An Exploratory Study of the Impact of Parameterization on JMH Measurement Results in Open-Source Projects

Hazem Samoaa
Chalmers | University of Gothenburg
Gothenburg, Sweden
samoaa@chalmers.se

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

ABSTRACT

The Java Microbenchmarking Harness (JMH) is a widely used tool for testing performance-critical code on a low level. One of the key features of JMH is the support for user-defined parameters, which allows executing the same benchmark with different workloads. However, a benchmark configured with n parameters with m different values each requires JMH to execute the benchmark m^n times (once for each combination of configured parameter values). Consequently, even fairly modest parameterization leads to a combinatorial explosion of benchmarks that have to be executed, hence dramatically increasing execution time. However, so far no research has investigated how this type of parameterization is used in practice, and how important different parameters are to benchmarking results. In this paper, we statistically study how strongly different user parameters impact benchmark measurements for 126 JMH benchmarks from five well-known open source projects. We show that 40% of the studied metric parameters have no correlation with the resulting measurement, i.e., testing with different values in these parameters does not lead to any insights. If there is a correlation, it is often strongly predictable following a power law, linear, or step function curve. Our results provide a first understanding of practical usage of user-defined JMH parameters, and how they correlate with the measurements produced by benchmarks. We further show that a machine learning model based on Random Forest ensembles can be used to predict the measured performance of an untested metric parameter value with an accuracy of 93% or higher for all but one benchmark class, demonstrating that given sufficient training data JMH performance test results for different parameterizations are highly predictable.

ACM Reference Format:
Hazem Samoaa and Philipp Leitner. 2021. An Exploratory Study of the Impact of Parameterization on JMH Measurement Results in Open-Source Projects. In *Twelfth ACM/SPEC International Conference on Performance Engineering (ICPE'21).* ACM, New York, NY, USA, 12 pages.

1 INTRODUCTION

Performance testing is a crucial element of the quality assurance strategy of many software systems. In addition to load and stress testing, recent years have seen software microbenchmarking become more and more important. Microbenchmarks are performance tests that operate on a similar level of granularity as unit tests, i.e., they measure the performance of small code units, such as classes or external libraries. However, different from unit tests where the outcome is binary, i.e., a test passes or fails, benchmarks produce outputs for a certain performance metric, such as execution time or throughput [18]. In the Java ecosystem, the Java Microbenchmark Harness (JMH) is prevalently used to implement microbenchmarks.

A key feature of JMH is parameterization: JMH benchmarks can have user-defined parameters, and during performance testing JMH then runs the benchmark once for every possible combination of parameter values. Previous work indicates that existing JMH projects make extensive use of this style of parameterization as an easy way to test different workloads using a single benchmark implementation [18]. However, a benchmark configured with n parameters with m different values each requires JMH to execute the benchmark m^n times (once for each combination of configured parameter values). Consequently, even fairly modest use of parameterization leads to dramatically increased execution time of the microbenchmark suite. As a direct consequence, we have observed in previous work that executing the microbenchmark suite of some open source projects takes hours or even days to run in its default configuration [16].

In light of this, it is problematic that we currently know little about how important these parameters actually are for reliable Java benchmarking. It is conceivable that some parameters have in practice low impact on the result, and could be skipped to speed up benchmarking. Further, it is unclear how many different parameter values, and which, should be chosen, or if untested parameter values could not be inferred from ones that have already been tested.

To address this gap, we present results from an exploratory study of the current state of JMH parameterization in open source projects. We study five well-known projects (including RxJava, Log4J2, and the Eclipse Collections framework) and 126 concrete benchmarks. We generate a dataset of approximately 1.4 million measurements with various parameterizations for these benchmarks, which we then analyze. Further we investigate the question whether machine learning models can be used to infer the value of untested parameterizations. Hence, we contribute two-fold to the state of microbenchmarking literature:

- We empirically analyze the impact of different parameter values for a selection of in total 126 benchmarks of five well-known open source software projects. We find that 40% of metric parameters in our data set do not correlate with the measurement result (i.e., varying these parameters does not impact the measurement). If there is a correlation, it is often

ICPE'21, ,

strongly predictable, following a power law, linear, or step function curve.
- We further show that Random Forest ensembles can be used to predict the benchmark output for untested parameter values with an accuracy of 93% or better for all but one of 26 benchmark classes.

