PeerJ Preprints NOT PEER-REVIEWED

Performance Testing in the Cloud. How Bad is it Really?

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

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

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

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, cloud environments are inherently unpredictable and variable with respect to their performance. In this study, we explore the effects of cloud environments on the variability of performance testing outcomes, and to what extent regressions can still be reliably detected. We focus on software microbenchmarks as an example of performance tests, and execute extensive experiments on three different cloud services (AWS, GCE, and Azure) and for different types of instances. We also compare the results to a hosted bare-metal offering from IBM Bluemix. In total, we gathered more than 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 (from 0.03% to > 100% relative standard deviation). We also observe that testing using Wilcoxon rank-sum generally leads to unsatisfying results for detecting regressions due to a very high number of false positives in all tested configurations. However, simply testing for a difference in medians can be employed with good success to detect even small differences. In some cases, a difference as low as a 1% shift in median execution time can be found with a low false positive rate given a large sample size of 20 instances.

Keywords

performance testing, microbenchmarking, cloud

1 Introduction

In many domains, renting computing resources from public clouds has largely replaced privately owning resources. This is due to economic factors (public clouds can leverage economies of scale), but also due to the convenience of outsourcing tedious data center or server management tasks [6]. However, one often-cited disadvantage of public clouds is that the inherent loss of control can lead to highly variable and unpredictable performance (e.g., due to co-located noisy neighbors) [7, 13, 17]. This makes adopting cloud computing for performance testing, where predictability and low-level control over the used hard- and software is key, a challenging proposition.

There are nonetheless many good reasons why researchers and practitioners might be interested in adopting public clouds for their experiments. They may not have access to dedicated hardware, or the hardware that they have access to may not scale to the experiment size that the experimenter has in mind. They may wish to evaluate the performance of applications under "realistic conditions", which nowadays often means running it in the cloud. Finally, they may wish to make use of the myriad of industrial-strength infrastructure automation tools, such as Chef[1] or AWS CloudFormation[2], which ease the setup and identical repetition of complex performance experiments.

In this paper, we ask the question whether using a standard public cloud for software performance experiments is always a bad idea. To manage the scope of the study, we focus on a specific class of cloud service, namely Infrastructure as a Service (IaaS), and on a specific type of performance experiment (evaluating the performance of open source software products in Java or Go using microbenchmarking frameworks, such as JMH[3]). In this context, we address the following research questions:

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

RQ 2. *How large does a regression need to be to be detectable in a given cloud environment? What kind of statistical methods lend themselves to confidently detect regressions?*

We base our research on 20 real microbenchmarks sampled from four open source projects written in Java or Go. Further, we study instances hosted in three of the most prominent public IaaS providers (Google Compute Engine, AWS EC2, and Microsoft Azure) and contrast the results against a dedicated bare-metal machine deployed using IBM Bluemix (formerly Softlayer). We also evaluate and compare the impact of common deployment strategies for performance tests in the cloud, namely running test and control experiments on different cloud instances, on the same instances, and randomized multiple interleaved trials as recently proposed as best practice [1].

We find that result variability ranges from 0.03% relative standard deviation to > 100%. 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 studied environments. We conclude that there are different types of instability (variability inherent to the benchmark, variability between trials, and variability between instances), which needs to be handled differently.

Further, we find that standard hypothesis testing (Wilcoxon rank-sum in our case) is an unsuitable tool for performance-regression detection in cloud environments due to false positive rates of 26% to 81%. However, despite the highly variable measurement results, medians tend to be stable. Hence, we show that simply testing for

[1] https://www.chef.io
[2] https://aws.amazon.com/cloudformation
[3] http://openjdk.java.net/projects/code-tools/jmh

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3507v1 | CC BY 4.0 Open Access | rec: 4 Jan 2018, publ: 4 Jan 2018

difference in medians can be used to surprisingly good results, as long as a sufficient number of samples can be collected. For the baseline case (1 trial at 1 instance), we are mostly only able to detect regression sizes >20%, if regressions are detectable at all. However, with increased sample sizes (e.g., 20 instances, or 5 trials), even a median shift of 1% is detectable for some benchmarks.

Following the findings, we can conclude that executing software microbenchmarks is possible on cloud instances, albeit with some caveats. Not all cloud providers and instance types have shown to be equally useful for performance testing, and not all microbenchmarks lend themselves to reliably finding regressions. In most settings, a substantial number of trials or instances are required to achieve good results. However, running test and control groups on the same instance, optimally in random order, reduces the number of required repetitions. Practitioners can use our study as a blueprint to evaluate the stability of their own performance microbenchmarks, and in the environments and experimental settings that are available to them.

2 Background

We now briefly summarize important background concepts that we use in our study.

2.1 Software Microbenchmarking

Performance testing is a common term used for a wide varity of different approaches. In this paper, we focus on one very specific technique, namely software microbenchmarking (sometimes also referred to as performance unit tests [12]). Microbenchmarks are short-running (e.g., < 1ms), unit-test-like performance tests, which attempt to measure fine-grained performance counters, 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 average execution time.

In the Java world, the Java Microbenchmarking Harness (JMH) has established itself as the de facto standard to implement software microbenchmarking [24]. JMH is part of the OpenJDK implementation of Java and allows users to specify benchmarks through an annotation mechanism. Every public method annotated with `@Benchmark` is executed as part of the performance test suite. Listing 1 shows an example benchmark from the *RxJava* project, which is used as one of the studied benchmarks (*rxjava-5*).

Conversely, in the Go programming language, a benchmarking framework is included directly in the standard library[4]. This framework is primarily based on conventions. For instance, benchmarks are defined in files ending with `_test.go` as functions that have a name starting with `Benchmark`.

