



Software Microbenchmarking in the Cloud. How Bad is it Really?

Downloaded from: <https://research.chalmers.se>, 2026-04-19 10:10 UTC

Citation for the original published paper (version of record):

Laaber, C., Scheuner, J., Leitner, P. (2019). Software Microbenchmarking in the Cloud. How Bad is it Really?. *Empirical Software Engineering*, 24(4): 2469-2508.
<http://dx.doi.org/10.1007/s10664-019-09681-1>

N.B. When citing this work, cite the original published paper.

Software Microbenchmarking in the Cloud. How Bad is it Really?

Christoph Laaber · Joel Scheuner ·
Philipp Leitner

the date of receipt and acceptance should be inserted later

This is a post-peer-review, pre-copyedit version of an article published in Empirical Software Engineering. The final authenticated version is available online at: <http://dx.doi.org/10.1007/s10664-019-09681-1>

Abstract Rigorous performance engineering traditionally assumes measuring on bare-metal environments to control for as many confounding factors as possible. Unfortunately, some researchers and practitioners might not have access, knowledge, or funds to operate dedicated performance-testing hardware, making public clouds an attractive alternative. However, shared public cloud environments are inherently unpredictable in terms of the system performance they provide. In this study, we explore the effects of cloud environments on the variability of performance test results and to what extent slowdowns can still be reliably detected even in a public cloud. We focus on software microbenchmarks as an example of performance tests and execute extensive experiments on three different well-known public cloud services (AWS, GCE, and Azure) using three different cloud instance types per service. We also compare the results to a hosted bare-metal offering from IBM Bluemix. In total, we gathered more than 4.5 million unique microbenchmarking data points from benchmarks written in Java and Go. We find that the variability of results differs substantially between benchmarks and instance types (by a coefficient of variation from 0.03% to > 100%). However, executing test and control ex-

Christoph Laaber
Department of Informatics, University of Zurich, Zurich, Switzerland
E-mail: laaber@ifl.uzh.ch

Joel Scheuner
Software Engineering Division, Chalmers | University of Gothenburg, Gothenburg, Sweden
E-mail: scheuner@chalmers.se

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

periments on the same instances (in randomized order) allows us to detect slowdowns of 10% or less with high confidence, using state-of-the-art statistical tests (i.e., Wilcoxon rank-sum and overlapping bootstrapped confidence intervals). Finally, our results indicate that Wilcoxon rank-sum manages to detect smaller slowdowns in cloud environments.

Keywords performance testing, microbenchmarking, cloud, performance-regression detection

1 Introduction

In many domains, renting computing resources from public clouds has largely replaced privately owning computational resources, such as server racks. This is due to economic factors, but also due to the convenience of outsourcing tedious data center or server management tasks (Cito et al 2015). However, one often-cited disadvantage of public clouds is that the inherent loss of control can lead to highly variable and unpredictable performance, for example due to co-located noisy neighbors (Leitner and Cito 2016; Farley et al 2012; Iosup et al 2011). Therefore, using cloud resources, such as virtual machines (VMs), in performance testing environments is a challenging proposition because predictability and low-level control over the hardware and software is key in traditional performance engineering. Nevertheless, there are many good reasons why researchers and practitioners might be interested in adopting public clouds as execution environments for their performance experiments. The experimenters might have insufficient access to dedicated hardware resources for conducting performance testing at a designated scale. They may wish to evaluate the performance of applications under “realistic conditions”, which nowadays often refers to cloud environments. They may wish to leverage industrial-strength infrastructure-automation tools (e.g., AWS CloudFormation¹) to easily provision resources on demand, which allows to massively parallelize the execution of large benchmarking suites.

In this paper, we ask the question whether using a standard public cloud as an execution environment for software performance experiments is always a bad idea. We focus on the cloud service model Infrastructure as a Service (IaaS) and on the performance testing type of software microbenchmarking in the programming languages Java and Go. IaaS clouds provide relatively low-level access and configurability while allowing for high scalability of experiments; and software microbenchmarks emerge as the performance evaluation strategy of choice for library-type projects. Software microbenchmarks can be seen as the unit-test equivalent for performance and are sometimes even referred to as performance unit tests (Stefan et al 2017). They are relatively short-running (e.g., < 1ms) performance tests against small software units (e.g., methods), which are typically tested in isolation without a fully-deployed system (as used for load testing). A microbenchmark is repeatedly

¹ <https://aws.amazon.com/cloudformation>

executed (called invocations) for a defined time period (e.g., 1s) and reports the mean execution time over all invocations (called iteration). The result of a microbenchmark is then the distribution of multiple iterations (e.g., 20).

In our previous work, we already studied performance stability of IaaS clouds using benchmarks for general system performance (Leitner and Cito 2016). We focused on running low-level, domain-independent system tests with different characteristics (e.g., IO or CPU benchmarks). Further, we have also previously executed software microbenchmarks in a bare-metal and one cloud environment to study the quality of open source microbenchmark suites (Laaber and Leitner 2018). Our results in these papers motivated a more extensive study, dedicated to the reliability of software microbenchmarking in public clouds, which is the core contribution of the present study. In particular, we quantify to what extent slowdowns can still be reliably detected in public clouds.

Concretely, we address the following research questions:

RQ 1 *How variable are microbenchmark results in different cloud environments?*

RQ 2 *Which slowdowns in microbenchmark results can we detect 95% of the time with at most 5% false positives?*

We base our research on 19 real microbenchmarks sampled from four open-source software (OSS) projects written in Java or Go. We aimed for 20 benchmarks, five for each project (see Section 3.1), however, due to a configuration execution error we lack results for one benchmark. We study cloud instances (i.e., VMs in cloud environments) in three of the most prominent public IaaS providers, namely Google Compute Engine (GCE), Amazon Elastic Compute Cloud (EC2), and Microsoft Azure, and we contrast these results against a dedicated bare-metal machine deployed using IBM Bluemix. We also evaluate and compare the impact of common deployment strategies for performance tests in the cloud, such as running experiments on different cloud instances of the same type or repeating experiments on the same instance. Hereby, we use randomized multiple interleaved trials as recently proposed as best practice (Abedi and Brecht 2017).

We find that result variability ranges from a coefficient of variation of 0.03% to more than 100% between repeated executions of the same experiments. This variability depends on the particular benchmark and the environment it is executed in. Some benchmarks show high variability across all studied instance types, whereas others are stable in only a subset of the environments. We conclude that instability originates from different sources including variability inherent to the benchmark, variability between trials (i.e., executions within an instance), and variability between instances.

We further find that two state-of-the-art statistical tests for performance evaluation (Bulej et al 2017), i.e., (1) Wilcoxon rank-sum with effect size

medium or larger and (2) overlapping confidence intervals of the mean computed with statistical simulation (bootstrapping) both falsely report high numbers of performance changes (i.e., false positives (FPs)) when in fact neither the benchmark nor production code has changed *and* the sample size (e.g., a single instance) is low. To mitigate this, benchmarks have to be repeatedly executed on multiple instances and multiple times within an instance to lower the numbers of FPs below an acceptable threshold ($\leq 5\%$ of 100 simulations), hence making it feasible to use cloud instances as performance-test execution environment.

Finally, we find that only 78% – 83% of the benchmark-environment combinations are able to reliably detect slowdowns at all, when test and control experiments are *not* run on the same instances and 20 instances each are used. Employing a strategy where test and control experiments are executed on the same instances, all benchmark-environment combinations find slowdowns with high confidence when utilizing ≥ 10 instances. In 77% – 83% of the cases, a slowdown below 10% is reliably detectable when using trial-based sampling and 20 instances. With respect to the difference between Wilcoxon rank-sum and overlapping confidence intervals, the Wilcoxon test is superior in two regards: (1) it is able to reliably detect smaller slowdowns, and (2) it is not as computational-intensive and therefore takes less time.

Following these findings, we conclude that executing software microbenchmarking experiments is possible on cloud instances, albeit with some caveats. Not all cloud providers and instance types are equally suited for performance testing, and not all microbenchmarks lend themselves to reliably detect slowdowns in cloud environments. In most settings, a substantial number of trials or instances is required to achieve robust results. However, running test and control groups on the same instances, optimally in random order, reduces the number of required repetitions (i.e., number of trials or instances). Practitioners can use our study as a blueprint to evaluate the stability of their own performance microbenchmarks within their custom experimental environment.

The remainder of this paper is structured as follows. Section 2 introduces relevant background information for this study, such as microbenchmarking, the Java Microbenchmarking Harness (JMH) framework, and IaaS cloud services. Section 3 outlines our research approach, describes the microbenchmark and cloud-provider selection, and details the execution methodology. Sections 4 and 5 discuss the study results for both research questions, while Section 6 discusses main lessons learned as well as threats to the validity of the study. Related research is discussed in Section 7, and finally the paper is concluded in Section 8.

2 Background

This section summarizes software microbenchmarking and IaaS clouds as important concepts we use in our study.

2.1 Software Microbenchmarking

Performance testing is a common term used for a wide variety of different approaches. In this paper, we focus on one specific technique, namely software microbenchmarking, sometimes also referred to as performance unit tests (Horky et al 2015). Microbenchmarks are short-running (e.g., < 1ms) unit-test-like performance tests that aim to measure fine-grained performance metrics, such as method-level execution times, throughput, or heap utilization. Typically, frameworks repeatedly execute microbenchmarks for a certain time duration (e.g., 1s) and report their mean execution time. The nature of these performance tests is different from traditional load tests where full applications are deployed and long-running load scenarios simulate the load of multiple users.

JMH is part of the OpenJDK implementation of Java and allows users to specify benchmarks through Java annotations. Every public method annotated with `@Benchmark` is executed as part of the performance test suite. Listing 1 shows an example benchmark from the *RxJava* project where the execution time and throughput of a latched observer are measured. Other examples measure logging (e.g., `logger.info`) in *Log4j2* or filter by search terms in the *bleve* text indexing library.

```
@State(Scope.Thread)
public class ComputationSchedulerPerf {

    @State(Scope.Thread)
    public static class Input
        extends InputWithIncrementingInteger {
        @Param({ "100" })
        public int size;
    }

    @Benchmark
    public void observeOn(Input input) {
        LatchedObserver<Integer> o =
            input.newLatchedObserver();
        input.observable.observeOn(
            Schedulers.computation()
        ).subscribe(o);
        o.latch.await();
    }
}
```

Listing 1: JMH example (*rxjava-5*) from the *RxJava* project.

The Go programming language includes a benchmarking framework directly in their standard library². This framework primarily follows the convention-over-configuration paradigm. For instance, benchmarks are defined in files ending with `_test.go` as functions that have a name starting with `Benchmark` (see Listing 2).

In our study, we use both JMH and Go microbenchmarks as test cases to study the suitability of IaaS clouds for performance evaluation.

² <https://golang.org/pkg/testing>

```

func BenchmarkTermFrequencyRowEncode(b *testing.B) {
    row := NewTermFrequencyRowWithTermVectors(...)
    b.ResetTimer()
    for i := 0; i < b.N; i++ {
        row.Key()
        row.Value()
    }
}

```

Listing 2: Go benchmarking example (*bleve-3*) from the *bleve* project.

2.2 Infrastructure-as-a-Service Clouds

The academic and practitioner communities have nowadays widely agreed on a uniform high-level understanding of cloud services following NIST (Mell and Grance 2011). This definition distinguishes three service models: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). These levels differ mostly in which parts of the cloud stack are managed by the cloud provider and what is self-managed by the customer. In IaaS, computational resources are acquired and released in the form of VMs or containers. Tenants are relieved from the burden of operating physical servers but are still required to administer their virtual servers. We argue that for the scope of our research, IaaS is the most suitable model at the time of writing as this model still allows for comparatively low-level access to the underlying infrastructure. Further, setting up performance experiments in IaaS is substantially simpler than doing the same in a typical PaaS system, where applications need to be adapted to provider-specific interfaces. Hence, we focus on IaaS in this paper.

In IaaS, a common abstraction is the notion of an instance: an instance bundles resources (e.g., CPUs, storage, networking capabilities, etc.) defined through an instance type and an image. The instance type governs how powerful the instance is supposed to be (e.g., what hardware it receives), while the image defines the software initially installed. More powerful instance types are typically more expensive, even though there is often significant variation even between individual instances of the same type (Ou et al 2012; Farley et al 2012). Instance types are commonly grouped into families, each representing a different usage class (e.g., general purpose, compute-optimized, or memory-optimized).

3 Approach

Traditionally, performance measurements are conducted in dedicated environments with the goal to reduce the non-deterministic factors inherent in all performance tests to a minimum (Mytkowicz et al 2009). Specifically, hardware and software optimizations are disabled on test machines, no background services are running, and each machine has a single tenant. These dedicated environments require high effort to maintain and have considerable acquisition costs. Conversely, cloud providers offer different types of hardware for

on-demand rental that have no maintenance costs and low prices. However, the lack of control over optimizations, virtualization, and multi-tenancy negatively affects performance measurements (Leitner and Cito 2016). To study the extent of these effects, we take the following approach. We sample a subset of benchmarks from four OSS projects written in two programming languages. These benchmarks are executed repeatedly on the same cloud instance as well as on different cloud-instance types from multiple cloud providers. The results are then compared in terms of variability and detectability of slowdowns.

Note that we are *not* benchmarking cloud infrastructure itself but rather software executed on it, which mostly falls into the category of library-like projects such as *Log4j2*. In this study, we are also not concerned with testing for performance of deployed and running applications (e.g., Agilefant³, Dell DVD Store⁴, or JPetStore⁵) in the fashion of a load test but evaluating performance of fine-grained software units (e.g., methods) using software microbenchmarks. We do not claim that software microbenchmarks are a replacement for load tests. Neither is this paper concerned with finding appropriate benchmarks for slowdown detection in cloud environments. Research into these other areas is required as part of potential future work. In the scope of this paper, we study the result variability and slowdown detectability of *existing* benchmark suites.

3.1 Project and Benchmark Selection

The study is based on 20 microbenchmarks selected from four OSS projects, two of which are written in Java and two in Go. Unfortunately, due to a configuration error, we are lacking all results for one benchmark (*bleve-1*) and consequently omit this benchmark from all remaining discussions. We decided to choose Java as it has been ranked highly in programming-language rankings (e.g., Tiobe⁶), is executed in a VM (i.e., the JVM) with dynamic compiler optimizations, and has a microbenchmarking framework available that is used by real OSS projects (Stefan et al 2017). The Go language complements our study selection as a new programming language being introduced in 2009. It is backed by Google, has gained significant traction, compiles directly to machine-executable code, and comes with a benchmarking framework⁷ as part of its standard library. We chose these languages due to their different characteristics, which improves generalizability of our results.

In an earlier study (Laaber and Leitner 2018), we investigated OSS projects in these languages that make extensive use of microbenchmarking. We chose real-world projects that are non-trivial and have existing microbenchmark suites. Table 1 shows detailed information about these projects, such as the Github URL, the commit snapshot used for all experiments, and the total

³ <https://github.com/Agilefant/agilefant>

⁴ <https://github.com/dvdstore/ds3>

⁵ <https://github.com/mybatis/jpetstore-6>

⁶ <https://www.tiobe.com/tiobe-index>

⁷ <https://golang.org/pkg/testing>

number of benchmarks in the project at the time of study. We also report popularity metrics, such as stars, watchers, and forks on GitHub. Note that *log4j2*'s star count is relatively low. This is due to the Github repository being a mirror of the main repository hosted directly by the Apache Software Foundation. The four selected projects represent good candidates of study subjects, as they are among the largest ones in their languages with respect to popularity (indicated by GitHub stars, watchers, and forks) and the size of their benchmark suites.