Our results provide a first understanding of practical usage of JMH parameters, and how they correlate with the measurements produced by benchmarks. Our machine learning experiments further demonstrate that given sufficient training data JMH performance test results for different parameterizations are highly predictable. We argue that collecting sufficient data in an industrial or open source project is computationally expensive but feasible, as long as it is sufficient to train models for a selection of benchmark classes only.

2 BACKGROUND

Performance testing is an umbrella term used for a wide variety of different approaches. The focus of the present work is *microbenchmarking*, i.e., short-running benchmarks that aim to measure fine-grained performance metrics, such as method-level execution times, throughput, or heap utilization. Different from application benchmarks, system tests, or load tests, the goal of microbenchmarking is not necessarily to enact the system under realistic, production-like conditions. Instead, microbenchmarks are often written specifically to test sensitive code elements for extreme conditions, or to compare multiple implementation variants of the same feature (e.g., different data structures or external libraries).

2.1 Java Microbenchmark Harness

By now, microbenchmarking frameworks are available for a wide range of programming languages, including C++, Java, and Go. In this paper, we specifically explore the Java Microbenchmark Harness[1] (JMH). JMH is part of the OpenJDK ecosystem and allows users to specify benchmarks through Java annotations, in a syntax clearly inspired by common unit testing frameworks such as JUnit. Laaber et al. have found in 2020 that 753 significant open-source projects on GitHub are actively using JMH [18], showing that the framework is widely accepted in practice.

An example of a real-life JMH benchmark class from the `JCTools` project is provided in Listing 1. The Java class `SetOps` defines four benchmarks, implemented in four methods carrying the `@Benchmark` annotation. In addition, four user-specified parameters are defined using `@Param`. The method annotated with `@Setup` is executed once at the beginning of the process to set up the necessary test data. In the remainder of this work, we refer to `SetOps` as the benchmark class, and `SetOps.add`, `SetOps.remove`, and so on as the concrete benchmarks in this class.

The execution model of JMH is straight-forward. For each benchmark, the framework generates the carthesian product of all combinations of parameter values, and instantiates the benchmark once for each possible combination of parameters. Each of these benchmark instance will then be executed $n + m$ times, where n is the number of warmup iterations (executions where the measurement result is discarded) and m is the number of measurement iterations.

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

```
@OutputTimeUnit(TimeUnit.NANOSECONDS)
@BenchmarkMode({ Mode.AverageTime })
@Warmup(iterations = 3, time = 1, timeUnit = TimeUnit.
    SECONDS)
@Measurement(iterations = 5, time = 1, timeUnit =
    TimeUnit.SECONDS)
public class SetOps {

    @Param("1024")
    int size;
    @Param("512")
    int occupancy;
    @Param("2048")
    int keyBound;
    @Param({ "java.util.HashSet",
        "org.jctools.sets.OpenHashSet" })
    String type;
    private Set<Key> set;
    private Key key;

    @Setup(Level.Trial)
    public void prepare() throws Exception {
        // set up test data according to config / params
    }

    @Benchmark
    public boolean add() {
        return set.add(key);
    }

    @Benchmark
    public boolean remove() {...}

    @Benchmark
    public boolean contains() {..}

    @Benchmark
    public int sum() {...}

}
```

Listing 1: Example of a JMH benchmark (from the `JCTools` project)

Each iteration consists of the framework executing the benchmark method as often as possible in a loop, until a configured timeout (commonly 1 or 10 seconds) is reached. The framework then records the measurements produced in the iteration (e.g., how often the method could be invoked, or, as is the case in the example, the average method execution time in nanoseconds).