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

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 [18]. This definition distinguishes three service

```
@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 from the *RxJava* project.

models: IaaS, Platform as a Service (PaaS), and Software as a Service (SaaS). These levels differ mostly in which parts of the cloud stack is managed by the cloud provider and what is left for the customer. Most importantly, in IaaS, computational resources are acquired and released in the form of virtual machines or containers. Tenants do not need to operate 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 underyling infrastructure. Further, setting up performance experiments in IaaS is significanly simpler than doing the same in a typical PaaS system.

In IaaS, a common abstraction is the notion of an instance: an instance is a bundle of resources (e.g., CPUs, storage, networking capabilites, etc.) defined through an instance type and an imagine. 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 [7, 22]. Instance types are commonly grouped into families, each representing a different usage class (e.g., compute-optimized, 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 [20]. 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 a 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 [17]. To study the extent of these effects, we take the following approach.

We choose a subset of benchmarks from four open-source software (OSS) projects written in two programming languages. These

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

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3507v1 | CC BY 4.0 Open Access | rec: 4 Jan 2018, publ: 4 Jan 2018

PeerJ Preprints NOT PEER-REVIEWED

Short Name	Package	Benchmark Name	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	BenchmarkTop100of50Scores	-
bleve-2	/index/upsidedown/benchmark_null_test.go	BenchmarkNullIndexing1Workers10Batch	-
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 1: Overview of selected benchmarks. For JMH benchmarks with multiple parameters, we also list the concrete parameterization we used in our experiments.

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 to each other in terms of variability, and detectability of regressions.

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. We decided to choose Java as it has been ranked highly in programming language rankings (e.g., Tiobe[5]), is executed in a virtual machine (VM) (i.e., the JVM) with dynamic compiler optimizations, and has a microbenchmarking framework available that is used by real OSS projects [24]. Go 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 framework[6] as part of its standard library. We chose these languages due to their different characteristics, which improves generalizability of our results.

In a pre-study, we investigated OSS projects in these languages that make extensive use of microbenchmarking. We chose *Log4j2*[7] and *RxJava*[8] for Java, and *bleve*[9] and *etcd*[10] for Go as study subjects. 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. We then ranked all benchmarks for each project in the order of result stability between these five repetitions and selected the ones that are: the most stable, the most unstable, the median, as well as the 25th and 75th percentile across repeated executions. Our intuition is to pick five benchmarks from each project that range from stable to unstable results, to explore the effect result variability has on regression detectability. Information about these benchmarks are depicted in Table 1.

[5]https://www.tiobe.com/tiobe-index
[6]https://golang.org/pkg/testing
[7]https://logging.apache.org/log4j
[8]https://github.com/ReactiveX/RxJava
[9]https://github.com/blevesearch/bleve
[10]https://github.com/coreos/etcd

Cloud Provider Selection. As cloud environments to test, we choose entry-level general purpose, compute-optimized, and memory-optimized instance types of three cloud service providers. We expect instance types with better specifications to outperform the entry-level ones, and therefore this study sets a base line of what is possible with the cheapest available cloud resource options. The selected providers are Amazon with Amazon Web Services (AWS) EC2, Microsoft with Azure, and Google with Google Compute Engine (GCE). Table 2 shows the different instance types selected and information about the data center region, CPU and memory specification, and hourly price of these. All cloud instances run Ubuntu 17.04 64-bit.

Provider	Data Center	Instance Type	vCPU	Mem	Optimized for	Cost [USD/h]
AWS	us-east-1	m4.large	2	8.00	GP	0.1
AWS	us-east-1	c4.large	2	3.75	CPU	0.1
AWS	us-east-1	r4.large	2	15.25	Mem	0.133
Azure	East US	D2s v2	2	8.00	GP	0.1
Azure	East US	F2s	2	4.00	CPU	0.1
Azure	East US	E2s v3	2	16.00	Mem	0.133
GCE	us-east1-b	n1-standard-2	2	7.50	GP	0.0950
GCE	us-east1-b	n1-highcpu-2	2	1.80	CPU	0.0709
GCE	us-east1-b	n1-highmem-2	2	13.00	Mem	0.1184

Table 2: Overview of used cloud instance types.

Additionally, we selected a bare-metal machine available for rent from IBM in its Bluemix (formerly known as Softlayer) 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 the entry-level bare-metal server equipped with a 2.1GHz Intel Xeon IvyBridge (E5-2620-V2-HexCore) processor and 4 x 4GB RAM, running Ubuntu 16.04 64-bit version, hosted in IBM's data center in Amsterdam, NL. We specifically deactivated Hyperthreading and Intel's TurboBoost. Moreover, we attempted to disable frequency scaling, but manual checks revealed that this setting is ineffective, and probably overridden by IBM.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3507v1 | CC BY 4.0 Open Access | rec: 4 Jan 2018, publ: 4 Jan 2018

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 2, we create 50 different *instances*. On each instance, we schedule 10 experiment *trials* of each benchmark in randomized order (following the method proposed by Abedi and Brecht [1]) without breaks between trials. Within each trial, every benchmark (e.g., *etcd-1*) consists of 50 repeated *executions* (e.g., using the `-i50` parameter of JMH) and every execution produces a single *data point*, which reports the average execution time in *ns*. For JMH benchmarks, we also run 10 warmup executions (after which steady-state performance is most likely reached [9]) prior to the test executions. The performance counters originating from warmup iterations are discarded. We use the same terminology of instances, trials, executions, and data points in the remainder of the paper. These concepts are also summarized in Figure 1.

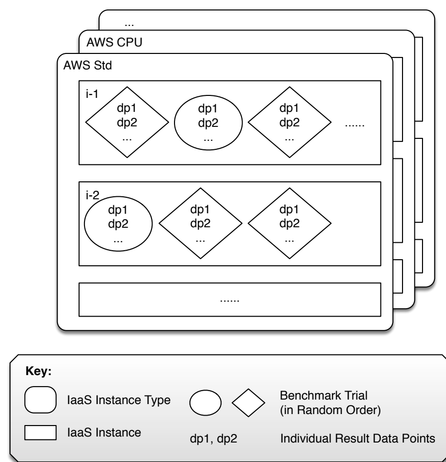

Figure 1: Schematic view of instance types, instances, and randomized-order trials on each instance.

For setting up the instances we used Cloud Workbench[11] [23], a toolkit for conducting cloud benchmarking experiments. Cloud Workbench sets up machines with Vagrant[12] and Chef, collects performance test results, and delivers the outcomes 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 5 million unique data points from our 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. Table 3 lists how many data points we have collected for each instance type across all instances, trials and executions, as well as how many data points remain after outlier removal. 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.