Project	Github URL https://github.com/	Commit	Stars	Watchers	Forks	Benchs (#)
Log4j2	apache/logging-log4j2	8a10178	369	58	256	437
RxJava	ReactiveX/RxJava	2162d6d	27742	1951	4882	977
bleve	blevesearch/bleve	0b1034d	3523	209	305	70
etcd	coreos/etcd	e7e7451	15084	920	2934	41

Table 1: Overview of study-subject projects.

For benchmark selection, we executed the *entire* benchmark suites of all study subjects five times on an in-house bare-metal server at the first author's university, requiring between 37.8 minutes (*etcd*) and 8.75 hours (*RxJava*) of execution time per trial (Laaber and Leitner 2018). For each project, we ranked all benchmarks in the order of result variability between these five trials and selected the ones that are: the most stable, the most unstable, the median, the 25th percentile, and the 75th percentile. Our intuition is to pick five benchmarks from each project that range from stable to unstable results to explore the effect of result variability on the ability to detect slowdowns. The selected benchmarks are summarized in Table 2, where the first benchmark of each project (e.g., *log4j2-1*) is the most stable and the last (*log4j2-5*) the most unstable according to our previous study.

3.2 Cloud Provider Selection

Within three of the most-prominent cloud providers, we choose three different families of instance types. The selected providers are Amazon with Amazon Web Services (AWS) EC2, Microsoft with Azure, and Google with Google Compute Engine (GCE). For each provider, we choose instance types in the families of entry-level general purpose (GP), compute-optimized (CPU), and memory-optimized (Mem). We expect instance types with better specifications to outperform the entry-level ones, and therefore this study establishes a base line of what is possible with the cheapest available cloud-resource options. Table 3 lists the selected instance types including information about the data-center region, processor and memory specification, and hourly prices at experiment time (summer 2017). All cloud instances run Ubuntu 17.04 64-bit.

Short Name	Package	Benchmark Name and Parameters
log4j2-1	org.apache.logging.log4j.perf.jmh	SortedArrayVsHashMapBenchmark.getValueHashContextData count = 5; length = 20
log4j2-2	org.apache.logging.log4j.perf.jmh	ThreadContextBenchmark.putAndRemove count = 50; threadContextMapAlias = NoGcSortedArray
log4j2-3	org.apache.logging.log4j.perf.jmh	PatternLayoutBenchmark.serializableMCNospace
log4j2-4	org.apache.logging.log4j.perf.jmh	ThreadContextBenchmark.legacyInjectWithoutProperties count = 5; threadContextMapAlias = NoGcOpenHash
log4j2-5	org.apache.logging.log4j.perf.jmh	SortedArrayVsHashMapBenchmark.getValueHashContextData count = 500; length = 20
rxjava-1	rx-operators	OperatorSerializePerf.serializedTwoStreamsSlightlyContended size = 1000
rxjava-2	rx-operators	FlatMapAsFilterPerf.rangeEmptyConcatMap count = 1000; mask = 3
rxjava-3	rx-operators	OperatorPublishPerf.benchmark async = false; batchFrequency = 4; childCount = 5; size = 1000000
rxjava-4	rx-operators	OperatorPublishPerf.benchmark async = false; batchFrequency = 8; childCount = 0; size = 1
rxjava-5	rx.schedulers	ComputationSchedulerPerf.observeOn size = 100
bleve-1	/search/collector/topn_test.go	BenchmarkTop10of50Scores
bleve-2	/index/upsidedown/benchmark_null_test.go	BenchmarkNullIndexingWorkers10Batch
bleve-3	/index/upsidedown/row_test.go	BenchmarkTermFrequencyRowDecode
bleve-4	/search/collector/topn_test.go	BenchmarkTop1000of100000Scores
bleve-5	/index/upsidedown/benchmark_goleveldb_test.go	BenchmarkGoLevelDBIndexing2Workers10Batch
etcd-1	/client/keys_bench_test.go	BenchmarkManySmallResponseUnmarshal
etcd-2	/integration/v3_lock_test.go	BenchmarkMutex4Waiters
etcd-3	/client/keys_bench_test.go	BenchmarkMediumResponseUnmarshal
etcd-4	/mvcc/kvstore_bench_test.go	BenchmarkStorePut
etcd-5	/mvcc/backend/backend_bench_test.go	BenchmarkBackendPut

Table 2: Overview of selected benchmarks. For JMH benchmarks with multiple parameters, we also list the concrete parameterization we used. The Go microbenchmark framework does not use the notion of parameters.

Provider	Data Center	Family	Instance Type	vCPU	Memory [GB]	Cost [USD/h]
AWS	us-east-1	GP	m4.large	2	8.00	0.1000
AWS	us-east-1	CPU	c4.large	2	3.75	0.1000
AWS	us-east-1	Mem	r4.large	2	15.25	0.1330
Azure	East US	GP	D2s v2	2	8.00	0.1000
Azure	East US	CPU	F2s	2	4.00	0.1000
Azure	East US	Mem	E2s v3	2	16.00	0.1330
GCE	us-east1-b	GP	n1-standard-2	2	7.50	0.0950
GCE	us-east1-b	CPU	n1-highcpu-2	2	1.80	0.0709
GCE	us-east1-b	Mem	n1-highmem-2	2	13.00	0.1184

Table 3: Overview of used cloud-instance types.

Additionally, we selected a bare-metal machine available for rent from IBM in its Bluemix cloud. A bare-metal instance represents the closest to a controlled performance testing environment that one can get from a public cloud provider. We used an entry-level bare-metal server equipped with a 2.1GHz Intel Xeon IvyBridge (E5-2620-V2-HexCore) processor and 16GB of memory, running Ubuntu 16.04 64-bit version, hosted in IBM’s data center in Amsterdam, NL. We specifically deactivated Hypertreading and Intel’s TurboBoost. Moreover, we attempted to disable frequency scaling, but manual checks revealed that this setting is ineffective and presumably overridden by IBM.

3.3 Execution

We use the following methodology to execute benchmarks on cloud instances and collect the resulting performance data. For each cloud instance type, as listed in Table 3, we create 50 different *instances*. On each instance, we schedule 10 consecutive experiment *trials* of each benchmark and randomize the order within each trial, following the method proposed by Abedi and Brecht (2017). Within each trial, every benchmark (e.g., *etcd-1*) consists of 50 *iterations* (e.g., using the `-i50` parameter of JMH) and every iteration produces a single *data point*, which reports the execution time in *ns*. For JMH benchmarks, we also run and discard 10 warmup iterations prior to the measurement iterations to reach steady-state performance (Georges et al 2007; Kalibera and Jones 2013). Note that 10 warmup iterations of 1 second might not be sufficient to bring the JVM into a steady state, depending on the microbenchmark under consideration. Ideally the warmup time and iterations would be dynamically determined (e.g., following an approach as outlined by Georges et al (2007)), which JMH does not support yet. For practical situations and the context of our study, 10 warmup iterations are considered sufficient. We use the same terminology of instances, trials, iterations, and data points in the remainder of the paper. These concepts are also summarized in Figure 1.

Formally, we run performance measurements for all benchmarks $b \in B$ (cf. Table 2) on all environments $e \in E$ (e.g., *AWS GP*). We refer to the combination of a benchmark b run on a particular environment e as configuration

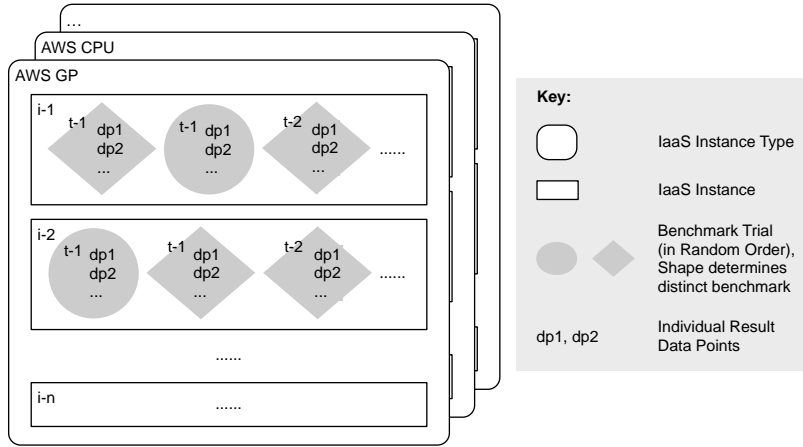


Fig. 1: Schematic view of instance types (e.g., *AWS GP*), instances (e.g., *i-1*), and randomized-order trials (e.g., *t-1*) on each instance.

$c \in C$ where $C = B \times E$. All configurations are executed on 50 instances $instances = 50$, with 10 trials per instance $trials = 10$, and 50 iterations $iters = 50$. The configuration results (i.e., data points reporting execution times in ns, see Section 2.1) of a single trial are defined as the set $M_{i,t}^c$ in Equation 1 (cf. Benchmark Trial in Figure 1, different shapes refer to distinct benchmarks), where the subscript i determines the instance, the subscript t represents the trial, and the superscript c describes the configuration where the respective results are from.

$$M_{i,t}^c = \{dp_j \in \mathbb{R}^+ \mid 0 < j \leq iters\} \quad (1)$$

Consequently, the configuration-execution order within an instance of a particular environment e is specified as the ordered set M_i^e in Equation 2, where B_t^e is the benchmark set B in potentially different randomized order of each trial t .

$$M_i^e = \{M_{i,t}^c \mid 0 < t \leq trials \wedge c = (b, e) \in C \wedge b \in B_t^e\} \quad (2)$$

The set of measurements for all benchmarks b , instances i , and trials t of a particular environment e is then defined as M^e in Equation 3.

$$M^e = \bigcup_{0 < i \leq instances} M_i^e \quad (3)$$

For setting up the instances we used Cloud Workbench⁸ (Scheuner et al 2014), a toolkit for conducting cloud benchmarking experiments. Cloud Workbench sets up machines with Vagrant⁹ and Chef, collects performance mea-

⁸ <https://github.com/sealuzh/cloud-workbench>

⁹ <https://www.vagrantup.com>

surement results, and delivers the results in the form of CSV files for analysis. We collected our study data between July and October 2017.

Using this study configuration, we collected more than 4.5 million unique data points from the selected benchmarks. However, due to the large scale of our experiments and the inherent instability of the environment, transient errors (e.g., timeouts) are unavoidable. We apply a conservative outlier removal strategy, where we remove all data points that are one or more orders of magnitude higher than the median. These outliers are due to the non-deterministic nature of cloud resources such as multi-tenancy, decreased network bandwidth, or instance volatility. None of the cloud providers offer information nor did we explicitly record cloud failure events that would explain the reasons for these outliers. Due to the long-tailed nature of performance-measurement data, we did not remove outliers more aggressively (e.g., 3 standard deviations from the mean) because they might be correct extreme values belonging to the long-tail of the result distribution. Table 4 lists for each instance type across all instances, trials, and iterations how many data points we have collected and how many data points remain after outlier removal.

Short Name	Instance Type	Total	Post-Cleaning
AWS GP	m4.large	474952	474656
AWS CPU	c4.large	475000	474999
AWS MEM	r4.large	474953	474951
Azure GP	D2s v2	472636	472122
Azure CPU	F2s	473132	473041
Azure MEM	E2s v3	470570	470400
GCE GP	n1-standard-2	474491	474490
GCE CPU	n1-highcpu-2	474725	474546
GCE MEM	n1-highmem-2	474716	474337
Bluemix (BM)	-	474436	474185

Table 4: Overview of the number of collected data points per cloud instance type, and how many data points remain after data cleaning.

In particular, Azure instances were most affected by outlier removal, where up to 8 of the studied benchmarks had at least one data point cleaned. The two benchmarks that have the most outliers removed are *log4j2-5* and *etcd-2*, which comes as no surprise as these two have also the highest variability (see Table 5). About 1% of the two benchmarks’ data points per instance type are cleaned.

4 Benchmark Variability in the Cloud

To answer RQ 1, we study the benchmarks of all projects in terms of their result variability in all chosen environments. For all configurations (i.e., all benchmarks on all instance types), we report the variability of each benchmark across all 50 instances, 10 trials on each instance, and 50 iterations

per trial. We use the coefficient of variation (CV), also referred to as relative standard deviation, of a set of benchmark measurements M in percent as the measure of variability $cv : M \rightarrow \mathbb{R}^+$. cv is defined as $cv(M) = \frac{\sigma(M)}{\mu(M)}$, with $\sigma(M)$ representing the standard deviation and $\mu(M)$ denoting the mean value of all data points of an set of measurements, where $M_{i,t}^c \in M$. CV is a statistical measure for dispersion among a population of values, in this case performance variability of microbenchark results as obtained from executions on cloud instances. CVs have been used previously in similar studies to ours, such as Leitner and Cito (2016).

The set of measurements of a specific configuration c across all trials and instances is defined as M^c in Equation 4.

$$M^c = \{M_{i,t}^{c'} \in M^e \mid c' = c\} \quad (4)$$

Further, the variability of a configuration (i.e., benchmark-environment combination) is defined as $V^{total,c}$ in Equation 5. $V^{total,c}$ represents the variability of a single configuration (i.e., the CV) of the union of *all* collected measurements of a benchmark b executed in an environment e across all instances i and trials t .

$$V^{total,c} = cv\left(\bigcup_{M_{i,t}^c \in M^c} M_{i,t}^c\right) \quad (5)$$

Table 5 reports these CV (in percent, that is $100 \cdot V^{total,c}$) variabilities for all studied configurations (e.g., *log4j2-1*, *AWS GP*) as numerical values and further provides an indication of the relative 95% confidence interval widths (RCIW) of these values. The RCIW describes the estimated spread of the population’s CV, as computed by statistical simulation (bootstrap with 1000 iterations). It provides an indication how variable the values in Table 5 are. A high variability of the benchmark in an environment (e.g., *log4j2-5* in all environments) does not necessarily mean that the CV value is highly variable itself (as indicated by the circles). \circ represents a RCIW below 30%, \bullet a RCIW between 30% and 60%, and \bullet a RCIW larger than 60%. Note that the dispersion observed in this table originates from three different sources of variability: (1) the difference in performance between different cloud instances of the same type, (2) the variability between different trials of an instance, and (3) the “inherent” variability of a benchmark, i.e., how variable the performance results are, even in the best case. Consequently, a large CV in Table 5 can have different sources such as an unpredictable instance type or an unstable benchmark. We elaborate on the different sources of variability later in this section.