In the example above, since `size`, `occupancy`, and `keybound` all only have a single value, only two distinct combinations of parameters need to be instantiated and executed by JMH. However, in the general (and more common) case where each parameter has multiple values, the presence of parameters leads to an explosion of the number of combinations that need to be run (and hence of the total time required to execute the benchmark suite of the project). This means that the innocent-looking action of adding a new parameter to a benchmark class with two values already doubles the time required for executing all benchmarks in this class – even if this new parameter is potentially not critical to the measurement result or does not provide many new insights. It is therefore not surprising that the benchmark suites of projects that routinely use parameters

with five or more different values (e.g., `eclipse-collections`) are very time-consuming, often taking multiple days to run in their standard configuration.

Given how central strategically choosing parameters and parameter values is to keeping the execution time of benchmark suites manageable, it is disappointing that parameters have not yet been explicitly studied in previous work on JMH. The goal of our study is to address this research gap.

3 APPROACH

We now introduce the subjects used in this study, and present our data collection and analysis approach.

3.1 Study Subjects

As common for exploratory research, we employ purposive sampling [3] on both, project and benchmark level. In purposive sampling, subjects are selected explicitly based on their perceived usefulness to the study goal (rather than drawn randomly from a population).

Concretely, we select five JMH-using study subjects (a) that frequently use parameterization in their benchmarks, (b) that are well-maintained, well-known, and important in practice, and (c) that cover a range of different types of projects. From each of these five projects, we manually select five to six concrete benchmark classes, again using a purposive sampling strategy. We select benchmark classes (a) that have parameters (typically multiple), and (b) that we suspect have different characteristics (due to having a different code structure, different parameters, or their location in different Java packages). Note that a benchmark class in JMH typically contains multiple concrete benchmarks (cf. Section 2), hence in total this selection procedure led to the execution of 126 concrete benchmarks. Table 1 summarizes the selected projects and their basic characteristics.

Project	Stars	Contr.	Description
eclipse-collections[1]	15k	74	*"Eclipse Collections is a collections framework for Java with optimized data structures and a rich, functional and fluent API."*
RxJava[2]	42k	263	*"RxJava – Reactive Extensions for the JVM – a library for composing asynchronous and event-based programs using observable sequences for the Java VM."*
JCTools[3]	2.4k	34	*"Java Concurrency Tools for the JVM. This project aims to offer some concurrent data structures currently missing from the JDK."*
Log4J2[4]	1k	99	*"Apache Log4j 2 is an upgrade to Log4j that provides significant improvements over its predecessor, Log4j 1.x, and provides many of the improvements available in Logback while fixing some inherent problems in Logback's architecture."*
jdk-microbenchmarks[5]	na	na	*"The JMH JDK Microbenchmarks is a collection of microbenchmarks for measuring the performance of the JDK API and JVM features using the JMH framework."*

1 – https://github.com/eclipse/eclipse-collections
2 – https://github.com/ReactiveX/RxJava
3 – https://github.com/JCTools/JCTools
4 – https://github.com/apache/logging-log4j2
5 – http://hg.openjdk.java.net/code-tools/jmh-jdk-microbenchmarks/

Table 1: Summary of selected study subjects and GitHub metadata (stars and contributors). Data has been extracted from GitHub on August 5th, 2020. Descriptions are quoted in verbatim from the project's websites. Both `Log4J2` and `jdk-microbenchmarks` are hosted outside of GitHub, explaining the relatively low number of stars for the former and the absence of data for the latter.

3.2 Data Collection

To collect data for our study, we built a tool that repeatedly executes the selected benchmarks with randomly generated parameter values. To do so, we manually inspected the selected benchmark classes and identified three types of parameters: (a) metric integer parameters, (b) metric decimal parameters, and (c) categorical parameters. For both types of metric parameters, we define minimum and maximum values loosely based on the current configuration of the projects on GitHub (extending the range if the range that is actually configured in the project is narrow), and randomly generate concrete parameter values through uniform sampling between minimum and maximum. For categorical parameters, we simply select randomly from all predefined parameter values. An overview of all selected benchmark classes as well as concrete parameters (and their ranges for generation) is given in Table 2. Our data collection tool is written in Python and available on GitHub[2].

