/10.1186/2192-113X-2-6

[16] Mark Grechanik, Chen Fu, and Qing Xie. 2012. Automatically Finding Performance Problems with Feedback-Directed Learning Software Testing. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 156–166. http://dl.acm.org/citation.cfm?id=2337223.2337242

[17] Mark Harman and Bryan F Jones. 2001. Search-Based Software Engineering. *Information and Software Technology* 43, 14 (2001), 833 – 839. https://doi.org/10.1016/S0950-5849(01)00189-6

[18] Christoph Heger, Jens Happe, and Roozbeh Farahbod. 2013. Automated Root Cause Isolation of Performance Regressions During Software Development. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering (ICPE '13)*. ACM, New York, NY, USA, 27–38. https://doi.org/10.1145/2479871.2479879

[19] Vojtěch Horký, Peter Libič, Lukáš Marek, Antonin Steinhauser, and Petr Tůma. 2015. Utilizing Performance Unit Tests To Increase Performance Awareness. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering (ICPE '15)*. ACM, New York, NY, USA, 289–300. https://doi.org/10.1145/2668930.2688051

[20] Peng Huang, Xiao Ma, Dongcai Shen, and Yuanyuan Zhou. 2014. Performance Regression Testing Target Prioritization via Performance Risk Analysis. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 60–71. https://doi.org/10.1145/2568225.2568232

[21] Alexandru Iosup, Simon Ostermann, Nezih Yigitbasi, Radu Prodan, Thomas Fahringer, and Dick Epema. 2011. Performance Analysis of Cloud Computing Services for Many-Tasks Scientific Computing. *IEEE Transactions on Parallel and Distributed Systems* 22, 6 (June 2011), 931–945. https://doi.org/10.1109/TPDS.2011.66

[22] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Trans. Softw. Eng.* 37, 5 (Sept. 2011), 649–678. https://doi.org/10.1109/TSE.2010.62

[23] Z. M. Jiang and A. E. Hassan. 2015. A Survey on Load Testing of Large-Scale Software Systems. *IEEE Transactions on Software Engineering* 41, 11 (Nov 2015), 1091–1118. https://doi.org/10.1109/TSE.2015.2445340

[24] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and Detecting Real-world Performance Bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 77–88. https://doi.org/10.1145/2254064.2254075

[25] Chung Hwan Kim, Junghwan Rhee, Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. 2016. PerfGuard: Binary-centric Application Performance Monitoring in Production Environments. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, New York, NY, USA, 595–606. https://doi.org/10.1145/2950290.2950347

[26] Christoph Laaber and Philipp Leitner. 2018. Dataset and Scripts "An Evaluation of Open-Source Software Microbenchmark Suites for Continuous Performance Assessment". (2018). https://doi.org/10.6084/m9.figshare.5982253

[27] Christoph Laaber and Philipp Leitner. 2018. GoABS. https://github.com/sealuzh/GoABS/releases/tag/msr18. (2018).

[28] Christoph Laaber and Philipp Leitner. 2018. JavaABS. https://github.com/sealuzh/JavaABS/releases/tag/msr18. (2018).

[29] Christoph Laaber and Philipp Leitner. 2018. JavaAPIUsageTracer. https://github.com/sealuzh/JavaAPIUsageTracer/releases/tag/msr18. (2018).

[30] Christoph Laaber and Philipp Leitner. 2018. Performance testing in the cloud. How bad is it really? *PeerJ PrePrints* 6 (2018), e3507v1. https://doi.org/10.7287/peerj.preprints.3507v1

[31] Philipp Leitner and Cor-Paul Bezemer. 2017. An Exploratory Study of the State of Practice of Performance Testing in Java-Based Open Source Projects. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering (ICPE '17)*. ACM, New York, NY, USA, 373–384. https://doi.org/10.1145/3030207.3030213

[32] Philipp Leitner and Jürgen Cito. 2016. Patterns in the Chaos - A Study of Performance Variation and Predictability in Public IaaS Clouds. *ACM Transactions on Internet Technology* 16, 3, Article 15 (April 2016), 23 pages. https://doi.org/10.1145/2885497

[33] Mario Linares-Vasquez, Christopher Vendome, Qi Luo, and Denys Poshyvanyk. 2015. How Developers Detect and Fix Performance Bottlenecks in Android Apps. *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)* 00 (2015), 352–361. https://doi.org/doi.ieeecomputersociety.org/10.1109/ICSM.2015.7332486

[34] Qi Luo, Denys Poshyvanyk, and Mark Grechanik. 2016. Mining Performance Regression Inducing Code Changes in Evolving Software. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR '16)*. ACM, New York, NY, USA, 25–36. https://doi.org/10.1145/2901739.2901765

[35] Daniel A. Menascé. 2002. Load Testing of Web Sites. *IEEE Internet Computing* 6, 4 (2002), 70–74. https://doi.org/10.1109/MIC.2002.1020328

[36] Shaikh Mostafa, Xiaoyin Wang, and Tao Xie. 2017. PerfRanker: Prioritization of Performance Regression Tests for Collection-Intensive Software. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '17)*. ACM, New York, NY, USA, 23–34. https://doi.org/10.1145/3092703.3092725