4.1 Differences between Benchmarks and Instance Types

It is evident that the investigated benchmarks have a wide spectrum of result variability, ranging from 0.03% for *rxjava-1* on *Bluemix*, to 100.68% for

Benchs	AWS			GCE			Azure			BM
	GP	CPU	Mem	GP	CPU	Mem	GP	CPU	Mem	
log4j2-1	45.41 ●	42.17 ○	48.53 ○	41.40 ○	43.47 ○	44.38 ○	46.19 ○	40.79 ●	51.79 ○	41.95 ○
log4j2-2	7.90 ○	4.89 ○	3.92 ●	10.75 ●	9.71 ●	11.29 ●	6.18 ●	6.06 ○	11.01 ●	3.83 ○
log4j2-3	4.86 ●	3.76 ●	2.53 ●	10.12 ○	9.18 ●	10.15 ●	13.89 ●	7.55 ●	15.46 ●	3.02 ●
log4j2-4	3.67 ○	3.17 ●	4.60 ○	10.69 ●	9.47 ●	10.52 ○	17.00 ●	7.79 ●	19.32 ●	6.66 ○
log4j2-5	76.75 ○	86.02 ○	88.20 ○	83.42 ○	82.44 ○	80.75 ○	82.62 ○	86.93 ○	82.07 ○	77.82 ○
rxjava-1	0.04 ○	0.04 ○	0.05 ○	0.04 ●	0.04 ○	0.04 ○	0.05 ○	0.05 ○	0.27 ●	0.03 ○
rxjava-2	0.70 ○	0.61 ●	1.68 ●	5.73 ●	4.90 ●	6.12 ●	9.42 ●	6.92 ●	13.38 ●	0.49 ●
rxjava-3	2.51 ●	3.72 ●	1.91 ○	8.16 ●	8.28 ●	9.63 ●	6.10 ●	5.81 ○	10.32 ●	4.14 ○
rxjava-4	4.55 ●	4.18 ●	7.08 ●	8.07 ●	10.46 ●	8.82 ●	17.06 ●	10.22 ●	21.09 ●	1.42 ○
rxjava-5	5.63 ●	2.81 ○	4.04 ○	14.33 ●	11.39 ●	13.11 ●	61.98 ●	64.24 ○	21.69 ●	1.76 ●
bleve-2	1.57 ●	1.32 ●	4.79 ●	5.56 ●	6.09 ●	5.78 ●	5.97 ●	5.48 ●	13.29 ●	0.27 ○
bleve-3	1.13 ●	7.53 ●	7.77 ●	10.08 ●	10.74 ●	14.42 ●	7.62 ●	6.12 ●	14.41 ●	0.18 ○
bleve-4	4.95 ●	4.38 ●	5.17 ●	11.24 ●	12.00 ●	14.52 ●	8.18 ●	7.11 ●	15.24 ●	0.62 ○
bleve-5	10.23 ○	9.84 ○	8.18 ○	57.60 ○	58.42 ○	59.32 ○	52.29 ●	46.40 ●	52.74 ○	10.16 ○
etcd-1	1.03 ●	3.17 ●	1.56 ●	6.45 ●	5.21 ●	7.62 ●	6.36 ●	4.89 ●	11.46 ●	0.15 ●
etcd-2	4.06 ○	4.45 ●	6.28 ●	66.79 ○	69.07 ○	69.18 ○	100.68 ●	94.73 ●	90.19 ○	29.46 ○
etcd-3	1.25 ●	0.69 ●	1.24 ●	7.15 ●	6.57 ●	9.26 ●	4.95 ●	4.31 ●	9.89 ●	0.14 ●
etcd-4	6.80 ○	6.00 ○	7.34 ○	34.53 ○	34.34 ○	34.37 ○	12.28 ●	12.39 ●	22.92 ●	8.09 ○
etcd-5	43.59 ○	22.46 ○	43.44 ○	27.21 ○	27.86 ○	27.17 ○	30.54 ○	31.40 ○	24.98 ○	23.73 ○

Table 5: Result variability in CV [%] for every combination of benchmark and instance-type in the study. The circles indicate the relative 95%-confidence-interval widths (RCIW) of the CV, computed with statistical simulation (i.e., bootstrap with 1000 simulations). ○ indicates a RCIW below 30%, ● between 30% and 60%, and ● greater than 60%.

etcd-2 on *Azure GP*. Consequently, the potential slowdown to be detected by the benchmarks will also vary drastically depending on the benchmark and instance type it is executed on. We observe three groups of benchmarks: (1) some have a relatively small variability across all providers and instance types (e.g., *rxjava-1*); (2) some show a high variability in any case (e.g., *log4j2-5*); and (3) some are stable on some instance types but unstable on others (e.g., *bleve-5*). The first group’s result indicate that variability is low, as desired for performance testing. However, the latter two groups are particularly interesting for further analysis to identify reasons for their instability.

The second group consists of benchmarks with high variability across all studied instance types. We observe three such benchmarks: *log4j2-1*, *log4j2-5*, and to a lesser extent *etcd-5*. There are two factors that lead to high variability, either the execution time of the benchmark is very short, or the benchmark itself produces unstable results. *log4j2-1* and *log4j2-5* are examples for the first case, with low execution times in the orders of only tens of nanoseconds. For these benchmarks, measurement inaccuracy becomes an important factor for variability. *log4j2-1* is also interesting because this benchmark has been identified as very stable in our preliminary studies. We speculate that for such extremely short-running benchmarks (i.e., 4.7ns on average), small variations in the environment (i.e., our pre-study was on a controlled host in the first author’s university) can have a substantial impact on the observed measurements and their stability. This makes such benchmarks of questionable

use for performance testing. In contrast, *etcd-5* has an execution time around 250000ns on *GCE Mem* with a CV of 27.17%. Figure 2 depicts the results for this benchmark on all 50 instances in beanplot notation. The variability of this benchmark is comparable in *all* instance types, with CVs ranging from 22.46% to 43.59%. Even the bare-metal machine from *Bluemix* has high variability of 23.73% CV. This indicates that the benchmark itself is rather low-quality and produces unstable measurement results, independently of where it is executed.

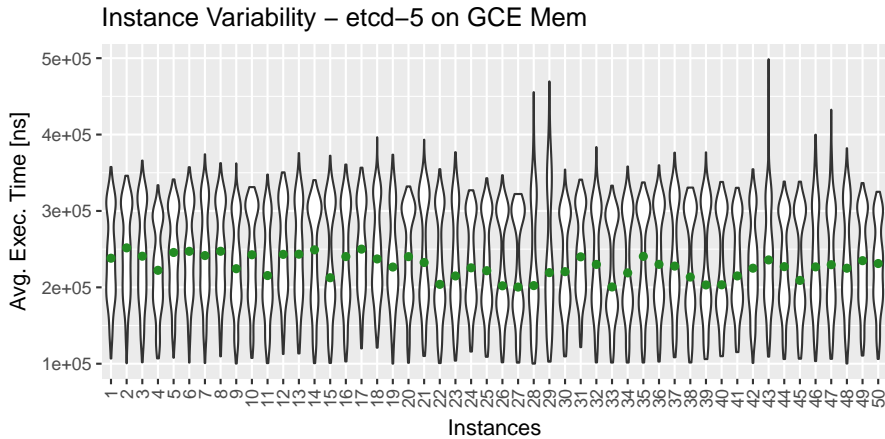


Fig. 2: Drilldown into the variability of *etcd-5*, an example of a benchmark with high result variability across all instance types.

The third group of benchmarks exhibits high variability on some, but not all, instance types. This group contains two sub-groups: (1) benchmarks that have high standard deviations but similar medians; and (2) benchmarks that have overall varying results, including substantially differing medians on different instances. An example for the first sub-group is *log₄j₂-3* on *GCE Mem* — and similarly on the other GCE and Azure instances — where the variability of the benchmark differs among the instances of the same instance types (see Figure 3). We observe that this benchmark on this instance type has a “long tail” distribution, which is common for performance data. However, the length of this long tail differs from instance to instance. A possible explanation for this phenomenon is the behavior of other tenants on the same physical machine as the instance. Other tenants may compete for resources needed by a benchmark causing longer tails in the data. We have observed this problem particularly in the case of the *log₄j₂* benchmarks where manual analysis of these benchmarks reveals that they tend to be IO-intensive (e.g., writing to log files). Previous work has shown that IO-bound operations suffer particularly from noisy neighbors in a cloud (Leitner and Cito 2016).

More severe variabilities can be observed with the second sub-group, where even medians are shifted substantially between instances. This is illustrated in

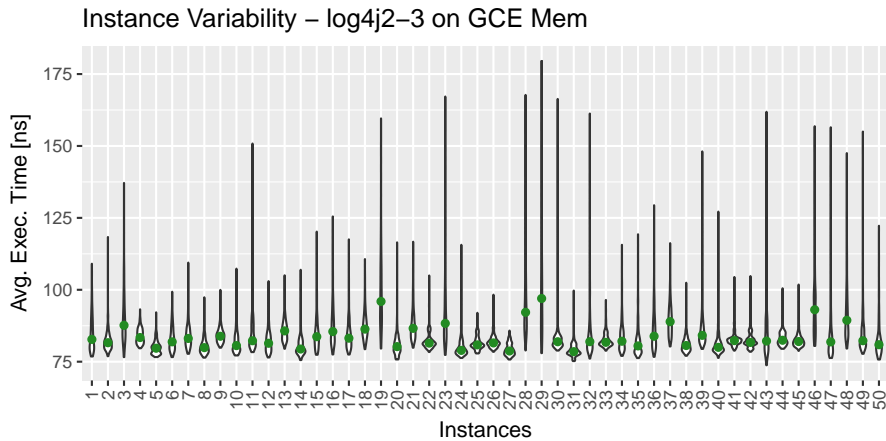


Fig. 3: Drilldown into the variability of $\log_4 j_2-3$, an example of a benchmark with differing variability between instances.

Figure 4 for *bleve-5* on Azure. A potential cause for this phenomenon is hardware heterogeneity (Ou et al 2012; Farley et al 2012) where different hardware configurations (e.g., different processor generations) are used for different instances of the same instance type. Given that the medians fall into a small number of different groups (only 2 in the case of Figure 4), we conclude that hardware heterogeneity rather than multi-tenancy is the culprit for the variability observed in these cases. In this *bleve-5* example on Azure, the hardware metadata supports our conclusion by revealing that two different versions of the same CPU model with distinct CPU clock speeds were provisioned.

Moreover, an interesting finding is that different instance-type families (e.g., general-purpose versus compute-optimized) of the same cloud provider mostly do *not* differ drastically from each other. The only cloud provider that consistently has different variabilities between its instance types is Azure, where the memory-optimized type does not perform as well as the general-purpose type and the compute-optimized type. A reason for the similarity of different instance types of the same provider may be that the different types are backed by the same hardware, just with different CPU and RAM configuration. We assume that the benchmarks under study do not fully utilize the provided hardware and thus show little difference.

4.2 Sources of Variability

We now discuss three different sources of variability in more detail. These sources are (1) variability inherent to a benchmark (“Benchmark Variability”), (2) variability between trials on the same instance (“Variability between Trials”), and (3) variability between different instances of the same type (“Total Variability”). This gives insight into whether variability originates from the

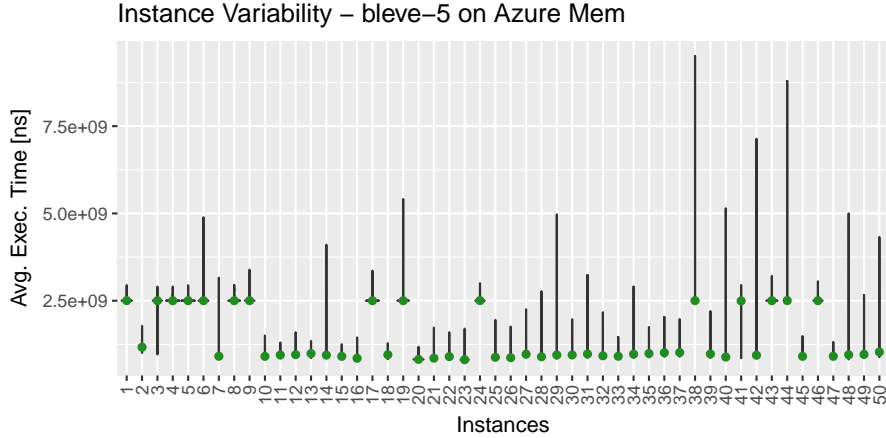


Fig. 4: Drilldown into the variability of *bleve-5* on Azure, an example for a benchmark with high benchmark-result variability due to differences in hardware.

benchmark itself (e.g., non-deterministic behavior of the code written or the runtime environment), from fluctuating performance within a single instance (e.g., due to noisy neighbors), or from different performance behavior of instances of the same type (e.g., hardware heterogeneity). Expectedly, the relative impact of these sources of variability differs for different configurations. Some examples are depicted in Figure 5. Each subfigure contrasts three different CV values: the mean CV per trial of a benchmark (*Benchmark Variability*), the mean CV per instance (*Variability between Trials*), and the total CV of a configuration as also given in Table 5 (*Total Variability*). The red error bars signify the standard deviation. Notice that this is not meaningful for the *Total Variability* that consists of a single CV value per configuration.

Recall the definition of the measurements for a configuration M^c (see Equation 4) and the variability across all measurements $V^{total,c}$ (see Equation 5). In a similar vein, we now define the other two variabilities discussed in this section, i.e., variability inherent to a benchmark $V^{trials,c}$ and variability between trials of an instance $V^{instances,c}$.

Equation 6 specifies the variability inherent to a benchmark $V^{trials,c}$. It is the set of all CVs per trial of a benchmark b executed in an environment e .

$$V^{trials,c} = \bigcup_{M_{i,t}^c \in M^c} cv(M_{i,t}^c) \quad (6)$$

$V^{instances,c}$ (see Equation 7) describes the variability within instances i for all trials t (i.e., $trials = 50$) on this particular instance. It is the set of CVs for each instance i (i.e., $instances = 50$), calculated from all measurements $M_{i,t}^c$ ($trials \times instances = 500$) taken on this instance.

$$V^{instances,c} = \bigcup_{i' \in [1, instances]} cv\left(\bigcup_{M_{i,t}^c \in M^c \wedge i=i'} M_{i,t}^c\right) \quad (7)$$

Figure 5 depicts the variabilities $V^{trials,c}$ (top bar), $V^{instances,c}$ (middle bar), and $V^{total,c}$ (bottom bar), for four configuration examples.

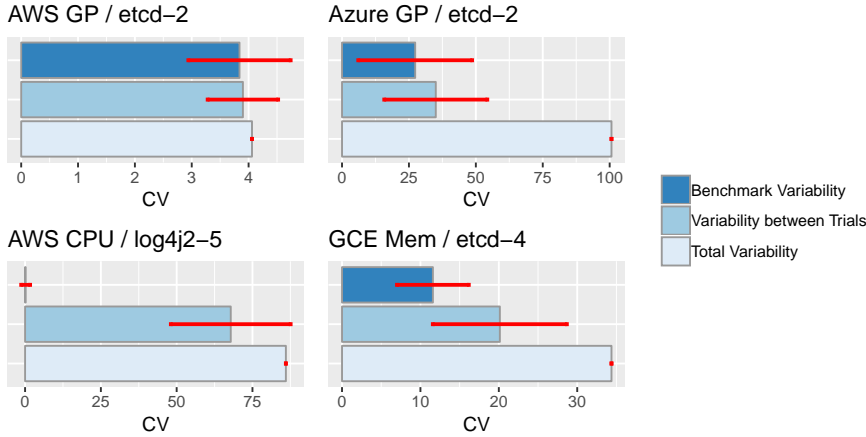


Fig. 5: Drilldown on the source of variability for four example configurations.