[11]https://github.com/sealuzh/cloud-workbench
[12]https://www.vagrantup.com

Short Name	Instance Type	Total	Cleaned
AWS Std	`m4.large`	484402	484056
AWS CPU	`c4.large`	484472	484471
AWS MEM	`r4.large`	474953	474951
Azure Std	`D2s v2`	472636	472122
Azure CPU	`F2s`	510640	510528
Azure MEM	`E2s v3`	470570	470400
GCE Std	`n1-standard-2`	474491	474490
GCE CPU	`n1-highcpu-2`	484138	483959
GCE MEM	`n1-highmem-2`	484168	483793
Bluemix	-	595037	594778

Table 3: Overview of the number of collected data points per cloud instance type, and how many data points remain after data cleaning.

4 Results

We now present the empirical study's results and answer the research questions stated in the introduction.

4.1 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. Specifically, we report the variability of each benchmark across all 50 instances, 10 trials on each instance, and 50 executions per trial, for all benchmarks and on all instance types. We refer to such a combination of benchmark and instance type as a configuration $c \in C$. We use the relative standard deviation (RSD) in percent as the measure of variability, defined as $100 \cdot \frac{\sigma(DP_c)}{\mu(DP_c)}$, with $\sigma(DP_c)$ representing the standard deviation and $\mu(DP_c)$ denoting the mean value of all data points collected for a configuration $c \in C$. We report all data in Table 4. Note that this table conflates three different sources of variability: (1) the difference in performance between different cloud instances, (2) the variability between different trials, and (3) the "inherent" variability of a benchmark, i.e., how variable the resulting performance counters are even in the best case. Consequently, a large RSD in Table 4 can have different sources, including, but not limited to, an unpredictable cloud instance type or an instable benchmark.

Differences between Benchmarks and Instance Types. It is evident that the investigated benchmarks have a wide range of result variability. Consequently, the potential regression to be detected by the benchmarks varies drastically depending on the benchmark and instance type it is executed on. We observe three classes 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. The first group's result indicate that variability is as desired low, making the latter two groups particular interesting to identify reasons for their instability.

We observe three benchmarks that have high result variability, namely *log4j2-1*, *log4j2-5*, and, to a lesser extent, *etcd-5* on all instance types. There are two factors that lead to high variability, either the execution time of the benchmark is very small, 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

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3507v1 | CC BY 4.0 Open Access | rec: 4 Jan 2018, publ: 4 Jan 2018

PeerJ Preprints
NOT PEER-REVIEWED

Benchs	AWS			GCE			Azure			BM
	Std	CPU	Mem	Std	CPU	Mem	Std	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	9.27	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 4: Result variability in RSD [%] for every benchmark and instance type configuration in the study.

inaccuracy becomes an important factor for variability. In contrast, *etcd-5* (see also Figure 2) has an execution time around 250000ns on *GCE Mem* with an RSD of 27.17%. This variability is similar to *all* other instance types, with RSDs ranging from 22.46% to 43.59%. Even the bare-metal machine from *Bluemix* has high variability of 23.73% RSD. This indicates that the benchmark itself is rather low-quality and produces unstable measurement results, independently of where it is executed.

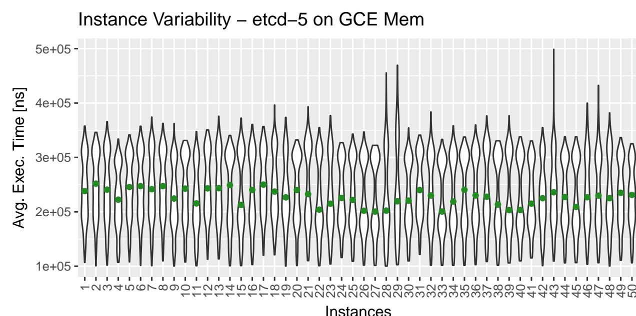

Figure 2: Drilldown into the variability of *etcd-5*, an example of a benchmark with high result variability across all instance types.

The second group we discuss exhibits high variability on some, but not all, instance types. This group contains two sub-types, (1) benchmarks that have high standard deviations, but where the median runtime is similar, 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 benchmark's variability differs among the instances of the same instance types (see Figure 3). We observe that this benchmark on this instance type, and unlike *etcd-5*, has a "long tail" distribution, which is not uncommon 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 *log4j2* benchmarks, as a 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 [17].

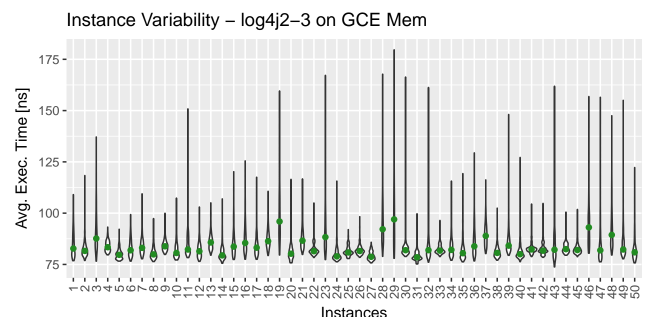

Figure 3: Drilldown into the variability of *log4j2-3*, an example of a benchmark with differing variability between instances.

More severe variabilities can be observed with the second subgroup, where even medians are shifted substantially between instances. This is illustrated in Figure 4 for *bleve-5* on Azure. A potential cause for this phenomenon is hardware heterogeneity [7, 22], i.e., different hardware configurations being used for different instances of the same instance type (e.g., different processor generations). Another potential root cause can be the Intel Turbo Boost Technology, which overclocks the CPU if compute-intensive applications are currently running on the particular hardware host. Given that the medians fall into a small number of different groups (only 2 in the case of Figure 4), we conclude that multi-tenancy is not the culprit for these cases.

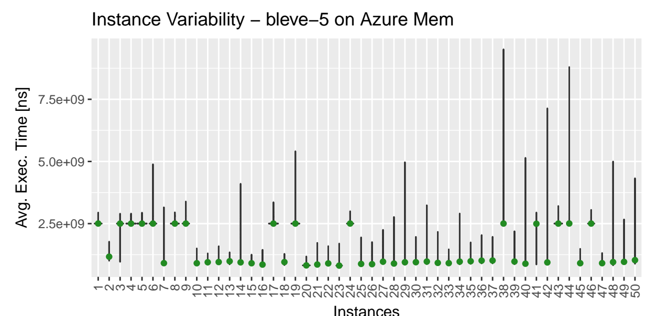

Figure 4: Drilldown into the variability of *bleve-5* on Azure, and example for a benchmark with high benchmark result variability due to differences in hardware.

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 from each other drastically.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3507v1 | CC BY 4.0 Open Access | rec: 4 Jan 2018, publ: 4 Jan 2018

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 and compute-optimized types. 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, therefore, show little difference. A detailed exploration of this finding is out of scope of the current work.

Sources of Variability. We now discuss the different sources of variability (between instance performance, between trials, inherent to the benchmark) in more detail. 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 RSD values: the total RSD of a configuration, as also given in Table 4 (*Total*), the average RSD per instance (*Per Instance*), and the average RSD per trial (*Per Trial*). For the latter two, error bars signifying the standard deviation are also provided. Error bars for *Total* are not meaningful, as this is a single RSD value per configuration.

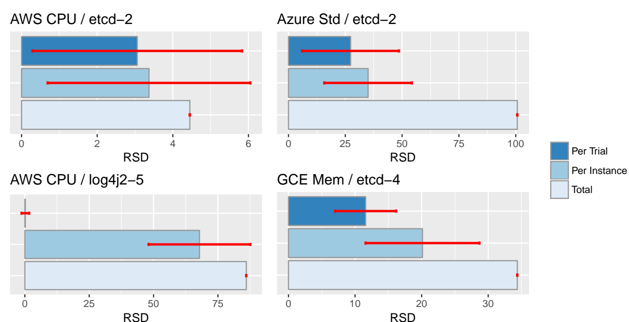

Figure 5: Drilldown on the source of variability for four example configurations.

The top left and right subfigures provide a drilldown on *etcd-2* in different clouds, and explores in more detail why this curious benchmark is remarkably stable on AWS and unstable in GCE and Azure (see also Table 4). The top left subfigure shows the benchmark on *AWS CPU*. There is little difference in the RSDs between trials, instances, and total, signifying that the little variability that the benchmark has largely originates from the benchmark itself. The large error bars signify that this variability differs from trial to trial, independently of which instance the trial is executed on. This is different for the same benchmark executed on Azure (top right subfigure). While the per-trial and per-instance RSD is also larger than on AWS, it is particularly the total RSD that is now very high. This signifies that the reason for the difference in stability between providers is indeed due to 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 miniscule, but there are substantial differences between different trials, largely independently of whether those 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 JVM just-in-time optimizations. 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.