[37] Thanh H. D. Nguyen, Meiyappan Nagappan, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. 2014. An Industrial Case Study of Automatically Identifying Performance Regression-Causes. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR 2014)*. ACM, New York, NY, USA, 232–241. https://doi.org/10.1145/2597073.2597092

[38] Adrian Nistor, Po-Chun Chang, Cosmin Radoi, and Shan Lu. 2015. Caramel: Detecting and Fixing Performance Problems That Have Non-Intrusive Fixes. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 902–912. http://dl.acm.org/citation.cfm?id=2818754.2818863

[39] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. 2013. Toddler: Detecting Performance Problems via Similar Memory-access Patterns. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 562–571. http://dl.acm.org/citation.cfm?id=2486788.2486862

[40] Michael Pradel, Markus Huggler, and Thomas R. Gross. 2014. Performance Regression Testing of Concurrent Classes. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA '14)*. ACM, New York, NY, USA, 13–25. https://doi.org/10.1145/2610384.2610393

[41] Thomas Rausch, Waldemar Hummer, Philipp Leitner, and Stefan Schulte. 2017. An Empirical Analysis of Build Failures in the Continuous Integration Workflows of Java-Based Open-Source Software. In *Proceedings of the 14th International Conference on Mining Software Repositories (MSR'17)*. ACM, New York, NY, USA.

[42] Julia Rubin and Martin Rinard. 2016. The Challenges of Staying Together While Moving Fast: An Exploratory Study. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 982–993. https://doi.org/10.1145/2884781.2884871

[43] Anand Ashok Sawant and Alberto Bacchelli. 2017. Fine-GRAPE: Fine-Grained APi Usage Extractor – an Approach and Dataset to Investigate API Usage. *Empirical Software Engineering* 22, 3 (01 Jun 2017), 1348–1371. https://doi.org/10.1007/s10664-016-9444-6

[44] Gerald Schermann, Dominik Schöni, Philipp Leitner, and Harald C. Gall. 2016. Bifrost: Supporting Continuous Deployment with Automated Enactment of Multi-Phase Live Testing Strategies. In *Proceedings of the 17th International Middleware Conference (Middleware '16)*. ACM, New York, NY, USA, Article 12, 14 pages. https://doi.org/10.1145/2988336.2988348

[45] Marija Selakovic and Michael Pradel. 2016. Performance Issues and Optimizations in JavaScript: An Empirical Study. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 61–72. https://doi.org/10.1145/2884781.2884829

[46] Daniel Ståhl and Jan Bosch. 2014. Modeling Continuous Integration Practice Differences in Industry Software Development. *Journal of Systems and Software* 87 (Jan. 2014), 48–59. https://doi.org/10.1016/j.jss.2013.08.032

[47] Petr Stefan, Vojtech Horky, Lubomir Bulej, and Petr Tuma. 2017. Unit Testing Performance in Java Projects: Are We There Yet?. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering (ICPE '17)*. ACM, New York, NY, USA, 401–412. https://doi.org/10.1145/3030207.3030226

[48] Chunqiang Tang, Thawan Kooburat, Pradeep Venkatachalam, Akshay Chander, Zhe Wen, Aravind Narayanan, Patrick Dowell, and Robert Karl. 2015. Holistic Configuration Management at Facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. ACM, New York, NY, USA, 328–343. https://doi.org/10.1145/2815400.2815401

[49] Carmine Vassallo, Gerald Schermann, Fiorella Zampetti, Daniele Romano, Philipp Leitner, Andy Zaidman, Massimiliano Di Penta, and Sebastiano Panichella. 2017. A Tale of CI Build Failures: an Open Source and a Financial Organization Perspective. In *Proceedings of the 33rd IEEE International Conference on Software Maintenance and Evolution (ICSME)*.

[50] Kaushik Veeraraghavan, Justin Meza, David Chou, Wonho Kim, Sonia Margulis, Scott Michelson, Rajesh Nishtala, Daniel Obenshain, Dmitri Perelman, and Yee Jiun Song. 2016. Kraken: Leveraging Live Traffic Tests to Identify and Resolve Resource Utilization Bottlenecks in Large Scale Web Services. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. GA, 635–651.

[51] Elaine J. Weyuker and Filippos I. Vokolos. 2000. Experience with Performance Testing of Software Systems: Issues, an Approach, and Case Study. *IEEE Transactions on Software Engineering* 26, 12 (Dec. 2000), 1147–1156. https://doi.org/10.1109/32.888628

[52] Shahed Zaman, Bram Adams, and Ahmed E. Hassan. 2012. A Qualitative Study on Performance Bugs. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories (MSR '12)*. IEEE Press, Piscataway, NJ, USA, 199–208. http://dl.acm.org/citation.cfm?id=2664446.2664477

[53] Hong Zhu, Patrick A. V. Hall, and John H. R. May. 1997. Software Unit Test Coverage and Adequacy. *Comput. Surveys* 29, 4 (Dec. 1997), 366–427. https://doi.org/10.1145/267580.267590