The top-left and top-right subfigures provide a drilldown on *etcd-2* in different clouds, and explore in more detail why this curious benchmark is remarkably stable on AWS and unstable in GCE and Azure (see also Table 5). The top-left subfigure shows the benchmark on *AWS GP*. There is little difference in the CVs between trials, instances, and total, indicating that the small variability of this benchmark largely originates from the benchmark itself. This is different for the same benchmark executed on Azure (top-right subfigure). While the per-trial and per-instance CV is also substantially larger than on AWS, it is particularly the total CV that is now very high. This indicates, that *etcd-2* is fairly stable within a trial and among multiple trials within an instance but becomes comparatively much more unstable when considering executions of the benchmark across multiple instances of the same type. Further, this indicates that the reason for the difference in stability between cloud providers is caused by varying instance performance.

A different example is provided by the bottom-left figure, which shows *log4j2-5* on *AWS CPU*. Here, the inherent benchmark variability is minuscule but there are substantial differences between different trials, largely independently of whether these trials happen on the same instance or not. This indicates that for this benchmark a large source of variability are trial-level effects, such as different types of resource contention (e.g., CPU, memory, IO) over the duration of its execution.

Finally, the example on the bottom-right shows *etcd-4* on *GCE Mem*. This benchmark has a high total variability, which is composed of a combination of large inherent benchmark variability and substantial performance variability between different instances. This is the case, where all three variability types have a similarly big fraction of the overall instability of the benchmark-execution's result.

Benchmark Variability: In RQ1, we studied the variability of microbenchmark results when executed on cloud instances. The CVs for the studied benchmark-environment configurations vary between 0.03% to 100.68%. This variability originates from three sources (variability inherent to a benchmark, between trials on the same instance, and between different instances) and different benchmark-environment configurations suffer to very different degrees from any of these sources. The bare-metal instance expectedly produces very stable results, however AWS is typically not substantially less stable. Based on these results, we conclude that GCE and Azure seem to lend themselves much less to performance testing than AWS at the time of study.

5 Reliably Detecting Slowdowns

To answer RQ2, we compare two standard statistical tests for evaluating whether the performance of a software has changed. First we outline two approaches how to run performance experiments and sample data from existing measurements, i.e., instance-based and trial-based sampling strategies. Then we investigate how many FPs the two tests report when executing A/A tests in the studied environments. Intuitively this gives us an indication which benchmarks in which configurations should not be used for performance testing at all, as they frequently indicate slowdowns even if comparing identical software versions. Lastly, we explore the minimal-detectable slowdowns (MDSs) of all benchmarks in all configurations with sufficiently low number of FPs during A/A testing.

5.1 Statistical Tests

For both the A/A tests and the MDS tests (also referred to as A/B tests), we use two standard statistical tests for evaluating whether a software's performance has changed. Literature on performance evaluation suggests usage of two test types: (1) hypothesis testing with Wilcoxon rank-sum (sometimes referred to as Mann-Whitney U test) combined with effect sizes (Bulej et al 2017), and (2) change-detection through testing for overlapping confidence intervals of the mean (Bulej et al 2017; Georges et al 2007; Kalibera and Jones 2012, 2013). Note that Wilcoxon rank-sum is a test for difference of medians while the confidence-interval test employed in this paper is for means.

5.1.1 Wilcoxon Rank-Sum

Firstly, we study the applicability of Wilcoxon rank-sum for performance-change detection. Due to the non-normality (i.e., long-tailed or, in some cases, multi-modal) of performance data, we choose the Wilcoxon rank-sum test because it is applicable to non-normal data. We formulate the null hypothesis H_0 as the two performance result populations (i.e., test and control group) having the same performance (mean execution time), when both groups contain performance results from the same benchmark in the same environment. Consequently, the alternative hypothesis H_1 states that the two compared performance populations do not have the same performance and hence we detect a performance change. All experiments use a 95% confidence level. That is, we report a statistically significant performance change *iff* the p -value of the Wilcoxon rank-sum is smaller or equal to 0.05 (i.e., $p \leq 0.05$). Further, due to the large number of data points, hypothesis tests might report a statistically significant difference with low effect size. Therefore, we combine the Wilcoxon test with a test for a minimum effect size as measured by Cliff’s Delta (Cliff 1996). We test for an effect size of “medium” (0.33) or larger, as defined in Romano et al (2006). Similar to the usage of Wilcoxon rank-sum, we utilize Cliff’s Delta as a measure of effect size due to its applicability to non-normal data. We have also conducted experiments testing for an effect size of small or larger ($|d| \geq 0.147$), but determined that this led to excessive FPs in most tested configurations. Hence, we do not report on these results here. However, the corresponding tables are part of our online appendix (Laaber et al 2019).

5.1.2 Confidence Intervals

Classic textbooks on performance analysis (Jain 1991; John and Eeckhout 2005) suggest that confidence intervals should be preferred over hypothesis testing. Therefore, we additionally study how well slowdowns can be detected by testing for overlapping 95% confidence intervals of the mean. We report a difference between test and control group *iff* the 95% confidence intervals of the mean of the two populations *do not* overlap. Due to the non-normality of performance data, a simple computation of the confidence interval of the mean would be invalid. To address this, we apply statistical simulation, i.e., bootstrapping (C. Davison and Hinkley 1997), with hierarchical random re-sampling (Ren et al 2010) and replacement. In detail, the approach employed is tailored for evaluating performance data and has been introduced by Kalibera and Jones (2012, 2013). Re-sampling happens on three levels: (1) instance-level, (2) trial-level, and (3) iteration-level. We run 100 bootstrap iterations for each confidence interval computation. Although Kalibera and Jones (2013) suggest to run 1000 bootstrap iterations, the additional layer of repeated simulations (100 times re-sampling of test and control group for the upcoming A/A tests and minimal-detectable slowdowns) adds drastically to the overall runtime, i.e., about 130 hours for A/A tests and 500 hours for detection tests. A manual analysis of the computed confidence intervals (for the benchmark

variabilities in Section 4) between 100 and 1000 bootstrap iterations suggested that the gain of using 1000 bootstrap iterations is relatively small. Hesterberg (2015) even suggests adopting 10000 bootstrap simulations as smaller numbers might suffer from Monte Carlo variability. Admitting that individual results might change due to the randomized nature of bootstrapping, the overall results presented in the paper are expected to remain stable. Especially because our experiments (A/A and detection tests) sample 100 different test and control groups (see Section 3) and compute bootstrapped confidence intervals for each of these 100 pairs with 100 bootstrap iterations, which in total reaches the 10000 samples suggested by Hesterberg (2015). We refer to (Kalibera and Jones 2012, p.24ff) and our online appendix (Laaber et al 2019) for details on how the confidence interval is computed.

5.2 Sampling Strategies

We now outline two sampling strategies along the dimensions of the study (i.e., trials and instances), which define how the test and control group of a performance experiment can be executed.

Recall from Sections 3.3 and 4 the formal definitions of the specific measurements $M_{i,t}^c$ of a configuration c for trial t and instance i (see Equation 1) and the definition for all measurements of a defined configuration M^c (see Equation 4). The selection of the test and control group is then defined as *select* in Equation 8.

$$select : \langle M^c, sel^{instances}, sel^{trials} \rangle \mapsto \langle M'^c, M''^c \rangle \quad (8)$$

select takes as input the measurements of a configuration M^c , the desired number of instances $sel^{instances}$ to sample, and the desired number of trials sel^{trials} to sample. The function returns a tuple of sampled configuration results $M'^c \subset M^c$ where the first element corresponds to the test group and the second element to the control group. Concrete implementations of *select* are the sampling strategies *instance-based sampling (ibs)* and *trial-based sampling (tbs)*, which have the same signature and are described in the following.

5.2.1 Instance-based Sampling

Instance-based sampling implements the idea of running test and control groups on *different* instances. In practical terms, this emulates the situation when a performance engineer wants to compare a new version of a system against older performance data, which has been measured at a previous time, for instance when the previous version was released. We assume that, between releases, cloud instances are terminated to save costs.

Figure 6a visualizes the instance-based sampling strategy, and Equation 9 formally defines the function *ibs* that performs this selection. It randomly selects (potentially) multiple *different* instances for each test and control group

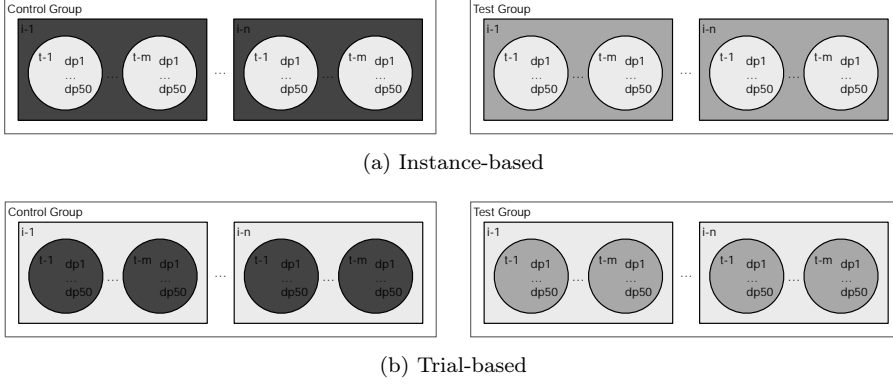


Fig. 6: Sampling strategies. (a) Instance-based: test and control group consist of disjoint instances (blue/green, b/w: dark/medium grey) with m trials each (light grey). (b) Trial-based: test and control group consist of the same instances (light grey), with disjoint m trials each taken from the same instances (blue/green, b/w: dark/medium grey).

(in blue and green; $sel^{instances}$) and uses random trials from each instance (in light grey; sel^{trials}).

$$\begin{aligned}
 ibs(M^c, sel^{instances}, sel^{trials}) = \langle \\
 \{M_{i,t}^c \in M^c \mid i \in I^{TG} \wedge t \in rand(T^\#, sel^{trials})\}, \\
 \{M_{i,t}^c \in M^c \mid i \in I^{CG} \wedge t \in rand(T^\#, sel^{trials})\} \rangle
 \end{aligned} \quad (9)$$

For Equation 9 the conditions in Equation 10 hold. The first clause defines the test group's instances I^{TG} of size $sel^{instances}$, which are randomly selected from $I^\# = [1, instances]$ with $instances$ being the total number of instances (i.e., $instances = 50$ as defined in Section 3.3). The second clause, similar to the first one, describes the control group's instances I^{CG} of size sel^{trials} , which have the same selection criterion as the test group's instances. Finally, the third clause ensures that the sets of instances for test and control group are disjoint (i.e., that we are not using the same instances for test and control group). Supporting the above and following equations, the function $rand$ takes as input an arbitrary set and a set-member counter and returns the counter's value of randomly selected members of the set.

$$\begin{aligned}
 I^{TG} &= rand(I^\#, sel^{instances}) \wedge \\
 I^{CG} &= rand(I^\#, sel^{instances}) \wedge \\
 I^{TG} \cap I^{CG} &= \emptyset
 \end{aligned} \quad (10)$$

5.2.2 Trial-based Sampling

Trial-based sampling implements the idea of running test and control groups on the *same* instances, as opposed to instance-based sampling where they are executed on *different* instances. This emulates the case where a performance engineer starts multiple instances and then runs both, the test and control group, potentially multiple times on the same instance in randomized order. This minimizes the impact of the specific instance's performance, which we have established to be an important factor contributing to variability in many cases. This approach can also be seen as a paired statistical test. Hence, we expect that this approach should generally lead to fewer FPs and smaller MDSs.

Figure 6b illustrates and Equation 11 formalizes trial-based sampling, which randomly selects (potentially) multiple instances ($sel^{instances}$) per benchmark and environment. However, different from the instance-based strategy, the control and test group now consist of the same instances (in light grey; $sel^{instances}$) with multiple different randomly selected trials each (in blue and green; sel^{trials}).

$$\begin{aligned}
 tbs(M^c, sel^{instances}, sel^{trials}) = \langle & \\
 \{M_{i,t}^c \in M^c \mid i \in I' \wedge t \in T_i^{TG}\}, & \\
 \{M_{i,t}^c \in M^c \mid i \in I' \wedge t \in T_i^{CG}\} \rangle & \quad (11)
 \end{aligned}$$

Equation 12 describes the conditions that hold for Equation 11. The first clause defines the test and control group's instances I' , which are randomly selected from $I^\# = [1, instances]$ with $instances$ being the total number of instances (i.e., $instances = 50$ as defined in Section 3.3). The second clause describes the trials selected from each instance for the test group T_i^{TG} , which have a size of sel^{trials} (e.g., 5), and are randomly selected for each instance i from $T^\# = [1, trials]$ with $trials$ being the total number of trials as defined in our methodology as $trials = 10$ (cf. Section 3.3). The third clause, similar to the second one, describes the control group's trials T_i^{CG} , which have the same selection criterion as the test group's trials. The fourth clause ensures that the sets of trials for test and control group are disjoint.

$$\begin{aligned}
 \forall i : I' = rand(T^\#, sel^{instances}) \wedge & \\
 T_i^{TG} = rand(T^\#, sel^{trials}) \wedge & \\
 T_i^{CG} = rand(T^\#, sel^{trials}) \wedge & \\
 T_i^{TG} \cap T_i^{CG} = \emptyset & \quad (12)
 \end{aligned}$$

5.3 A/A Testing

In this section, we perform A/A tests of all selected benchmarks with different sample sizes and the two sampling strategies ($select$). The goal of A/A

testing is to compare samples that, by construction, *do not* stem from a differently performing application (i.e., running the same benchmark in the same environment). Thus, the goal of these tests is to validate that the experiment environment does not report a slowdown if, by construction, a slowdown is not possible. Following common statistical practice, we define the upper FP threshold to be 5% and consider a specific benchmark in a specific environment to be too unreliable if it exceeds the 5% FPs threshold. That is, from 100 random tests between identical versions, we hope for 5 or less FPs.

Recall the approach from Section 3 depicted in Figure 1: we executed every benchmark on each individual instance 10 times ($trials = 10$), and repeated these trials 50 times on different instances of the same instance type ($instances = 50$). We now randomly select 1, 2, 3, 5, 10, 15, and 20 instances ($sel^{instances}$), and then randomly select 1, 2, 3, and 5 trials (sel^{trials}) for test and control group. As selection strategy ($select$) we use both instance-based (ibs) and trial-based (tbs) sampling. For each benchmark-environment configuration (e.g., $log4j2-1$ in *AWS GP*), we repeat this experiment 100 times with different randomly-sampled instances and trials for test and control groups. To account for the increased likelihood of rejecting H_0 when testing 100 times a distinct benchmark-environment configuration with Wilcoxon rank-sum, we apply a Bonferoni correction to the resulting p-values of the 100 test-control-group samples.

5.3.1 Example

Table 6 shows an example of the FP-rates for *bleve-3*. The first four rows use ibs whereas the last four rows use tbs . Intuitively, the table reports in percent how often a confidence-interval test falsely reported a performance change when neither benchmark nor code had in fact changed.