IaaS Instances vs. Bare-metal Cloud. The study's idea is to not only compare performance test results on different resources from a diverse set of cloud providers, but also compare the nine IaaS instance types with results obtained from a hosted bare-metal server, in our case from IBM Bluemix. Our initial hypothesis was that the bare-metal server would provide substantially more stable results than any other instance type. Based on the data, we can not support that hypothesis. Benchmarks with high result variability across all instance types (*log4j2-1*, *log4j2-5*, and *etcd-5*) perform badly on bare-metal as well. An interesting observation is that AWS performs about the same compared to Bluemix in terms of result variability. Some benchmarks (e.g., *rxjava-3*) even have more reliable results in AWS than the bare-metal solution. One potential reason for AWS being surprisingly close to the stability of a hosted bare-metal server is its provisioned IOps[13] which is the default for all three used AWS instance types. Provisioned IOps throttles the IO operations to 10 Mbps, but at the same time guarantees consistent bandwidth, whereas the other providers generally offer IO bandwidth on a best effort basis. However, Bluemix consistently leads to less variable results than the other cloud providers in the study.

4.2 Detecting Regressions

To answer RQ 2, we first explore the usability of standard hypothesis testing (Wilcoxon rank-sum in our case) for reliable regression detection in cloud environments, and then study the configuration effects (increased number of measurements) that intra- and inter-instance variability has on detectable regression sizes.

4.2.1 A/A Testing We initially study the applicability of Wilcoxon rank-sum (also referred to as Mann-Whitney U test) for performance-change detection by performing A/A tests of all project's benchmarks with different sample sizes. 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) and observe whether the statistical test environment is not able to reject H_0 (i.e., the test environment does not find a regression if, by construction, there is none).

Recall the approach from Section 3, Figure 1: we executed every benchmark on each individual instance 10 times (number of trials) and repeated these trials 50 times on different instances of the same instance type. We now randomly select 1, 5, 10, and 20 instances (as indicated by the column "# VMs") and then randomly select 5 trials for the test and 5 trials for the control group. This implements a recent best practice for low cloud measurement variabilities as proposed by Abedi and Brecht [1].

For each benchmark we repeated this experiment 100 times with different instances, test, and control groups. Table 5 reports in percent how often a Wilcoxon rank-sum test between the test and

[13] https://aws.amazon.com/de/about-aws/whats-new/2012/07/31/announcing-provisioned-iops-for-amazon-ebs/

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3507v1 | CC BY 4.0 Open Access | rec: 4 Jan 2018, publ: 4 Jan 2018

PeerJ Preprints NOT PEER-REVIEWED

control group rejects H_0 for a p-value < 0.05. Wilcoxon rank-sum is stable for non-normally distributed data, which most of our data is, and, therefore, fits the needs of this study [4]. For readability, we aggregate the data over all benchmarks of the projects. Intuitively, the table reports in percent how often a Wilcoxon rank-sum test falsely reported a performance change when neither benchmark nor code had in fact changed.

Project	# VMs	AWS			GCE			Azure			BM
		Std	CPU	Mem	Std	CPU	Mem	Std	CPU	Mem	
log4j2	1	74	81	79	76	73	70	72	72	71	77
	5	68	78	71	72	63	60	62	73	66	65
	10	66	79	70	71	59	57	55	75	64	60
	20	63	77	62	72	57	49	52	72	52	51
RxJava	1	64	68	60	58	62	58	44	47	42	65
	5	62	62	59	60	56	55	39	44	44	61
	10	61	60	56	59	56	58	39	45	42	56
	20	63	62	59	58	54	62	38	42	35	53
bleve	1	55	55	58	72	66	65	53	46	47	64
	5	64	65	62	66	70	70	55	55	55	62
	10	58	63	64	68	65	67	56	56	60	55
	20	55	67	58	69	70	67	53	46	60	46
etcd	1	41	41	40	53	54	50	34	34	28	47
	5	39	41	41	47	47	48	33	33	30	42
	10	42	38	43	45	46	43	29	28	30	42
	20	41	44	44	44	42	44	28	26	28	35

Table 5: False positive rates in percent of A/A testing with Wilcoxon rank-sum tests, testing between multiple trials on multiple instances.

Overall, the rate of falsely reported differences (false positives (FPs)) between identical executions is very high, ranging from 26% up to 81% detected changes in the already conservative test set up. For the 40 different configurations, in 10 cases, a single instance produced the least amount of FPs. This shows that even an increase of sample size does not necessarily help to reduce the number of FPs.

Different instance types of the same cloud provider report roughly the same number of FPs. In comparison to the variability results (see Section 4.1), the FP-rate between cloud providers are different. Whereas AWS and Bluemix exhibit the lowest variabilities, the statistical test shows the fewest FPs on Azure instances (12/16 Azure configurations). Unexpectedly, in 9 out of 16 cases, AWS shows the highest FP-rate with a minimum of 39% for *etcd* with 5 VMs. Further, the Bluemix bare-metal server is the only one that consistently has less FPs with increased sample sizes. Azure has the lowest numbers of FPs for all projects but *Log4j2*. Nevertheless, FP-rates are above 25%, which still indicates that this approach can not be used for reliable detection of performance changes in our environment. Hence, we conclude that hypothesis testing using Wilcoxon rank-sum tests is *not* a suitable vehicle for detecting performance degradations in cloud environments. It should be noted that we did not conduct extensive experiments with other statistical testing methods (e.g., comparison of confidence intervals or ANOVA). However, given the drastically unsatisfying results with Wilcoxon rank-sum tests, we do not expect other hypothesis testing methods to perform much better as to become practically useful. Given the overall high numbers of FPs we believe it is unlikely that other tests would outperform Wilcoxon rank-sum such that the FP-rate would consistently drop to 5% or less. Nonetheless, an extensive comparative study between the different types of statistical tests is open for future research.

4.2.2 Detectable Regression Sizes.

While our results using hypothesis testing are disappointing, the fairly stable medians we have observed in Section 4.1 (see also Figures 2 and 3, which depict the medians in green dots) indicate that a simpler testing method may prove useful. In the following, we experiment with what regressions can confidently be found when simply evaluating whether the test and control group differ in median by at least a pre-defined threshold.