We deployed the tool on three AWS EC2 `m5d.large` instances (each with two virtual CPUs, 8 GB RAM, and a 1st or 2nd generation Intel Xeon Platinum 8000 series processor). To minimize the impact of performance variation between cloud instances [17, 20], we collected all data for one project on the same instance. Further, we ensured that only one data collection process was running on an instance at a time (i.e., we scheduled the first three study subjects on the instances first, and only collected data for the remaining two subjects once the first projects were completed). We executed each benchmark 300 times, and configured 3 different (randomly selected) parameter values for each of the parameters of a benchmark in each execution. We executed JMH with a single fork, five warmup iterations, and 10 measurement iterations. In total, our data collection consumed approximately 400 compute hours, and produced over 1.4 million individual measurement points. The complete dataset, along with analysis scripts, is available as part of our replication package [27].

[2]https://github.com/icetlab/jmh_params_generator

3.3 Analysis

Our initial goal in analysis was to identify how strongly using different parameter values impacts the resulting benchmark measurements. To this end, we used a combination of visual analysis and feature selection.

3.3.1 Visual Analysis. For each combination of benchmark and (metric) parameter, we plotted the measurements as a function of the metric parameter. As categorical parameters are often used to implement different test configurations (e.g., using different data structures) we controlled for specific categorical parameter values in these plots. We then iteratively discussed and classified the resulting plots.

3.3.2 Feature Selection. Visual analysis as discussed above reaches its limits in cases of multiple metric parameters which strongly impact the benchmark result. Hence, we additionally applied feature selection (FS) for all benchmark classes with two or more parameters. FS is an important pre-processing step used in the fields of machine learning and data mining, and is known to have a major

Project	Class	BMs	Parameter	Range
eclipse-collections	LongIntMapTest	5	mapSizeDividedBy16000	[1; 100000]
			fullyRandom	{true,false}
	FunctionalInterfaceTest	13	megamorphicWarmupLevel	[0; 10]
	IntIntMapTest	4	mapSizeDividedBy64	[1; 100000]
			fullyRandom	{true,false}
	TroveMapPutTest	1	size	[250000; 10000000]
			isPresized	{true,false}
			loadFactor	[0.4; 0.6]
	ChainMapPutTest	3	size	[250000; 10000000]
			isPresized	{true,false}
			loadFactor	[0.6; 0.9]
RxJava	ParallelPerf	3	count	[100; 100000]
			compute	[1; 10000]
			parallelism	[1; 10]
	RangePerf	1	times	[1; 1000000]
	FlowableFlatMapCompletableSyncPerf	2	size	[1; 1000000]
			maxConcurrency	[1; 1024]
	ObservableFlatMapMaybePerf	3	count	[1; 1000000]
	ObservableConcatMapSinglePerf	3	count	[1; 1000000]
	FlattenRangePerf	2	times	[1; 1000000]
JCTools	QueueThroughputBackoffNone	1	qType	{SpscArrayQueue, MpscArrayQueue, SpmcArrayQueue, MpmcArrayQueue}
			qCapacity	[10000; 1000000]
	SetOps	4	type	{java.util.HashSet, org.jctools.sets.OpenHashSet}
			size	[8; 16384]
			occupancy	[8; 16384]
	SpscChannelThroughputTest	2	capacity	[10000; 100000]
	QueueOfferPoll	1	qType	{SpscArrayQueue, MpscArrayQueue, SpmcArrayQueue, MpmcArrayQueue}
			burstSize	[10; 1000]
			qCapacity	[10000; 1000000]
	ChannelThroughputBackoffNone	1	type	{Spsc, Mpsc}
			capacity	[10000; 1000000]
Log4J2	AsyncAppenderLog4j2Benchmark	12	configFileName	{perf5AsyncApndNoLoc-noOpAppender.xml, perf5AsyncApndDsrptrNoLoc-noOpAppender.xml, perf5AsyncApndMpscQNoLoc-noOpAppender.xml, perf5AsyncApndXferQNoLoc-noOpAppender.xml}
	ThreadContextBenchmark	6	threadContextMapAlias	{Default, CopyOpenHash, CopySortedArray, NoGcOpenHash, NoGcSortedArray}
			count	[5; 1000]
	ConcurrentAsyncLoggerToFileBenchmark	2	queueFullPolicy	{ENQUEUE, ENQUEUE_UNSYNCHRONIZED, SYNCHRONOUS}
			asyncLoggerType	{ASYNC_CONTEXT, ASYNC_CONFIG}
	MDCFilterBenchmark	3	size	[0; 10]
	SortedArrayVsHashMapBenchmark	17	count	[1; 1000]
			length	[1; 100]
jdk-microbenchmarks	ForkJoinPoolForking	5	workers	[0; 5]
			size	[1000; 100000000]
			threshold	[1; 10]
	ForkJoinPoolThresholdAutoSurplus	2	workers	[0; 5]
			size	[1000; 100000000]
			threshold	[1; 10]
	ArrayCopyUnalignedBoth	4	length	[1; 1200]
	Limit	24	size	[100; 1000000]
	URLEncodeDecode	2	count	[8; 16384]
			maxLength	[8; 16384]
			mySeed	[1; 10]