	# Ins.	# Trials	AWS			GCE			Azure			BM
			GP	CPU	Mem	GP	CPU	Mem	GP	CPU	Mem	
ibs	1	1	81	81	90	82	88	92	86	87	71	43
	1	5	69	35	85	71	73	69	81	72	65	13
	10	1	6	7	4	6	5	6	6	7	1	6
	20	5	3	2	0	4	4	2	5	4	1	0
tbs	1	1	49	67	64	67	65	65	27	41	26	38
	1	5	11	13	5	6	8	10	3	6	8	2
	3	1	3	2	0	4	4	5	0	2	3	8
	3	5	0	0	0	1	0	0	0	0	0	5

Table 6: FP-rates in percent testing for overlapping confidence intervals for *bleve-3* for instance-based (ibs , top) and trial-based (tbs , bottom) sampling, across all studied environments, with example configurations.

The smallest sample sizes (i.e., one instance and one trial) for both sampling strategies show a high number of FPs across all environments, with the

bare-metal one from Bluemix (column “BM”) being the “most reliable” environment with approximately 40% FPs. Increasing the number of trials and/or instances yields fewer FPs for both strategies. *ibs* does not achieve much better results when only considering more trials but still only one instance for both test and control group (row 2). However, increasing the number of instances to 10 (row 3) results in less than 10% FPs for this particular benchmark. Increasing the sample size even further to 20 instances and 5 trials per instance, the instance-based strategy has at most 5% FPs across all studied environments, which we consider acceptable. In comparison when running test and control group on the same instances (*tbs*), already an increase to five trials on a single instance produces much fewer FPs (row 6). A small sample size such as three instances with five trials already reduces to FP-rate of all but one cloud environments to 0. Interestingly, we still observe 5% FPs in the Bluemix environment, despite a very low CV. In this particular case (i.e., for benchmark *bleve-3*), the confidence intervals are extremely narrow and therefore already minor shifts of the mean in the control group are statistically significant changes.

We omit detailed results for other benchmarks and the Wilcoxon rank-sum testing because they are generally in line with the example provided above. Full details for every benchmark, both sampling strategies, and both statistical tests can be found in the online appendix (Laaber et al 2019).

5.3.2 Impact of Sampling Strategy

We now provide a more detailed look into the impact of the different sampling strategies and chosen number of samples. Figure 7 shows the density plots of FPs rates for all studied environments. The left subfigures (7a) show the Wilcoxon rank-sum test and the right subfigures (7b) show the results for the confidence-interval test. The first two subplots are instance-based sampling results with one instance and one trial and ten instances and one trial respectively. The lower three subplots depict trial-based sampling results with two instances and five trials, five instances and two trials, and five instances and five trials respectively. The blue horizontal line indicates the 5%-FP threshold, which we consider as acceptable upper bound across the 100 simulations.

These figures make obvious that for instance-based sampling, even with 10 instances, testing using Wilcoxon leads to a large number of FPs for most benchmark-environment configurations (intuitively, most of the area under the curve is on the right-hand side of the 5% border). The smallest sample size (one instance with one trial using *ibs*; top row) produces FP-rates $> 5\%$ for almost all configurations and only a few are below the 5% threshold: 18/190 for Wilcoxon and 8/190 for confidence intervals. This improves when considering 10 instances with one trial and *ibs* (second row). Across all configurations only 44 in 190 have 5% FPs or less, 48 have between 10% and 20%, and still 54 have more than 20% FPs when using Wilcoxon. If using only a single trial from each instance for test and control group (e.g., if a performance engineer were to run the benchmarks only once for each release) the rate of reported

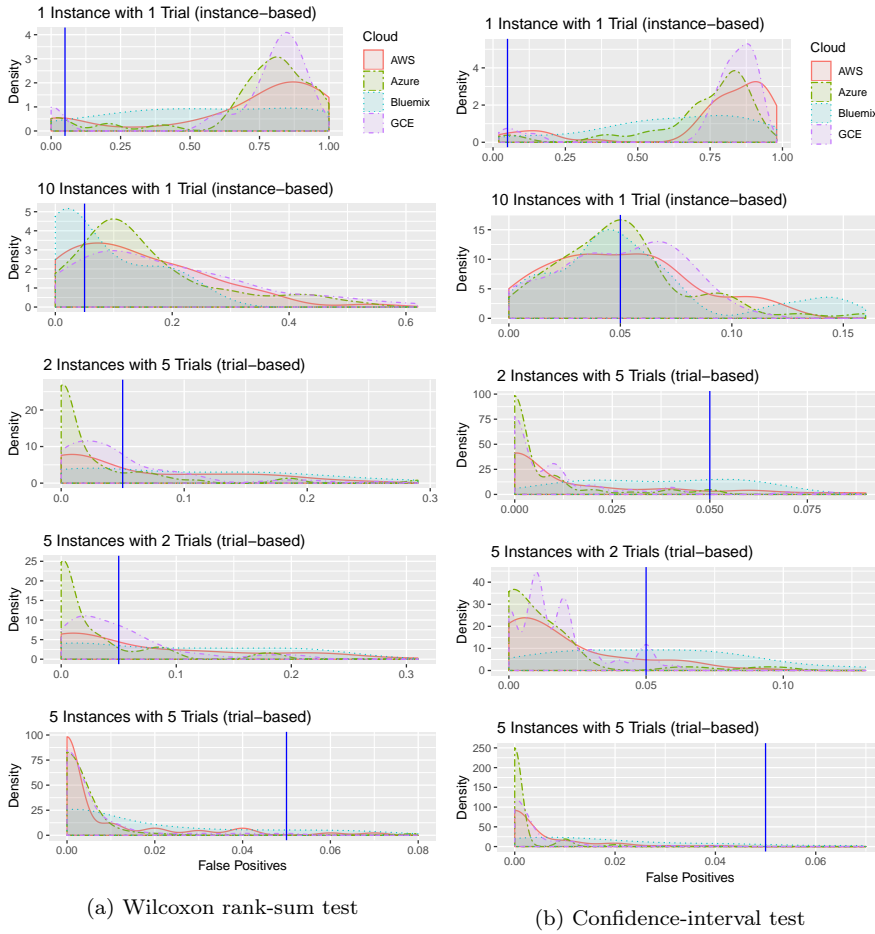


Fig. 7: False-positive differences between environments for Wilcoxon rank-sum and confidence-interval tests

FPs sometimes even exceeds 50% for 4 configurations. Using overlapping confidence intervals and 10 instances, the situation improves drastically (110 of 190 configurations, or close to 60%, have $\leq 5\%$ FPs), although still 80 configurations exhibit too many FPs. 11 configurations have a FP rate of 10% or more, but none has 20% or more.

Changing to trial-based sampling improves the overall FP rate across all environments. Both statistical tests show below 5% FPs for almost all configurations (185 of 190, or 97% with Wilcoxon, and 189 of 190 with confidence intervals) when increasing the sample size to five instances with five trials. Generally, confidence intervals produce slightly fewer total FPs than Wilcoxon rank-sum. From a total of 19000 simulated tests (19 benchmarks * 10 environ-

ments * 100 simulations), only 75 (0.4%) and 106 (0.6%) FPs were reported when testing with confidence intervals and Wilcoxon respectively.

Comparing the different cloud providers shows interesting results. Studying the worst case (one instance with one trial using *ibs*, top row) and the best case (five instance with with trials using *tbs*, bottom row), we can hardly notice any differences among the cloud providers with respect to their FP rates. For the worst case, only 2 (Azure, Bluemix) to 3 (AWS, GCE) benchmarks have < 5% FPs in all environments. In the best case, 0 (AWS, GCE, Azure) or 1 (Bluemix) benchmarks are *not* below this threshold. Counter-intuitively, we observe that the more stable cloud environments (AWS and Bluemix, based on results from Section 4) lead to more FPs in the other three examples (rows two, three, and four). This is because environments with high dispersion between individual measurements often lead to the Wilcoxon test not being able to reject H_0 as well as wide confidence intervals, which incidentally helps to reduce the number of FPs. However, we expect that the number of accurately identified actual slowdowns (see Section 5.4) will be low as well in the more unstable environments. That is, these results should not be taken to mean that an unstable environment is more useful for performance testing than a stable one, just that a specific type of problem (namely a falsely identified slowdown) is less likely to happen. Another interesting observation is that the IaaS environments (AWS, GCE, and Azure), and in particular the more unreliable environments (GCE and Azure), benefit more from employing *tbs* rather than *ibs*. This indicates that the methodology RMIT introduced by Abedi and Brecht (2017) can indeed be considered a best-practice.

We observe the same trends for the different projects under study. Trial-based sampling outperforms instance-based sampling and confidence intervals produce slightly more reliable results than Wilcoxon rank-sum tests. An investigation of the FP rate differences between instance families (i.e., GP, CPU, Mem) was inconclusive. We did not see substantial differences between instance families of the same cloud provider in the density plots, nor did an examination of the detailed per-benchmark results reveal major differences.

5.3.3 Minimal Number of Required Samples

In the examples above, we see that an increase of samples (instances and/or trials) as well as the sampling strategy has a direct impact on the reliability of the obtained results, i.e., the number of wrongly reported performance changes (FPs). Figure 8 shows the minimal number of required samples (number of instances and trials) for the statistical tests to report below 5% FPs. For every configuration (benchmark-environment combination), we report a single dot that indicates the lowest sampling which has below 5% FPs. Intuitively, the further to the bottom-left corner of each plot, the fewer samples are required to *not* suffer from excessively many FPs and therefore having a sufficiently stable environment for slowdown detection. The lowest sample number is defined by the product of number of instances and number of trials. As this can result in the same number of samples (e.g., one instance and five trials vs. five instances

and one trial), we first check more trials of the same instances, because an increase of trials compared to an increase of instances tends to have a smaller impact on the FP rate.

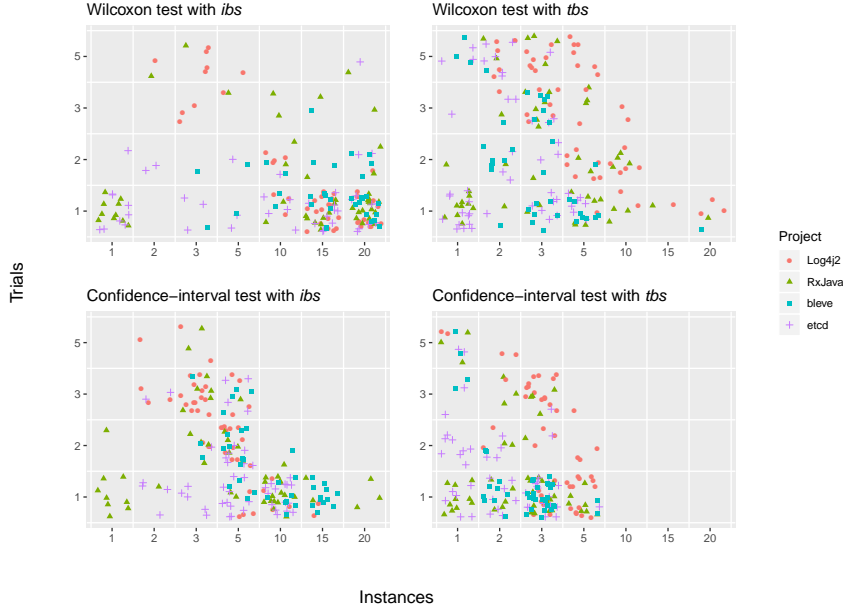


Fig. 8: Minimally required samples to reach $<5\%$ false-positives. The top row shows the Wilcoxon rank-sum test, and the bottom row depicts confidence intervals. The left column shows instance-based and the right column shows trial-based sampling. Note that this is a jitter plot for discrete values, therefore all data points within one box correspond to the same value.

For both tests (Wilcoxon test in the first row and confidence-interval test in the second row) and sampling strategies (instance-based left in the left column and trial-based in the right) one instance and one trial is sufficient only for a small number of benchmarks. These configurations are exclusively from the *RxJava* and *etcd* projects. For instance-based (left column), just an increase in number of trials does not produce better results. The majority of the results analyzed with Wilcoxon require repeating the measurements on 10 instances or more to reach $<5\%$ FPs. A few configurations only drop below that threshold with two or more trials for each instance. In comparison, confidence intervals (bottom-left plot) require between five and 15 instances when using one trial. Interestingly, especially *Log4j2* configurations benefit more from an increased number of trials than the other projects when using confidence intervals. Considering all studied benchmarks, an upper bound of minimally required samples for *ibs* is 5000 (20 instances * 5 trials * 50 iterations) when

testing with Wilcoxon and 1000 (20 instances * 1 trial * 50 iterations) when employing the confidence-interval test.

For trial-based sampling (right column) we see a similar trend, where confidence intervals require fewer instances/trials than testing with Wilcoxon. Even more evident, *Log4j2* turns out to be the project with the highest FP rate. As expected by the design of trial-based sampling (running test and control group on the same instances), an increase in number of trials often reduces the FP rate. However, a surprising result is that many benchmark-environment combinations already yield low FP rates for a single trial on two to five instances when testing with confidence intervals. Over all benchmarks, at most 1500 samples (10 instances * 3 trials * 50 iterations) for Wilcoxon and 750 samples (5 instances * 3 trials * 50 iterations) for the confidence-interval test are required for a maximum of 5% FPs when employing *tbs*.

Our A/A testing results show that for the microbenchmarks under study the overlapping confidence-interval test (bottom row) indeed yields better results than the Wilcoxon rank-sum test (top row). That is, less samples are required to have similarly stable results. This result is in line with previous scientific arguments (Jain 1991; Kalibera and Jones 2013).

A/A Testing: Identical measurements (e.g., same benchmark executed in the same environment without any code changes) suffer from falsely-reported performance changes when they are taken from a small number of instances and trials. Nonetheless, an increase in samples yields a low number of acceptable FPs (i.e., <5%) for all studied benchmarks and environments, both sampling-strategies, and both statistical tests. Hence, testing for performance with software microbenchmarks in (unreliable) cloud environments is possible. However, a substantial number of experiment repetitions are required, optimally executed using a trial-based strategy. We have confirmed that testing using overlapping confidence intervals yields substantially less false positives with the same setup than using a Wilcoxon test with medium Cliff's delta effect size.

5.4 Minimal-Detectable Slowdown Sizes

The previous section showed that depending on the sample size (number of instances and trials), we are able to utilize cloud instances for microbenchmark execution with low FP-rates when performing A/A tests. Especially when employing trial-based sampling (running test and control group on the same instances), already using five instances with five trials produces results that have below 5% FPs for most studied benchmarks and environments. Taking a step further, we now investigate the sizes of slowdowns that are detectable with both sampling strategies (*ibs* and *tbs*) using both studied statistical tests (Wilcoxon rank-sum with Cliff's Delta effect-size measure of medium or larger and overlapping 95% confidence intervals for the mean computed using bootstrapping).