```
Input: Benchmark results for all instances and trials of an instance type
       Sampling strategy select for test and control group
Data: Regressions to test for R = {1, 2, 3, 4, 5, 10, 15, 20, 25, 50, 75, 100} [%]
Result: Smallest regression size detectable
foreach r ∈ R in descending order do
    for 100 times do
        select random control (c) and test (t) group
        if |median(t) − median(c)| > r/2 then
            report FP
        end
        if |median(add_reg_to(t, r)) − median(c)| > r/2 then
            report TP
        end
    end
    if FP-rate ≤ 0.05 and TP-rate ≥ 0.95 then
        return r
    end
end
return no regression detectable
```

Algorithm 1: A simple approach for detecting regressions based on medians.

Threshold-based Approach for Regression Detection. To find the minimal regression detectable by a benchmark in a particular environment, we performed testing based on the approach described in Algorithm 1. This approach finds the smallest detectable regression across 100 random samples, where at least 95% of the time this regression was detected, which we refer to as true positive (TP), and less than 5% of the time an A/A test reported a change, referred to as FP. The sampling strategy (*select*) defines the control and test groups, i.e., which instances and trials both consist of. We describe the two sampling strategies we experimented with below.

Figure 6 shows an example illustrating the approach for *etcd-4* on *AWS Std*. The y-axis shows the percentage of detected regressions (either TPs or FPs), and the x-axis shows the regression tested for. The left subfigure shows the data for 1 trial per instance, the right for 5 trials. Both figures use only a single instance as sample size. The red line indicates the TPs, whereas the blue line depicts FPs. The horizontal dashed line illustrates the regression detectable by the benchmark in the chosen environment. Note that with increasing simluated regression, TPs typically goes up (it is easier to find larger regressions), but the number of FPs goes down (when we assume larger regressions, the threshold that we test for, which is fixed to be half the regression, goes up, rendering the A/A test to be more robust).

In the illustrated example, the minimum regression that can be detected is 20% for both number of trials. In both figures we notice that the TP-rate is high for all regressions, and for slowdowns <20% the FP-rate is high. In this example the FP-rate is the deciding factor when a regression is detectable abiding our approach.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3507v1 | CC BY 4.0 Open Access | rec: 4 Jan 2018, publ: 4 Jan 2018

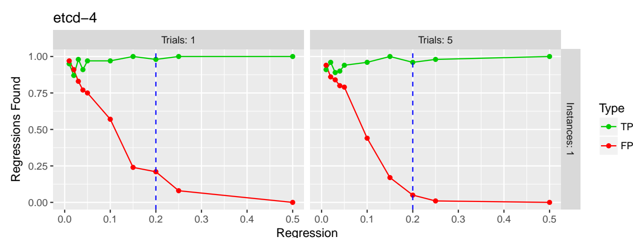

Figure 6: An illustration for *etcd-4* on *AWS Std* how the detectable regressions are calculated.

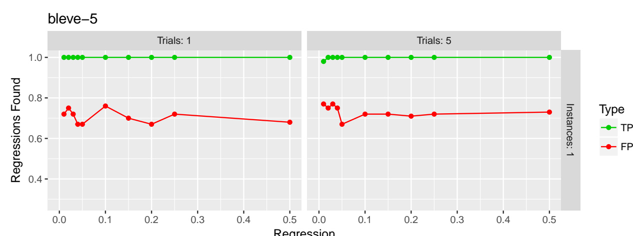

Figure 7: In the case of *bleve-5* on *Azure Mem*, and for a sample size of a single instance, no regression that we tested was large enough due to the very high rate of FPs.

A further example in Figure 7 shows that with such a small measurement sample size, some benchmarks (e.g., *bleve-5* in *Azure Mem*) are entirely unstable and produce a high rate of FPs. Therefore, they can not reliably detect regressions at all. We now investigate the detectable regressions for all configurations using this strategy.

Instance-Based Sampling. We will use and contrast two different sampling procedures. First, we randomly select 1, 5, and 20 different instances, and use 1 random trial from each instance. This simulates the case of a performance experiment where the test and control group are run on different cloud instances. This can happen, for instance, when a software team runs a performance test for each new release, and uses as control group the data gathered in the previous release. Between releases, cloud instances are terminated to save costs. We refer to this as the *instance-based sampling strategy*. Table 6 shows the results for this sampling strategy. Values represent a regression by x%. Cells with the value "no" (colored in red), indicate that for the given configuration no regression can be detected with the approach described above.

We investigate 190 combinations overall. This sampling strategy combined with the approach outlined above detects a slowdown in 159 combinations. The remaining 31 combinations generate too unstable result to reliably find the regressions that we tested for (up to 100%). This indicates that some benchmark-environment combinations produce too unstable results for reliable regression detection. In 12 out of the 31 undetectable cases, an increase to 5 instances, and in further 8 cases 20 instances, allows us to find at least large regressions reliably. In 15 of the examined combinations, namely *log4j-5* on all instance types, even a sample size increase did not allow us to reliably detect a regression of 100% or lower.

Bench	# VMs	AWS Std	AWS CPU	AWS Mem	GCE Std	GCE CPU	GCE Mem	Azure Std	Azure CPU	Azure Mem	BM
log4j2-1	1	5	4	2	20	no	50	20	no	50	1
	5	1	1	1	10	10	15	10	10	10	1
	20	1	1	1	10	5	10	4	4	4	1
log4j2-2	1	10	25	3	25	20	25	20	50	25	1
	5	3	20	1	15	10	15	10	15	10	1
	20	1	5	1	10	10	10	4	5	4	1
log4j2-3	1	10	5	5	25	20	25	50	25	25	4
	5	4	3	1	15	15	15	10	15	10	2
	20	2	1	1	10	5	10	4	5	5	1
log4j2-4	1	10	10	15	25	50	50	50	50	25	15
	5	5	4	5	15	15	15	15	15	10	5
	20	3	3	2	10	5	10	5	10	5	3
log4j2-5	1	no	no	no	no	no	no	no	no	no	no
	5	no	no	no	no	no	no	no	no	no	no
	20	no	no	no	no	no	no	no	no	no	no
rxjava-1	1	1	1	1	1	1	1	1	1	1	1
	5	1	1	1	1	1	1	1	1	1	1
	20	1	1	1	1	1	1	1	1	1	1
rxjava-2	1	2	4	3	20	25	50	10	10	10	2
	5	1	1	2	15	15	20	5	5	4	1
	20	1	1	1	10	5	10	2	3	2	1
rxjava-3	1	10	5	10	25	25	25	20	25	20	20
	5	5	4	3	10	15	15	10	10	10	10
	20	3	3	2	10	10	10	4	4	4	10
rxjava-4	1	15	15	50	50	50	20	15	25	15	5
	5	10	10	50	15	15	10	10	10	10	3
	20	4	3	10	10	10	5	5	4	3	1
rxjava-5	1	10	25	50	no	no	75	50	50	no	10
	5	10	10	20	25	25	20	50	25	50	2
	20	5	4	5	15	10	15	25	20	20	1
bleve-2	1	5	10	15	20	25	25	25	15	25	1
	5	2	3	10	10	10	20	10	10	10	1
	20	2	2	4	10	5	10	4	4	3	1
bleve-3	1	3	10	10	20	25	25	50	20	50	1
	5	1	2	5	15	15	20	15	15	10	1
	20	1	1	3	10	5	10	5	10	5	1
bleve-4	1	15	10	20	20	50	50	50	50	100	1
	5	5	5	15	15	15	50	20	20	15	1
	20	3	3	4	10	10	10	10	10	10	1
bleve-5	1	75	50	50	75	no	no	no	no	no	50
	5	25	25	10	15	20	no	no	no	no	15
	20	15	10	10	10	10	25	100	no	100	10
etcd-1	1	2	5	10	20	20	25	20	20	20	1
	5	1	2	2	10	10	20	10	10	10	1
	20	1	1	2	10	5	10	3	4	3	1
etcd-2	1	3	4	3	50	no	no	no	no	no	20