Table 2: Overview of selected study subjects, benchmark classes, and parameters. BMs is the number of distinct benchmarks in this class. For metric parameters, the range is provided as an interval of minimum and maximum values. For categorical parameters, all legal values are listed. All benchmarks in a class use the same parameters.

impact on the performance of learning models [1]. FS allows us to identify and remove irrelevant and redundant features (benchmark parameters in our case). In machine learning, FS results in optimized time and space requirements along with an enhanced performance of the learning model. We use FS two-fold: firstly, FS allows us to quantify the relative importance of parameters to the benchmark measurement result; secondly, FS is required as an integral step in building a machine learning model for inferring untested parameter values in Section 5.

The three main methods for FS are (1) wrapper, (2) filter, and (3) embedded approaches [24, 25]. Their main difference is that wrapper approaches include a learning algorithm in the feature subset evaluation step. The learning algorithm is used as a "black box" by a wrapper to evaluate the goodness (i.e., the model performance) of the selected features. A filter feature selection process is independent of any learning algorithm. Filter algorithms are often computationally less expensive and more general than wrapper algorithms. However, filters ignore the performance of the selected features on a learning algorithm, whereas wrappers evaluate the feature subsets based on the learning performance, which usually results in better performance achieved by wrappers than filters for a particular learning algorithm [24, 25, 31]. Embedded methods

are integrate feature selection and predictor learning into a single process, i.e., feature selection is performed as part of the model construction process. Embedded methods combine the advantages of, both wrapper and filter methods – they include the interaction of features with the learning model just like wrapper methods, and like filter methods, they are far less computationally intensive. In addition, embedded methods are able to detect the interaction between variables as they evaluate the entire data set at the same time, and also they tend to find the feature subset that is good for the algorithm being trained.

Preliminary experimentation with our dataset has shown that using an embedded FS approach based on importance derived from decision trees and Random Forest ensembles indeed performs best on our data. Hence, we use this FS approach in our analysis.

3.4 Threats to Validity

Despite careful research design, a study such as the one described in this paper is always subject to limitations and threats to validity, the most important of which we describe in the following.

In terms of **external validity**, our study investigated only a specific version of each benchmark project. While we argue that the chosen versions of benchmarks are representative of the problem we want to tackle, this evidence cannot be generalized to other versions, particularly since the performance value may change for each version even with the same set of parameters values. Similarly, we are not able to generalize to other open source projects. We have employed purposive sampling when identifying relevant study subjects and benchmarks. This is common in exploratory research, but inherently does not allow us to draw conclusions about the population in general. Similarly, our study does not generalize to other benchmark frameworks or programming languages. To summarize, our results should not be interpreted as a comprehensive survey of benchmark parameters in JMH or microbenchmarking overall.