5.4.1 Approach

To find the *minimal-detectable slowdown (MDS)*, i.e., the slowdown that can be identified in 95% of cases, with at most 5% of FPs, by a benchmark in a particular environment, we performed experiments based on the following procedure. For each benchmark and each instance type, we investigate a fixed list of 11 simulated slowdowns ranging from a tiny 0.1% slowdown to a massive one of 1000% (i.e., the respective functionality got 10 times slower). Concretely, we experiment with the following slowdowns: 0.1%, 0.2%, 0.5%, 1%, 1.5%, 2%, 5%, 10%, 50%, 100%, 1000%. For each simulated slowdown, we use both sampling strategies (*ibs* and *tbs*) to sample a test and control group from the data set. We simulate a slowdown in the test group by increasing the execution time of each data point by $x\%$ where x is the simulated slowdown size (e.g., for a slowdown of 1%, we add 1% to each data point) and compare test and control group with both statistical tests, i.e., Wilcoxon and confidence-interval tests. We believe this is the most-straightforward way of injecting slowdowns that comes with the fewest parameters to be applied correctly. Note that increasing each data point of the test group by a fixed slowdown percentage represents an “idealized” slowdown. As our experiments share the same data, the comparison between the studied approaches and statistical tests remains valid, although it might overestimate the number of detected slowdowns.

Similar to the A/A tests, for each slowdown we repeat this process 100 times by re-sampling different control and test groups and count how often we find a statistically significant change — a true positive (TP) — either by rejecting H_0 with the Wilcoxon test and an effect size of medium or larger, or by non-overlapping 95% confidence intervals. Again, for the Wilcoxon rank-sum test and every benchmark environment combination, we applied a Bonferoni correction to the resulting p-values. If the rate of TPs is at least 95% (i.e., a minimum of 95 in the 100 repetitions) and the rate of FPs (as identified by the A/A tests from Section 5.3) is not higher than 5% (i.e., a maximum of 5 in the 100 repetitions), we conclude that a slowdown of this size can be found reliably using one of the two studied statistical tests. MDS is then the *smallest* slowdown from the list that can be found with this approach.

In the following, we discuss the MDSs using the instance-based (*ibs*) and trial-based (*tbs*) strategies. Due to the runtime of our simulations, we were not able to run the detection experiments for the same number of selected samples as we did for the A/A tests. The runtime is especially a limiting factor for the confidence interval simulations. We randomly sample 100 test and control groups and for each of these groups we run a bootstrap technique with another 100 iterations to compute the confidence intervals of the mean. In our implementation, this simulation takes around 500 hours for all benchmarks and environments, when executed on 8 core machines. We used a cluster of 10 instances in a private cloud in the first author’s university to execute the experiments.

5.4.2 Instance-based Sampling

We evaluate the instance-based sampling strategy for different sample sizes. Concrete parameters for the sampling function *ibs* are $sel^{instances} \in \{1, 2, 3, 5, 10, 15, 20\}$ and $sel^{trials} = 1$.

Table 7 shows exemplary results for three benchmarks (i.e., *log4j2-3*, *bleve-3*, and *etcd-2*) and a subset of the tested sample sizes (i.e., 10, 20 instances). We omit smaller sample sizes from the example as they hardly detect slowdowns reliably using *ibs*. We provide the full results as part of the online appendix (Laaber et al 2019). Cell values represent the MDS in percent that could be detected in this setup. Cells with the value “ ∞ ” (colored in red), indicate that for the given configuration no slowdown could be detected using our approach.

Stat.	Bench	# Ins.	# Trials	AWS			GCE			Azure			BM
				GP	CPU	Mem	GP	CPU	Mem	GP	CPU	Mem	
WRS test	log4j2-3	10	1	∞	∞	1.5	∞	∞	∞	10	∞	∞	∞
	log4j2-3	20	1	∞	2	1.5	10	∞	10	5	5	5	∞
	bleve-3	10	1	∞	∞	∞	∞	∞	∞	∞	∞	∞	0.2
	bleve-3	20	1	0.5	1	5	∞	5	10	∞	∞	∞	0.2
	etcd-2	10	1	1	2	5	∞	∞	∞	∞	∞	∞	50
	etcd-2	20	1	1.5	2	5	50	50	100	∞	∞	∞	50
CI test	log4j2-3	10	1	5	∞	5	50	50	50	50	10	50	10
	log4j2-3	20	1	5	∞	5	10	10	10	10	∞	10	5
	bleve-3	10	1	∞	∞	10	∞	50	∞	∞	∞	50	∞
	bleve-3	20	1	2	2	5	50	50	50	10	10	10	0.2
	etcd-2	10	1	5	2	5	∞	∞	1000	1000	1000	1000	50
	etcd-2	20	1	2	2	2	100	1000	1000	1000	1000	1000	50

Table 7: MDSs for *log4j2-3*, *bleve-3*, and *etcd-2* using instance-based sampling (*ibs*) with both statistical tests, Wilcoxon test (“WRS test”) and confidence-interval test (“CI test”).

The results depict that, using *ibs*, a high number of samples is required to reliably find slowdowns for many configurations (benchmark-environment combinations). A clear trend is that benchmarks run on AWS are able to detect smaller slowdowns as compared to GCE, Azure, and even Bluemix (BM). This trend also applies to the other benchmarks not displayed in the example.

Although the example might suggest that some instance families (i.e., *AWS Mem*) yield smaller MDS than the other instance families of the same provider, an investigation of the full data set did not reveal substantial differences. In the case of AWS and tested with confidence intervals, all three instance types had roughly the same number of undetectable slowdowns over all samples (total of 7 sample sizes * 19 benchmarks), i.e., 85 (64%; *AWS GP*), 89 (67%; *AWS CPU*), 80 (60%; *AWS Mem*). In terms of MDS, the same instance types find slowdowns of $\leq 10\%$ in 35 (26%; *AWS Std*), 36 (27%; *AWS CPU*), and

41 (31%; *AWS Mem*) of the cases. Generally speaking, the MDS differences between the instance types of the same provider are negligible for the tested benchmarks and environments.

Figure 9 shows an overview histogram over all configurations using *ibs* for testing with Wilcoxon (left) and confidence intervals (right). The bars indicate how many configurations (y-axis) had a MDS of size x on the x-axis. We can see that the majority of configurations do not find any slowdown reliably (“Inf”). This is the case for benchmarks with A/A FPs of more than 5%, or for benchmarks where we could not find the introduced slowdown in 95% or more of the simulations for any slowdown size.

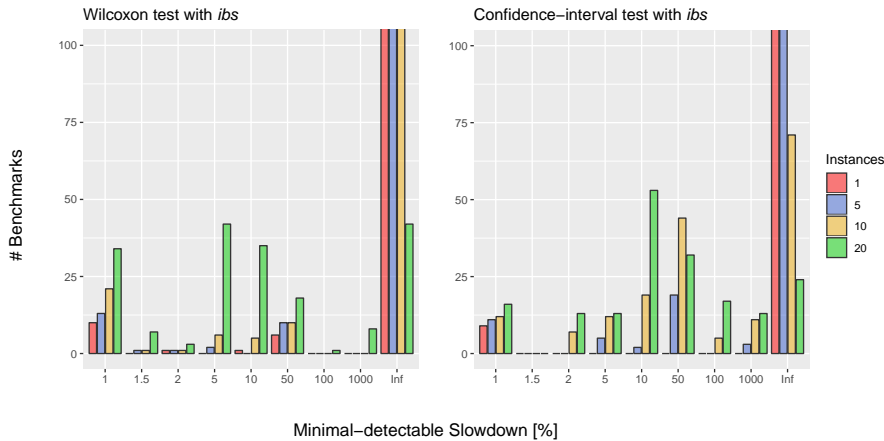


Fig. 9: MDSs found with instance-based sampling (*ibs*) by applying the Wilcoxon test (left) and confidence-interval test (right). The x-axis shows the slowdowns we tested for, and the y-axis shows for how many benchmark-environment combinations the corresponding slowdown is the smallest that is detectable. MDSs $\leq 5\%$ are aggregated at 1. If no slowdown is detectable, *Inf* is depicted.

From the 190 configurations (19 benchmarks * 10 environments), no slowdown could be reliably detected for 172 (91%) benchmarks-environment combinations with one instance, for 162 (85%) combinations with five instances, for 146 (77%) combinations with ten instances, and 42 (22%) combinations with 20 instances when testing with Wilcoxon. The confidence-interval test performs slightly better. 154 (81%) combinations with one instance do not find a slowdown of any size, 141 (74%) combinations with five instances, 71 (37%) with ten instances, and 24 (13%) with 20 instances.

When increasing the sample size to five instances, Wilcoxon finds slowdowns in 27 (14%) configurations, out of which 17 (9%) are below 10% slowdown size. In comparison, the confidence-interval test finds slowdowns in 40 (21%) of the studied configurations, where 18 (9%) are below 10% slowdown

size. Doubling the sample size to ten instances has a small effect when testing with Wilcoxon, i.e., 44 (23%) configurations find a slowdown reliably out of which 34 (18%) detect slowdowns $\leq 10\%$. Going to ten instances has a substantially larger impact on which slowdowns are reliably detectable when testing with confidence intervals. 110 (58%) configurations find a slowdown and 50 (26%) are even able to detect slowdowns $\leq 10\%$. With 20 instances Wilcoxon is able to detect slowdowns in 148 (78%) configurations, and confidence intervals expose slowdowns in 157 (83%) configurations; while 121 (64%) and 95 (50%) reliably report slowdowns below 10% respectively.

These results show that instance-based sampling, i.e., the sampling strategy where test and control group are executed on different instances, is only able to reliably detect slowdowns (of any size) in 78% of the tested configurations when using Wilcoxon and 83% when testing with confidence intervals, even when utilizing a fairly large number of 20 instances.

5.4.3 Trial-based Sampling

Following the results from the instance-based MDSs, we now investigate whether trial-based sampling (*tbs*) is able to detect slowdowns for smaller sample sizes. In Section 5.3, we observed that *tbs*, i.e., running test and control group on the same instances in interleaved order, results in fewer FPs with smaller sample sizes than *ibs*. Based on this, we expect that *tbs* yields more reliable and more stable detection results and therefore smaller MDSs.

We evaluate the trial-based sampling strategy for different sample sizes analogous to the instance-based sampling. The parameters for the sampling function *tbs* are $sel^{instances} \in \{1, 2, 3, 5, 10, 15, 20\}$ and $sel^{trials} = 5$.

Table 8 shows the same examples as above, but now for *tbs* instead of *ibs*. Again, all other detailed results are provided as part of the online appendix (Laaber et al 2019). The three benchmark examples provided already give an indication that *tbs* indeed leads to more reliable results in terms of slowdown detectability. Although running test and control group on a single instance, slowdowns are reliably detectable for benchmarks *bleve-3* and *etcd-2* in some environments with both statistical tests (e.g., *bleve-3* on Azure with the Wilcoxon test). Similar to *ibs*, we see a trend that in AWS and Bluemix, smaller slowdowns can be detected than in GCE and Azure. We again observe that AWS works even better than Bluemix for at least some benchmarks (notably *etcd-2*, but this is not true universally. Regarding the differences between instance types of the same cloud provider we again observe no notable difference. This is in line with the variability results from Section 4, which indicate that these providers appear to be particularly useful for performance microbenchmarking experiments.

In line with the presentation of the instance-based sampling results, we now depict an overview of all configurations in Figure 10 as histogram. The results confirm our intuition from the examples in Table 8. The majority of configurations reliably find at least some slowdown when considering sample sizes bigger than one instance. Nevertheless from 190 configurations (19

Stat.	Bench	Ins. #	Trials #	AWS			GCE			Azure			BM
				GP	CPU	Mem	GP	CPU	Mem	GP	CPU	Mem	
WRS test	log4j2-3	1	5	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
	log4j2-3	5	5	5	∞	1.5	10	5	10	5	5	10	1.5
	log4j2-3	10	5	2	5	1.5	10	5	5	5	5	10	1.5
	log4j2-3	20	5	2	5	1.5	5	5	5	5	5	10	1.5
	bleve-3	1	5	∞	∞	∞	∞	∞	∞	5	5	5	∞
	bleve-3	5	5	1	0.5	2	5	10	10	5	5	10	0.2
	bleve-3	10	5	1	0.5	2	5	5	5	5	5	10	0.2
	bleve-3	20	5	0.5	0.5	2	5	5	5	5	5	10	0.2
	etcd-2	1	5	5	1.5	5	∞	∞	∞	50	50	50	50
	etcd-2	5	5	2	1.5	2	100	50	100	50	50	50	50
	etcd-2	10	5	2	1.5	2	50	50	50	50	50	50	50
	etcd-2	20	5	1.5	1	2	50	50	50	50	50	50	50
CI test	log4j2-3	1	5	∞	∞	∞	∞	∞	50	50	∞	∞	∞
	log4j2-3	5	5	5	10	5	50	50	50	50	10	50	5
	log4j2-3	10	5	10	5	5	50	10	50	50	10	10	5
	log4j2-3	20	5	5	5	2	10	10	10	10	5	10	5
	bleve-3	1	5	∞	∞	5	∞	∞	∞	10	∞	∞	1
	bleve-3	5	5	5	5	10	50	50	50	50	10	50	0.2
	bleve-3	10	5	2	5	50	50	50	50	10	10	10	0.2
	bleve-3	20	5	2	5	10	10	10	10	10	5	10	0.2
	etcd-2	1	5	5	2	5	∞	1000	∞	100	∞	1000	∞
	etcd-2	5	5	2	5	5	1000	1000	1000	1000	1000	1000	50
	etcd-2	10	5	2	2	2	100	100	100	1000	1000	1000	50
	etcd-2	20	5	2	2	2	100	100	100	1000	1000	1000	10

Table 8: MDSs for *log4j2-3*, *bleve-3*, and *etcd-2* using trial-based sampling (*tbs*) with both statistical tests, Wilcoxon test (“WRS test”) and confidence-interval test (“CI test”).

benchmarks * 10 environments), still 142 (75%) with the Wilcoxon test and 117 (62%) with the confidence-interval test are not able to find any of the slowdowns we tested for with a single instance. Increasing the sample size to five instances (and beyond), changes the MDSs drastically for the better. Five instances already allow finding slowdowns in 185 (97%) and 189 (99%) configurations when tested with the Wilcoxon test and confidence-interval test respectively.

Interestingly and contrary to the A/A-testing results, the Wilcoxon test supports finding smaller MDSs. With five instances already 150 (79%) configurations find slowdowns of 10% or smaller when tested with the Wilcoxon test whereas only 74 (39%) when the confidence-interval test is applied. Similar trends are observable for 10 instances, where 156 (82%; Wilcoxon) and 113 (60%; confidence interval), and 20 instances, where 157 (83%; Wilcoxon) and 147 (77%; confidence interval), find a slowdown of 10% or less. 34 (17%; Wilcoxon) and 43 (22%; confidence interval) configurations reliably detect a slowdown of 10% or less when *only* using a single instance. These configurations are spread across all studied environments but only include 12 unique benchmarks from *RxJava*, *bleve*, and *etcd*. We observe that the Wilcoxon test is more aggressive in rejecting H_0 . The leads to more FPs in the A/A tests,

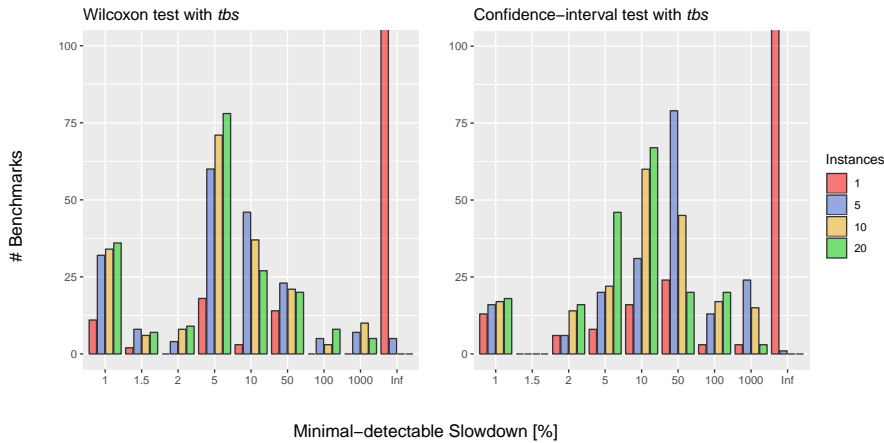


Fig. 10: MDSs found with trial-based sampling by applying the Wilcoxon test (left) and confidence-interval test (right). The x-axis shows the slowdowns we tested for, and the y-axis shows for how many combinations the corresponding slowdown is the smallest that is detectable. MDSs $\leq 5\%$ are aggregated at 1. If no slowdown is detectable, *Inf* is depicted.

but ultimately means that smaller slowdowns are correctly observed in the present experiments.