	5	2	3	2	20	50	no	no	no	no	10
	20	1	1	1	15	20	100	50	75	50	10
etcd-3	1	3	5	4	25	20	25	20	15	25	1
	5	1	2	2	15	10	25	10	10	10	1
	20	1	1	1	10	5	10	3	4	3	1
etcd-4	1	20	20	25	50	50	no	no	no	no	25
	5	10	15	15	15	20	no	100	100	75	15
	20	5	10	10	10	10	15	75	75	50	10
etcd-5	1	5	15	15	no	no	75	50	50	50	20
	5	4	10	10	25	50	20	20	50	25	10
	20	2	4	4	20	20	10	15	15	15	4

Table 6: Minimum detectable regression using instance-based sampling and a single trial per instance.

Apart from *log4j2-5*, AWS and Bluemix are the only providers that do not have result variabilities such that our approach is unable to detect regressions reliably. Further, across all benchmarks, these two providers offer a platform for the lowest detectable regressions, which is in line with our results from Section 4.1. This is especially evident for the first four *log4j2*, the first three *bleve*, and the first three *etcd* benchmarks. For those, on AWS and Bluemix, it is possible to detect regressions between 1% and 10%, whereas in comparison on GCE and Azure slowdown sizes above 10% are the usual case. In line with the variability results, IBM's bare-metal environment

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3507v1 | CC BY 4.0 Open Access | rec: 4 Jan 2018, publ: 4 Jan 2018

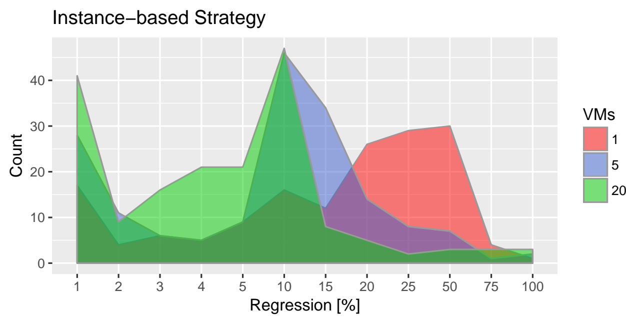

Figure 8: Regressions found using the instance-based sampling strategy. With one sample, many benchmarks are only able to reliably detect regressions in the 20% to 50% range. With 20 instances, 3% to 10% regressions can often be found.

is best for detecting regressions with the instance-based strategy, although AWS achieves very good results as well. Sometimes AWS even outperforms Bluemix (e.g., *etcd-2*).

Taking a step back and highlighting the big picture, Figure 8 shows the number of combinations that capture a particular regression size as a density plot. This figure depicts the minimal regression sizes across various providers, instance types, and benchmarks, such that they are detectable with a given number samples (i.e., repetitions on different instances). An increase in the number of instances overall leads to smaller regressions becoming detectable.

Trial-Based Sampling. For evaluating the second strategy, we randomly select 1 and 20 instances per benchmark and environment. However, different from the instance-based strategy, the control and test group now consist of the same instances with five different randomly selected trials each. This simulates the case where a performance engineer starts 1 or 20 instances and then runs both, the test and control group, multiple times on the same instance in randomized order. This minimizes the impact of the performance of the specific instance, which we have established to be an important factor contributing to variability in many cases. Hence, we expect that this approach should generally lead to smaller regressions being detectable. The results are presented in Table 7.

The results are generally in line with expectations. *log4j2-5* remains unable to reliably detect regressions in any configuration. Similar to the instance-based strategy, the same benchmarks show undetectable regressions. However, in general, the number of configurations for which no regression can be found goes down considerably, even if just a single instance is used.

An increase in numbers of instances reduces the inter-instance variability drastically as depicted by Figure 9. Comparing the 20 instances cases, this 5 trial-based sampling reduces the detectable regression sizes from around 10% to 15% for the single trial instance-based sampling to below 4% for most configurations.

This confirms our initial expectation, as well as the previous result by Abedi and Brecht [1]: 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 regressions can be found with high confidence even in a comparatively unstable environment. This is largely due to the medians of benchmark results actually not varying much between runs, even of the distributions themselves have high standard deviation or a long tail.

Bench	# VMs	AWS			GCE			Azure			BM
		Std	CPU	Mem	Std	CPU	Mem	Std	CPU	Mem	
log4j2-1	1	5	1	1	10	75	15	25	50	15	1
	20	1	1	1	2	3	3	3	3	2	1
log4j2-2	1	15	25	3	10	10	10	20	25	15	1
	20	1	5	1	3	2	3	3	3	3	1
log4j2-3	1	10	4	4	10	10	10	15	20	15	5
	20	2	1	1	3	2	2	2	2	3	1
log4j2-4	1	10	10	15	15	10	15	15	20	15	20
	20	2	3	2	3	2	4	3	3	3	3
log4j2-5	1	no	no	no	no	no	no	no	no	no	no
	20	no	no	no	no	no	no	no	no	no	no
rxjava-1	1	1	1	1	1	1	1	1	1	1	1
	20	1	1	1	1	1	1	1	1	1	1
rxjava-2	1	2	2	1	4	10	10	10	10	10	2
	20	1	1	1	2	2	3	1	1	1	1
rxjava-3	1	10	5	5	5	10	10	10	10	10	15
	20	3	3	2	2	2	3	2	2	2	5
rxjava-4	1	10	10	10	10	10	10	10	10	10	5
	20	2	3	2	3	3	2	3	2	3	1
rxjava-5	1	3	4	5	no	no	10	10	5	10	2
	20	1	1	2	2	2	2	4	3	4	1
bleve-2	1	1	2	3	4	4	10	10	4	5	1
	20	1	1	1	2	1	3	1	2	1	1
bleve-3	1	1	1	2	3	4	10	10	10	15	1
	20	1	1	1	1	1	2	2	2	2	1
bleve-4	1	3	5	3	3	4	10	15	15	15	1
	20	1	1	1	1	2	2	2	2	3	1
bleve-5	1	5	5	10	10	4	15	no	no	no	50
	20	2	2	1	3	2	5	15	20	20	10
etcd-1	1	1	2	1	4	3	10	5	10	10	1
	20	1	1	1	1	1	2	1	2	1	1
etcd-2	1	1	2	1	20	50	50	100	no	no	20
	20	1	1	1	10	5	50	15	20	20	5
etcd-3	1	1	1	1	5	3	10	10	10	15	1
	20	1	1	1	1	1	3	1	2	1	1
etcd-4	1	4	5	4	10	10	10	no	50	50	20
	20	1	1	2	2	2	3	15	15	15	4
etcd-5	1	3	10	15	10	10	50	25	25	25	20
	20	1	2	2	3	3	4	5	10	5	3

Table 7: Minimum detectable regression using a trial-based sampling strategy, and 5 trials per instance.

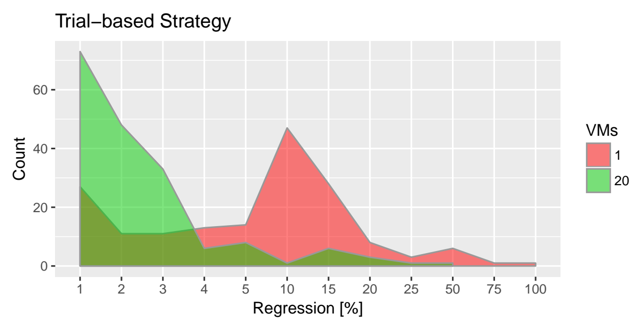

Figure 9: Regressions found using the trial-based sampling strategy. With one instance, regressions between 5% and 15% can often be found. With 20 instances, we are able to discover regressions as small as 4% in most configurations.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3507v1 | CC BY 4.0 Open Access | rec: 4 Jan 2018, publ: 4 Jan 2018

PeerJ Preprints NOT PEER-REVIEWED

5 Discussion

In this section, we discuss the implications of this study's results for researchers and practitioners. First and foremost, we want to address the titular question, "Performance Testing in the Cloud. How Bad is it Really?". The answer to this question 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 regression analysis, and (4) which regression sizes are desired to be detected. Following, we explicitly address threats to the validity and limitations of our study.

5.1 Implications and Main Lessons Learned