In terms of **internal validity**, some design decisions had to be made when defining the range of possible parameter values for metric parameters. Exploratory data analysis has not suggested that any benchmark's behavior would have been radically different outside of the ranges we used, but this is evidently impossible to prove. For our last research contribution, we trained our machine learning model based on only one version of the study subjects. Hence, the model we trained and the conclusions we drew are not necessarily accurate for the projects in other points in time. However, we do speculate that small code modifications would likely not invalidate the models entirely. Another internal validity threat is that we executed all experiments on a single hardware platform. Some benchmarks in our study may potentially be impacted by our choice of hardware for data collection. However, we consider it unlikely that the general validity of our results is threatened by the specific hardware chosen for the study.

4 BENCHMARK PARAMETERS AND THEIR IMPACT ON MEASUREMENT RESULTS

As a first step, we discuss the results of an analysis of the impact of different types of parameters on the benchmark result. Our main goal is to map the landscape of JMH benchmark parameters used in practice, and to support developers in selecting appropriate benchmark parameter values.

4.1 Categorical Parameters

Categorical parameters often fundamentally change what exactly a benchmark tests (e.g., Listing 2, where a categorical parameter is used to configure what specific data structure to benchmark). In our data set, 11 of 27 (41%) benchmark classes contain at least one categorical parameter. One class (`ConcurrentAsyncLoggerToFileBenchmark` in the `Log4J2` project) has only categorical parameters. However, we observe that there are noticeable differences between projects – most categorical parameters are found in the `JCTools` and `Eclipse-Collections` projects (where almost every benchmark has one), while they are rarely or never used in the other three projects.

```
1 public class QueueOfferPoll {
2 // ...
3     @Param(value = { "SpscArrayQueue", "MpscArrayQueue",
       "SpmcArrayQueue", "MpmcArrayQueue" })
4     String qType;
5 // ...
6 }
```

Listing 2: Example categorical parameter (from the `JCTools` project)

In all cases, choosing different values for categorical parameters leads to a multi-modal distribution in the resulting benchmark measures. An example is depicted in Figure 1: depending on which data structure is benchmarked, the resulting distribution changes. Note that in some cases, two or more categorical parameter values may lead to distributions that are not statistically different (e.g., `SpmcArrayQueue` and `MpscArrayQueue` in Figure 1).

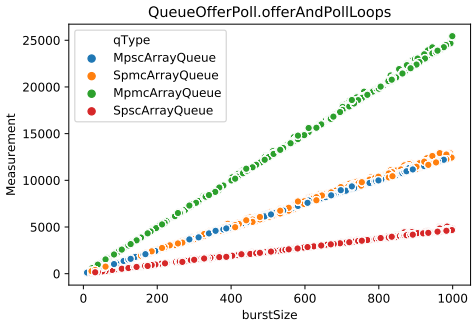

Figure 1: A categorical parameter leading to a multi-modal distribution of measurement values (example benchmark from `JCTools`, *burstSize* is a metric parameter, and *qType* is categorical)

In the example in Figure 1, all distribution modes are linear. However, we have also observed cases where different categorical parameter values have led to completely different types of distributions (see Figure 2 for an example). This example is particularly interesting in that the metric parameter *occupancy* only seems to matter if the categorical parameter *type* is set to *org.jctools.sets.OpenHashSet*, while being irrelevant if the other value is chosen.