The *tbs* results confirm the previous result by Abedi and Brecht (2017): executing performance experiments on the same instances in randomized order can indeed be considered a best practice. However, it is a surprising outcome how small slowdowns ($<10\%$) can often be found with high confidence even in a comparatively unstable environment.

Minimal-Detectable Slowdowns: Our experiments re-confirm that testing in a trial-based fashion leads to quite substantially better results than executing on different instances. If using instance-based sampling, a large number of instances (the maximum of 20 in our experiments) is required, otherwise our test benchmarks are not able to discover *any* regression for most benchmark-environment combinations. However, even with 20 instances, slowdowns need to be in the range of 5% to 10% to be reliably detectable in most cases. With trial-based sampling, already 5 instances are sufficient to find similar slowdowns (at least when using Wilcoxon, which has shown to be more aggressive in rejecting H_0 than our alternative confidence interval based approach). A small number of benchmarks in our study is able to reliably identify small changes in performance (less than 5%), but for most combinations such small slowdowns cannot reliably be detected in the cloud, at least not with the maximum number of instances we tested, i.e., 20.

6 Discussion

In this section, we discuss the implications of this study’s results for researchers and practitioners. First and foremost, we want to address whether cloud resources can be employed to assess software performance with microbenchmarks. This has multiple aspects to it, (1) which cloud provider and instance type is used, (2) how many measurements are taken, (3) which measurements are considered for slowdown analysis, (4) which slowdown sizes are desired to be detected, (5) and which statistical method to use. Following, we explicitly address the threats to validity and limitations of our study. Lastly, we briefly mention potential future directions of performance measurements/testing in unstable and uncontrollable environments such as public clouds.

6.1 Implications and Main Lessons Learned

In this section we discuss implications and main lessons learned from executing microbenchmarks in cloud environments. In detail, we comment on results for different cloud providers and instance types, discuss the impact of sample size and different sampling strategies on result variability, and elaborate on MDSs of this study’s subjects when following the adopted approaches.

6.1.1 Cloud Provider and Instance Type

The reliable detection of slowdowns directly depends on the choice of cloud provider. We have observed relevant differences between providers, and more stable providers eventually will lead to benchmark results with lower result variability. Based on the variability (see Section 4) and the results of our A/A and A/B testing (see Section 5), we see an indication across all experiments performed between July and October 2017 that benchmarks executed on AWS produce more stable results compared to GCE and Azure. Even better results are obtained when utilizing a bare-metal machine rented from IBM Bluemix. Surprisingly, variability and MDS are not far apart in AWS and Bluemix. An interesting conclusion drawn from the presented data is that there is no big difference between instance types of the same provider. When using microbenchmarks, there is no indication in our data that a particular instance-type family should be chosen over another. Nevertheless, a reader should carefully evaluate this for his/her application and microbenchmark types as this might not hold true for other applications and benchmarks. Especially if other performance metrics are tested for, e.g., IO throughput or memory consumption, a difference between specialized instance types might manifest. AWS recently introduced bare-metal instances¹⁰ as well, but this new instance type has only become available after concluding data gathering for the current study. We

¹⁰ <https://aws.amazon.com/de/blogs/aws/new-amazon-ec2-bare-metal-instances-with-direct-access-to-hardware/>

expect performance to be comparable to IBM's bare-metal offering. However, a detailed comparison is subject to future research.

6.1.2 Measurement Strategy

Both instance-based and trial-based sampling come with their own advantages and disadvantages. From a purely statistical point of view, the trial-based strategy leads to substantially better slowdown detection results. However, we argue that there is still room for the instance-based strategy, as it inherently lends itself better to typical continuous software development scenarios. The instance-based strategy can be implemented easily, but requires substantially higher sample sizes. One advantage which, to some extent, alleviates the problem of large sample size is that this strategy supports parallelization of test executions nicely. That is, in a public cloud, the only real factor preventing an experimenter from launching an experiment on hundreds of instances in parallel are costs. A disadvantage of instance-based sampling is that results need to be persisted such that a comparison is possible as soon as the new version's performance-test results are available. A more pressing issue is that you can not be sure that the cloud instances have not been upgraded by the provider between test executions of two versions. Therefore a comparison between two versions run at different points in time might not be fair and a potentially detected slowdown could be caused by a change to the underlying system. These problems are alleviated by implementing trial-based sampling. However, if the time budget is constrained (e.g., 2 hours per build job on TravisCI), one can only run half of the performance tests compared to instance-based sampling. Nonetheless *if* detecting the smallest-possible slowdowns is the primary goal, employing trial-based sampling — running test and control group on the same instances, ideally in randomized order — is preferable.

6.1.3 Required Number of Measurements

The result variability as well as the MDS is directly affected by the number of repeated benchmark executions. A naive suggestion would be to run as many as possible, which is at odds with temporal and monetary constraints. With the “limitless” availability of cloud instances, performance tests of a new software version can be executed in parallel. Even long-running benchmark suites could be split into subsets and run in parallel in order to reduce overall execution time. The only sequential operation are the number of trials on the same instance. Unfortunately, there is no generalizable rule for how many measurements are required to reliably detect slowdowns, as it depends on project, benchmark, and cloud configuration. Nevertheless our results show that for finding slowdowns of 10% or less, the number of repeated measurements has to include 20 instances with five trials each to be sure to catch slowdowns of this size even for less stable benchmarks. Our results also show that a few benchmarks (e.g., *rxjava-1*) are extremely stable even if only one instance per

test and control group is used. A tool chain utilizing cloud instances for performance testing, e.g., as part of a continuous integration (CI) build, should track benchmark-result variabilities and if needed reconfigure the repeated executions required to find small slowdowns for each benchmark individually.

6.1.4 Minimal-Detectable Slowdown Size

The minimal slowdown size we target to reliably detect directly influences the performance-testing-environment configuration. If it is sufficient to have a sanity check whether the performance of a software has not plummeted, an instance-based strategy with 20 instances and a single trial might be sufficient for most benchmarks. With this configuration some benchmarks will still be able to find relatively small slowdowns but others would only detect changes in the order of 100% or more. However, if slowdowns below 10% are desired to be detected — which is often the case for software-performance changes (Mytkowicz et al 2009) — a trial-based strategy with at least 20 instances and 5 trials is required for most of the studied benchmarks. Even with extensive testing on multiple instances, it is not guaranteed that all benchmarks of a project will reliably detect slowdowns of a certain size. We have observed multiple benchmarks, most evidently *log4j2-5*, that are inherently not able to detect realistic slowdowns in our setting. We argue that there is a need for developer tooling which supports developers in writing good benchmarks that are able to detect small slowdowns. It is a prerequisite to start from a stable benchmark in an ideal environment which can then be executed on cloud environments. If the benchmark is already inherently instable, moving to cloud-based execution will only make matters worse. For IO-intensive benchmarks, like most of the ones from *log4j2*, the transition to cloud infrastructure is troublesome as IO is particularly unreliable.

In any case, researchers as well as practitioners are encouraged to follow a similar approach to ours when executing performance experiments in cloud environments. That is, always perform A/A testing to validate whether the detected changes between different versions are due to software-performance changes or unreliable environments.

6.1.5 Testing Using Wilcoxon vs. Overlapping Confidence Intervals

Although both statistical tests are state-of-the-art in performance evaluation (Bulej et al 2017), traditional text books (Jain 1991) and research (Georges et al 2007; Kalibera and Jones 2012, 2013) tend to prefer confidence intervals of the mean over hypothesis testing (in form of Wilcoxon rank-sum). Our results show that for both testing approaches (A/A in Section 5 and A/B tests 5.4), Wilcoxon rank-sum reports more changes compared to overlapping confidence intervals. This implies that Wilcoxon is more sensitive towards changes and computed confidence intervals are more conservative. One advantage of Wilcoxon is that this test is computationally cheaper to run, as

the bootstrapping required for long-tailed performance data is fairly expensive. Which test is more recommendable for practitioners depends largely on whether the expensive bootstrapping is problematic, and whether false positives (i.e., false warnings about non-existent slowdowns) are worse than false negatives (i.e., missed slowdowns). However, it should be noted that both statistical tests can also be customized to be more or less conservative. For instance, one can easily imagine to test for a lower p-value or larger effect sizes in Wilcoxon.

6.2 Threats to Validity and Future Directions

As with any empirical study, there are experiment design trade-offs, threats, and limitations to the validity of our results to consider.

6.2.1 Threats to Internal and Construct Validity

Experiments in a public cloud always need to consider that the cloud provider is, for all practical purposes, a black box that we cannot control. Although reasonable model assumptions can be made (e.g., based on common sense, previous literature, and information published by the providers), we can fundamentally only speculate about the reasons for any variability we observe. Another concern that is sometimes raised for cloud experimentation is that the cloud provider may in theory actively impact the scientific experiment, for instance by providing more stable instances to benchmarking initiatives than they would do in production. However, in practice, such concerns are generally unfounded. Major cloud providers operate large data centers on which our small-scale experiments are expected to have a neglectable impact and historically, providers have not shown interest to directly interfere with scientific experiments. For the present study, we investigated entry-level instance types only. A follow-up study is required to investigate whether variability and detectability results improve for superior cloud hardware. Note that the compute-optimized instance of GCE has considerably lower memory (1.8GB vs. 3.6GB vs. 4GB) compared to the compute-optimized instance types of AWS and Azure. Hence a direct comparison between these instance types might not be valid. Another threat to the internal validity of our study is that we have chosen to run all experiments in a relatively short time frame. This was due to avoid bias from changes in the performance of a cloud provider (e.g., through hardware updates). This decision means that our study only reports on a specific snapshot and not on longitudinal data as would be observed by a company using the cloud for performance testing over a period of years. Another threat is concerned with the simulated slowdowns where each data point of the test group is increased by a fixed slowdown percentage, which represents an “idealized” situation. Nevertheless our comparison of results between approaches and statistical tests remain valid as they share the same data.

6.2.2 Threats to External Validity

We have only investigated microbenchmarking in Java and Go for a selected sample of benchmarks in two OSS projects. Other programming language paradigms (e.g., functional) and purely interpreted languages were considered out of scope. Nevertheless, we believe that with Java (dynamically compiled) and Go (statically compiled) we cover two of the most-used language-compilation/execution types. In addition, the availability of projects with microbenchmark suites limits the language options to study. In terms of performance testing paradigms, we do not claim that software microbenchmarking is a replacement for traditional load testing. A comparison between these two is not the goal of the underlying study but should be investigated in future research. Further, we have focused on three, albeit well-known, public cloud providers and a single bare-metal hosting provider. A reader should carefully evaluate whether our results can be generalized to other languages, projects, and cloud providers. Even more so, our results should not be generalized to performance testing in a private cloud, as many of the phenomena that underlie our results (e.g., noisy neighbors, hardware heterogeneity) cannot necessarily, or not to the same extent, be observed in a private cloud. Similarly, we are not able to make claims regarding the generalizability of our results to other types of performance experiments, such as stress or load tests. In this work, we focused on mean (by using bootstrapped overlapping confidence intervals) and median (by applying Wilcoxon rank-sum) execution-time performance. Future work should investigate other performance metrics (e.g., memory consumption, IO operations, or lock contention), as well as explore best/worst-case performance characteristics of microbenchmark suites. Finally, readers need to keep in mind that any performance benchmarking study run on cloud infrastructure is fundamentally aiming at a moving target. As long as virtualization, multi-tenancy, or control over hardware optimizations is managed by providers, we expect the fundamental results and implications of our work to be stable. Nevertheless, detailed concrete results (e.g., detectable slowdown sizes on particular provider/instance types) may become outdated as providers update their hardware or introduce new offerings.

6.3 Future Directions

So far, we studied how bad performance testing is in cloud environments and whether we are able to reliably detect slowdowns. This is a first step towards utilizing cloud infrastructure for performance measurements/testing, but naturally it is not the end of the story. Future research in this area should particularly address supporting developers in creating and executing performance tests on cloud infrastructure. Two concrete topics we envision are (1) studying the properties of benchmarks that contribute to better or worse stability and therefore slowdown detectability in uncontrolled environments and (2) supporting developers in writing appropriate benchmarks for execution in these.

The properties to study are root causes that contribute to higher variability in cloud environments (e.g., writing to files, network access, or execution configuration of benchmarks). We then foresee great potential for better tooling that (a) supports developers through IDE extensions to write adequate benchmarks (e.g., compiler optimizations that invalidate benchmark results) and hint which kind of properties might lead to higher result variability and (b) suggests/adapts execution configuration (i.e., repetitions along trials and instances) for a given cloud-resource type through continuous monitoring of benchmark result variability of prior executions.

7 Related Work

Software performance is a cross-cutting concern affected by many parts of a system and therefore hard to understand and study. Two general approaches to software performance engineering (SPE) are prevalent: measurement-based SPE, which executes performance experiments and monitors and evaluates their results, and model-based SPE, which predicts performance characteristics based on the created models (Woodside et al 2007). In this paper, we focus on measurement-based SPE.

It has been extensively studied that measuring correctly and applying the right statistical analyses is hard and much can be done wrong. Mytkowicz et al (2009) pinpoint that many systems researchers have drawn wrong conclusions through measurement bias. Others report on wrongly quantified experimental evaluations by ignoring uncertainty of measurements through non-deterministic behavior of software systems, such as memory placement or dynamic compilation (Kalibera and Jones 2012). Dealing with non-deterministic behavior of dynamically optimized programming languages, Georges et al (2007) summarize methodologies to measure languages such as Java, which dynamically compile and run on VMs. Moreover, they explain which statistical methods lend themselves for performance evaluation of these languages. All of these studies expect an as-stable-as-possible environment to run performance experiments on. More recently, Arif et al (2017) study the effect of virtual environments on load tests. They find that there is a discrepancy between physical and virtual environments, which are most strongly affected by unpredictability of IO performance. Our paper augments this study, which looks at result unreliability of load tests, whereas we investigate software microbenchmarks. Additionally, our study differs by conducting measurements in cloud environments rather than virtual environments on controlled hardware.