Cloud Provider and Instance Type. The reliable detection of regressions 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. Across all experiments performed we see an indication that at the time of writing, benchmarks executed on AWS produce results that are most stable compared to GCE and Azure. Even better results are obtained when utilizing a bare-metal machine rented from Bluemix. Surprisingly, variability and detectable regressions in AWS and Bluemix are not far apart. An interesting conclusion drawn from the presented data is that there is no big difference between instance types of the same provider, suggesting that using the cheapest option is often the right choice for performance experiments.

Required Number of Measurements. The result variability as well as the detectable slowdown is directly affected by the number of repeated benchmark executions. A naive suggestion would be to run as many as possible, which is obviously 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. There is no definite answer to exactly how many measurements are required for all benchmarks to reliably detect slowdowns, as this depends on project, benchmark, and cloud configuration. However, we have observed that even a single instance can often be used to detect small regressions if trials are repeated five times, and both, test and control groups, can be executed on the same instance within a short time frame and in random order.

Measurement Strategy. Both, instance-based and trial-based sampling, come with their own advantages and disadvantages. The trial-based strategy leads to substantially better regression detection results, but may not lend itself to typical continuous software development scenarios. The instance-based strategy can be implemented easily, but requires substantially higher sample sizes, or an experimenter who is willing to compromise on the regressions that can realistically be found. One advantage which, to some extend, alleviates the problem of high sample size is that this strategy supports parallelization of test executions nicely.

Detectable Regression Size. The regression size that is desired to detect influences the performance testing environment configuration. For example, if it is sufficient to detect regressions in the order of 20% to 50%, an instance-based strategy with 5 instances might be sufficient. However, if regressions below 10% are desired to be detected, more instances with multiple trials might be required. Even with extensive testing on multiple instances, it is not guaranteed that all benchmarks of a project will reliably detect regressions of a certain size. We have observed multiple benchmarks, most evidently *log4j2-5*, that are inherently not able to detect realistic regressions in our setting. We argue that there is a need for developer tooling which constantly tracks result variability of benchmarks on the current as well as other cloud providers to dynamically adapt to software as well as provider changes. Further, as shown in Section 4.2, standard hypothesis testing is not an appropriate tool for regression detection of performance measurements conducted on cloud instances. Before relying on detected performance variations, research as well as practitioners should validate whether the observed change is certain or potentially caused by the flakiness of the experiment environment. A/A testing can and should be used to assure the experimenter of this.

5.2 Threats to Validity and Limitations

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

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, which we consider to produce baseline results compared to better instances. A follow-up study is required to investigate whether variability and detectability results improve for superior cloud hardware. 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), but 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.

Threats to External Validity. We have only investigated microbenchmarking in Java and Go, and only for a selected sample of benchmarks in two OSS projects. 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 projects and 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

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3507v1 | CC BY 4.0 Open Access | rec: 4 Jan 2018, publ: 4 Jan 2018

results (e.g., noisy neighbors, hardware heterogeneity) cannot, 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. Further, we focus in our study on the detection of median execution-time related performance regressions. To keep the scope of our experiments manageable, we chose to not discuss worst-case performance or different performance counters (e.g., memory consumption or IO operations) in the present work. Finally, readers need to keep in mind that any cloud-provider-benchmarking study 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 regression sizes on particular provider/instance types) may become outdated as providers update their hardware or introduce new offerings. For instance, when AWS introduced provisioned IOps, the IO-related performance characteristics of their service changed sufficiently that previously-published academic benchmarking studies became outdated.

6 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 [26]. 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. [20] 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 (e.g., memory placement, dynamic compilation) [15]. Dealing with non-deterministic behavior of dynamically optimized programming languages, Georges et al. [9] summarize methodologies to measure languages like Java, and explain statistical methods to use for performance evaluation. All of these studies expect an as stable as possible environment to run performance experiments on. More recently, Arif et al. [2] study the effect virtual environments have 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 investigates 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 [3, 14, 19, 25]. By now, such performance tests are academically well-understood, and recent research focuses on industrial applicability [8, 21] or how to reduce the time necessary for load testing [11]. Studies of software microbenchmarking have not received main stream attention previously, but academics have recently started investigating it [5, 12, 24]. Similarly, Leitner and Bezemer [16] recently investigated different practices of microbenchmarking of OSS written in Java. However, none of these studies report on result reliability.

A substantial body of research has investigated the performance, and stability of performance, of cloud providers independently of software performance engineering experiments. For instance, Leitner and Cito [17] study the performance characteristics of cloud environments across multiple providers, regions, and instance types. Iosup et al. [13] evaluate the usability of IaaS clouds for scientific computing. Gillam et al. [10] focus on a fair comparison of providers in their work. Ou et al. [22] and Farley et al. [7] 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 regressions can be detected.

7 Conclusions

This paper empirically studied "how bad" performance testing (i.e., software microbenchmarking) in cloud environments actually is. We investigated result variability and minimal detectable regression sizes of microbenchmark suite subsets of two Java projects (*Log4j2* and *RxJava*) and two Go projects (*bleve* and *etcd*). The test suites were executed on general purpose, compute-optimized, and memory-optimized instance types of three public IaaS providers, i.e., Amazon's EC2, Google's GCE, and Microsoft Azure, and as comparison a hosted bare-metal environment from IBM Bluemix. Result variability, indicated as RSD, ranges for the studied benchmark-environment configurations vary between 0.03% to 100.68%. This variability originates from three sources (variability between instances, between trials, inherent to the benchmark), 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. Both, GCE and Azure seemed to lend themselves much less to performance testing in our experiments. Further, we found that A/A tests using Wilcoxon rank-sum tests produce high false-positive rates between 26% and 81%. However, a simple strategy based on comparing medians can be applied successfully, often to detect surprisingly small differences. For high sample sizes (e.g., 20 instances), performance differences as small as 1% can be found. If this is not possible, detectable regressions range between 10% and 50% for most cases or even beyond for some, depending on benchmark, instance type, and sample size.

Acknowledgments

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). Further, this work was partially supported by the Wallenberg Autonomous Systems and Software Program (WASP) funded by Knut and Alice Wallenberg Foundation.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3507v1 | CC BY 4.0 Open Access | rec: 4 Jan 2018, publ: 4 Jan 2018

References

[1] Ali Abedi and Tim Brecht. 2017. Conducting Repeatable Experiments in Highly Variable Cloud Computing Environments. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering (ICPE '17)*. ACM, New York, NY, USA, 287–292. https://doi.org/10.1145/3030207.3030229

[2] Muhammad Moiz Arif, Weiyi Shang, and Emad Shihab. 2017. Empirical study on the discrepancy between performance testing results from virtual and physical environments. *Empirical Software Engineering* (03 Oct 2017). https://doi.org/10.1007/s10664-017-9553-x

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

[4] Lubomír Bulej, Vojtech Horký, and Petr Tůma. 2017. Do We Teach Useful Statistics for Performance Evaluation?. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion (ICPE '17 Companion)*. ACM, New York, NY, USA, 185–189. https://doi.org/10.1145/3053600.3053638

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

[6] Jürgen Cito, Philipp Leitner, Thomas Fritz, and Harald C. Gall. 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 (ESEC/FSE 2015)*. ACM, New York, NY, USA, 393–403. https://doi.org/10.1145/2786805.2786826

[7] Benjamin Farley, Ari Juels, Venkatanathan Varadarajan, Thomas Ristenpart, Kevin D. Bowers, and Michael M. Swift. 2012. More for Your Money: Exploiting Performance Heterogeneity in Public Clouds. In *Proceedings of the Third ACM Symposium on Cloud Computing (SoCC '12)*. ACM, New York, NY, USA, Article 20, 14 pages. https://doi.org/10.1145/2391229.2391249

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

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

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

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

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

[13] Alexandru Iosup, Nezih Yigitbasi, and Dick Epema. 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 (CCGRID '11)*. IEEE Computer Society, Washington, DC, USA, 104–113. https://doi.org/10.1109/CCGrid.2011.22

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

[15] Tomas Kalibera and Richard Jones. 2012. *Quantifying Performance Changes with Effect Size Confidence Intervals*. Technical Report 4–12. University of Kent. 55 pages. http://www.cs.kent.ac.uk/pubs/2012/3233

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

[17] Philipp Leitner and Jürgen Cito. 2016. Patterns in the Chaos&Mdash;A Study of Performance Variation and Predictability in Public IaaS Clouds. *ACM Trans. Internet Technol.* 16, 3, Article 15 (April 2016), 23 pages. https://doi.org/10.1145/2885497

[18] Peter Mell and Timothy Grance. 2011. *The NIST Definition of Cloud Computing*. Technical Report 800-145. National Institute of Standards and Technology (NIST), Gaithersburg, MD.

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

[20] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. 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 (ASPLOS XIV)*. ACM, New York, NY, USA, 265–276. https://doi.org/10.1145/1508244.1508275

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

[22] Zhonghong Ou, Hao Zhuang, Jukka K. Nurminen, Antti YlÃď-JÃďÃďski, and Pan Hui. 2012. Exploiting Hardware Heterogeneity Within the Same Instance Type of Amazon EC2. In *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Ccomputing (HotCloud'12)*. USENIX Association, Berkeley, CA, USA, 4–4. http://dl.acm.org/citation.cfm?id=2342763.2342767

[23] Joel Scheuner, Philipp Leitner, Jürgen Cito, and Harald Gall. 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 (CLOUDCOM '14)*. IEEE Computer Society, Washington, DC, USA, 246–253. https://doi.org/10.1109/CloudCom.2014.98

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

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

[26] Murray Woodside, Greg Franks, and Dorina C. Petriu. 2007. The Future of Software Performance Engineering. In *2007 Future of Software Engineering (FOSE '07)*. IEEE Computer Society, Washington, DC, USA, 171–187. https://doi.org/10.1109/FOSE.2007.32

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3507v1 | CC BY 4.0 Open Access | rec: 4 Jan 2018, publ: 4 Jan 2018