Traditionally, performance testing research was conducted in the context of system-scale load and stress testing (Menascé 2002; Jiang and Hassan 2015; Weyuker and Vokolos 2000; Barna et al 2011). By now, such performance tests are academically well-understood, and recent research focuses on industrial applicability (Nguyen et al 2014; Foo et al 2015) or on how to reduce the time necessary for load testing (Grechanik et al 2012). Studies of software microbenchmarking have not received main stream attention previously, but

academics have recently started investigating it (Stefan et al 2017; Horkey et al 2015; Chen and Shang 2017). Similarly, Leitner and Bezemer (2017) recently investigated different practices of microbenchmarking of OSS written in Java. However, none of these studies report on the reliability of detecting slowdowns.

A substantial body of research has investigated the performance and stability of performance of cloud providers independently of software-performance-engineering experiments. Iosup et al (2011) evaluate the usability of IaaS clouds for scientific computing. Gillam et al (2013) focus on a fair comparison of providers in their work. Ou et al (2012) and Farley et al (2012) specifically focus on hardware heterogeneity, and how it can be exploited to improve a tenant’s cloud experience. Our study sets a different focus on software performance tests and goes a step further to investigate which slowdowns can be detected.

7.1 Comparison to Our Previous Work

In this section, we compare the results of the work presented here to our two previous papers that investigated performance of cloud instances. In Leitner and Cito (2016), we studied the performance characteristics of cloud environments across multiple providers, regions, and instance types. We assessed the stability of four public cloud providers (the same as in the current work) using low-level, domain-independent CPU and IO system benchmarks (e.g., calculating a series of prime numbers, or writing and reading big files to/from hard disk), using a methodology not unlike previous benchmarking works (Iosup et al 2011; Gillam et al 2013). This paper laid the groundwork for the presently presented research, but findings from these isolated, domain-independent tests are not easy to transfer to any concrete usage domain, such as performance benchmarking.

In Laaber and Leitner (2018), we studied the quality of open-source performance test suites. Similarly to the present work, we measured the variability of performance tests in different environments, but the ultimate goal was to study benchmark test coverage. Hence, we were unable to go deeper in this topic. This has brought us to the idea to conduct a more targeted study, which ultimately led to the present work, which can be seen as applying a rigorous cloud-benchmarking methodology, similar to Leitner and Cito (2016), to the study of microbenchmarking-based performance tests.

It should be noted that the detailed results between these two previous works and the present one differ in some aspects. Most importantly, AWS has been benchmarked to be fairly unreliable in Leitner and Cito (2016), while the results of the present study show surprisingly stable results for the same provider. While such results may seem inconsistent, they are unavoidable in practice. Cloud providers routinely buy new hardware, roll out different server management policies, and offer entirely new and different services. Hence, one should not read the outcomes of a cloud benchmarking study as a set-in-stone truth. Rather, the goal is to identify common themes, trends, and benchmark-

ing methodologies, which remain valid significantly longer than the detailed data. Similarly, some microbenchmarks that have been identified in Laaber and Leitner (2018) as particularly stable have proven much less so in the present study (e.g., `log4j-1`, which we explicitly selected as a stable benchmark, but which has proven to be highly unstable in the present study). This primarily underlines how important a larger study of the subject, as presented here, is.

8 Conclusions

This paper empirically studied “how bad” performance testing with software microbenchmarks in cloud environments actually is. By executing microbenchmark suites of two Java projects (*Log4j2* and *RxJava*) and two Go projects (*bleve* and *etcd*) in bare-metal and cloud environments, we studied result variability, investigated falsely-detected performance changes (A/A tests), and identified minimal-detectable slowdown sizes. The results of the A/A tests and minimal-detectable slowdown sizes were retrieved by applying two state-of-the-art statistical tests, namely Wilcoxon rank-sum and overlapping bootstrapped confidence intervals of the mean.

We found result variabilities of the studied benchmark-environment configurations ranging between 0.03% to 100.68%, with the bare-metal and environment and AWS delivering the best results. In terms of falsely detect performance changes, the A/A test results show experiments with small sample sizes (e.g., one instance and one trial) suffer drastically from high false-positive rates, irrespective of which statistical test, sampling strategy, and execution environment is used. With increased sample sizes (e.g., five instance and five trials) though, most benchmark-environment combinations show acceptable numbers of false positives ($\leq 5\%$) when repeatedly executed, hence making it feasible to use cloud instances as performance-test execution environment. With regards to minimal-detectable slowdowns (MDSs), executing test and control group on the same instances (trial-based sampling) enables finding slowdowns with high confidence in all benchmark-environment combinations when utilizing ≥ 10 instances. In 77 – 83% of the time, a slowdown below 10% is reliably detectable when using trial-based sampling and 20 instances. We further found that Wilcoxon rank-sum is superior to overlapping confidence intervals in two regards: (1) it detects smaller slowdowns reliably and (2) it is not as computational-intensive and therefore takes less time.

Following these findings, we conclude that executing software microbenchmarking experiments is, to some degree, possible on cloud instances. Not all cloud providers and instance types perform equally well in terms of detectable slowdowns. However in most settings, a substantial number of trials or instances and the co-location of test and control group on the same instances is required to achieve robust results with small detectable slowdowns. Practitioners can use our study as a blueprint to evaluate the stability of their own performance microbenchmarks within their custom experimental environment.

Acknowledgements We are grateful for the anonymous reviewers' feedback, which helped to significantly improve this paper's quality. The research leading to these results has received funding from the Swiss National Science Foundation (SNF) under project MINCA – Models to Increase the Cost Awareness of Cloud Developers (no. 165546), the Wallenberg AI and Autonomous Systems and Software Program (WASP) funded by Knut and Alice Wallenberg Foundation, and Chalmers' ICT Area of Advance.

References

- Abedi A, Brecht T (2017) Conducting repeatable experiments in highly variable cloud computing environments. In: Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering, ACM, New York, NY, USA, ICPE '17, pp 287–292, DOI 10.1145/3030207.3030229, URL <http://doi.acm.org/10.1145/3030207.3030229>
- Arif MM, Shang W, Shihab E (2017) Empirical study on the discrepancy between performance testing results from virtual and physical environments. *Empirical Software Engineering* DOI 10.1007/s10664-017-9553-x, URL <https://doi.org/10.1007/s10664-017-9553-x>
- Barna C, Litoiu M, Ghanbari H (2011) Autonomic Load-testing Framework. In: Proceedings of the 8th ACM International Conference on Autonomic Computing, ACM, New York, NY, USA, ICAC '11, pp 91–100, DOI 10.1145/1998582.1998598, URL <http://doi.acm.org/10.1145/1998582.1998598>
- Bulej L, Horký V, Tůma P (2017) Do we teach useful statistics for performance evaluation? In: Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion, ACM, New York, NY, USA, ICPE '17 Companion, pp 185–189, DOI 10.1145/3053600.3053638, URL <http://doi.acm.org/10.1145/3053600.3053638>
- C Davison A, Hinkley D (1997) Bootstrap methods and their application 94
- Chen J, Shang W (2017) An Exploratory Study of Performance Regression Introducing Code Changes. In: Proceedings of the 33rd International Conference on Software Maintenance and Evolution, New York, NY, USA, ICSME '17
- Cito J, Leitner P, Fritz T, Gall HC (2015) The making of cloud applications: An empirical study on software development for the cloud. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ACM, New York, NY, USA, ESEC/FSE 2015, pp 393–403, DOI 10.1145/2786805.2786826, URL <http://doi.acm.org/10.1145/2786805.2786826>
- Cliff N (1996) Ordinal Methods for Behavioral Data Analysis, 1st edn. Psychology Press
- Farley B, Juels A, Varadarajan V, Ristenpart T, Bowers KD, Swift MM (2012) More for your money: Exploiting performance heterogeneity in public clouds. In: Proceedings of the Third ACM Symposium on Cloud Computing, ACM, New York, NY, USA, SoCC '12, pp 20:1–20:14, DOI 10.1145/2391229.2391249, URL <http://doi.acm.org/10.1145/2391229.2391249>
- Foo KC, Jiang ZMJ, Adams B, Hassan AE, Zou Y, Flora P (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, IEEE Press, Piscataway, NJ, USA, ICSE '15, pp 159–168, URL <http://dl.acm.org/citation.cfm?id=2819009.2819034>
- Georges A, Buytaert D, Eeckhout L (2007) Statistically Rigorous Java Performance Evaluation. In: Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications, ACM, New York, NY, USA, OOPSLA '07, pp 57–76, DOI 10.1145/1297027.1297033, URL <http://doi.acm.org/10.1145/1297027.1297033>
- Gillam L, Li B, O'Loughlin J, Tomar APS (2013) Fair Benchmarking for Cloud Computing Systems. *Journal of Cloud Computing: Advances, Systems and Applications* 2(1):6, DOI 10.1186/2192-113X-2-6, URL <http://dx.doi.org/10.1186/2192-113X-2-6>
- Grechanik M, Fu C, Xie Q (2012) Automatically Finding Performance Problems with Feedback-Directed Learning Software Testing. In: Proceedings of the 34th International

- Conference on Software Engineering, IEEE Press, Piscataway, NJ, USA, ICSE '12, pp 156–166, URL <http://dl.acm.org/citation.cfm?id=2337223.2337242>
- Hesterberg TC (2015) What teachers should know about the bootstrap: Resampling in the undergraduate statistics curriculum. *The American Statistician* 69(4):371–386, DOI 10.1080/00031305.2015.1089789, URL <https://doi.org/10.1080/00031305.2015.1089789>, pMID: 27019512, <https://doi.org/10.1080/00031305.2015.1089789>
- Horky V, Libic P, Marek L, Steinhäuser A, Tuma P (2015) Utilizing performance unit tests to increase performance awareness. In: *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ACM, New York, NY, USA, ICPE '15, pp 289–300, DOI 10.1145/2668930.2688051, URL <http://doi.acm.org/10.1145/2668930.2688051>
- Iosup A, Yigitbasi N, Epema D (2011) On the performance variability of production cloud services. In: *Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, IEEE Computer Society, Washington, DC, USA, CCGRID '11, pp 104–113, DOI 10.1109/CCGrid.2011.22, URL <http://dx.doi.org/10.1109/CCGrid.2011.22>
- Jain R (1991) *The Art of Computer Systems Performance Analysis*. Wiley
- Jiang ZM, Hassan AE (2015) A Survey on Load Testing of Large-Scale Software Systems. *IEEE Transactions on Software Engineering* 41(11):1091–1118, DOI 10.1109/TSE.2015.2445340
- John LK, Eeckhout L (2005) *Performance Evaluation and Benchmarking*, 1st edn. CRC Press
- Kalibera T, Jones R (2012) Quantifying performance changes with effect size confidence intervals. Technical Report 4–12, University of Kent, URL <http://www.cs.kent.ac.uk/pubs/2012/3233>
- Kalibera T, Jones R (2013) Rigorous benchmarking in reasonable time. In: *Proceedings of the 2013 International Symposium on Memory Management*, ACM, New York, NY, USA, ISMM '13, pp 63–74, DOI 10.1145/2464157.2464160, URL <http://doi.acm.org/10.1145/2464157.2464160>
- Laaber C, Leitner P (2018) An evaluation of open-source software microbenchmark suites for continuous performance assessment. In: *MSR '18: 15th International Conference on Mining Software Repositories*, ACM, New York, NY, USA, DOI 10.1145/3196398.3196407, URL <https://doi.org/10.1145/3196398.3196407>
- Laaber C, Scheuner J, Leitner P (2019) Dataset, scripts, and online appendix "software microbenchmarking in the cloud. how bad is it really?". DOI 10.6084/m9.figshare.7546703, URL <https://doi.org/10.6084/m9.figshare.7546703>
- Leitner P, Bezemer CP (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*, ACM, New York, NY, USA, ICPE '17, pp 373–384, DOI 10.1145/3030207.3030213, URL <http://doi.acm.org/10.1145/3030207.3030213>
- Leitner P, Cito J (2016) Patterns in the chaos—a study of performance variation and predictability in public iaas clouds. *ACM Trans Internet Technol* 16(3):15:1–15:23, DOI 10.1145/2885497, URL <http://doi.acm.org/10.1145/2885497>
- Mell P, Grance T (2011) *The nist definition of cloud computing*. Tech. Rep. 800-145, National Institute of Standards and Technology (NIST), Gaithersburg, MD
- Menascé DA (2002) Load Testing of Web Sites. *IEEE Internet Computing* 6(4):70–74, DOI 10.1109/MIC.2002.1020328, URL <http://dx.doi.org/10.1109/MIC.2002.1020328>
- Mytkowicz T, Diwan A, Hauswirth M, Sweeney PF (2009) Producing wrong data without doing anything obviously wrong! In: *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM, New York, NY, USA, ASPLOS XIV, pp 265–276, DOI 10.1145/1508244.1508275, URL <http://doi.acm.org/10.1145/1508244.1508275>
- Nguyen THD, Nagappan M, Hassan AE, Nasser M, Flora P (2014) An Industrial Case Study of Automatically Identifying Performance Regression-Causes. In: *Proceedings of the 11th Working Conference on Mining Software Repositories*, ACM, New York, NY, USA, MSR 2014, pp 232–241, DOI 10.1145/2597073.2597092, URL <http://doi.acm.org/10.1145/2597073.2597092>

- Ou Z, Zhuang H, Nurminen JK, Ylä-Jääski A, Hui P (2012) Exploiting hardware heterogeneity within the same instance type of amazon ec2. In: Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Computing (HotCloud'12), USENIX Association, Berkeley, CA, USA, pp 4–4, URL <http://dl.acm.org/citation.cfm?id=2342763.2342767>
- Ren S, Lai H, Tong W, Aminzadeh M, Hou X, Lai S (2010) Nonparametric bootstrapping for hierarchical data. *Journal of Applied Statistics* 37(9):1487–1498, DOI 10.1080/02664760903046102, URL <https://doi.org/10.1080/02664760903046102>, <https://doi.org/10.1080/02664760903046102>
- Romano J, Kromrey J, Coraggio J, Skowronek J (2006) Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen'sd for evaluating group differences on the NSSE and other surveys? In: Annual Meeting of the Florida Association of Institutional Research, pp 1–3
- Scheuner J, Leitner P, Cito J, Gall H (2014) Cloud work bench – infrastructure-as-code based cloud benchmarking. In: Proceedings of the 2014 IEEE 6th International Conference on Cloud Computing Technology and Science, IEEE Computer Society, Washington, DC, USA, CLOUDCOM '14, pp 246–253, DOI 10.1109/CloudCom.2014.98, URL <http://dx.doi.org/10.1109/CloudCom.2014.98>
- Stefan P, Horky V, Bulej L, Tuma P (2017) Unit testing performance in java projects: Are we there yet? In: Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering, ACM, New York, NY, USA, ICPE '17, pp 401–412, DOI 10.1145/3030207.3030226, URL <http://doi.acm.org/10.1145/3030207.3030226>
- Weyuker EJ, Vokolos FI (2000) Experience with Performance Testing of Software Systems: Issues, an Approach, and Case Study. *IEEE Transactions on Software Engineering* 26(12):1147–1156, DOI 10.1109/32.888628, URL <http://dx.doi.org/10.1109/32.888628>
- Woodside M, Franks G, Petriu DC (2007) The future of software performance engineering. In: 2007 Future of Software Engineering, IEEE Computer Society, Washington, DC, USA, FOSE '07, pp 171–187, DOI 10.1109/FOSE.2007.32, URL <http://dx.doi.org/10.1109/FOSE.2007.32>