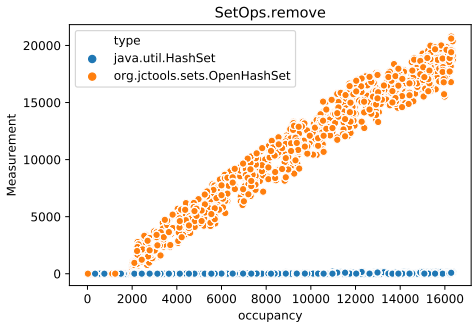

Figure 2: A categorical parameter where different values lead to fundamentally different benchmark behavior (example benchmark from `JCTools`, *occupancy* is a metric parameter, and *type* is categorical)

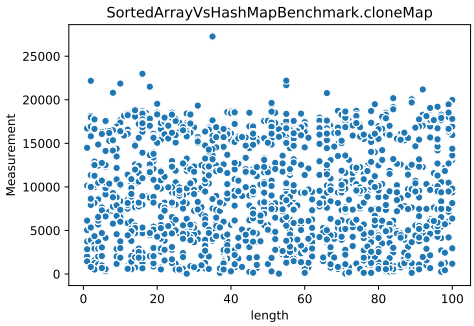

Figure 3: A metric parameter which does not impact the benchmark result (example benchmark from `Log4J2`, *length* is a metric parameter)

In general, it is not feasible to skip specific categorical parameter values without fundamentally changing the coverage of a benchmark suite. In some isolated cases, statistical analysis may indicate that two or more parameter values behave sufficiently similar that testing one of the values is sufficient. However, in the general case, it seems appropriate to think about benchmarks with categorical parameters as multiple independent benchmarks (one for each distinct parameter value).

4.2 Metric Parameters

We now present different categories of how parameters correlate with the measured performance values. From the 164 combinations of metric parameters and benchmarks distributed across 26 benchmark classes, we identified the following main types of correlation.

No Correlation. In 66 combinations of benchmarks and metric parameters (40%), we observed that the parameter was uncorrelated with the measured benchmark value. In other words, these parameters do not appear to impact the benchmark result at all, and developers can at least consider removing them entirely (however, there may still be good reasons to keep them, which we will discuss later).

An example from `Log4J2` is given in Figure 3: varying the integer parameter `length` has no observable impact on the resulting measurements. A similar (but more interesting) example is given in Figure 4. In this example from the same benchmark class, the parameter again has no correlation with the measured performance, but clearly there is a second variable that impacts the result (another metric parameter in this case, indicating a limitation of visually analysing parameters one by one).

For developers, identifying parameters of this type is valuable, as these parameters can – in principle – be removed without loss of coverage in the benchmark suite. In our data set, developers largely seemed to be aware that these parameters do not actually impact the result – in 49 of these 66 cases, the default configuration entails running the benchmark using only a single (fixed) parameter value. An example of a configuration with three parameters with a single value each is given in Listing 3.

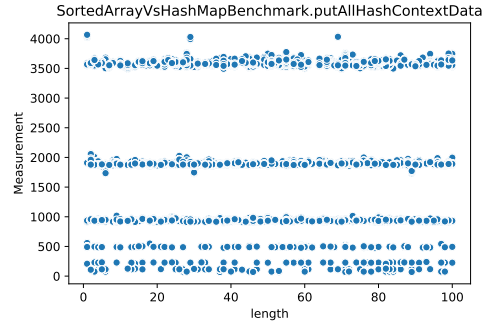

Figure 4: A metric parameter which does not impact the benchmark result, but a different parameter does (example benchmark from `Log4J2`, *length* is a metric parameter)

```
public class ForkJoinPoolForking {
// ...
    @Param("0")
    private int workers;
    @Param("10000000")
    private int size;
    @Param("10")
    private int threshold;
// ...
}
```

Listing 3: Example parameter configured with only a single value (from the `jdk-microbenchmarks` project)

In 17 cases, the developers are indeed testing with different values. However, there may still be practical justifications for doing so. (1) In JMH, parameters and parameter values are defined on benchmark class level, so if a parameter matters to *any* of the benchmarks in a class it cannot be omitted only for the ones where this parameter is not relevant. However, in our experiments, we could not observe any case where a parameter impacted one or more benchmarks in a class but not all of them (although the shape and type of distribution may vary). (2) There may be cases where developers have observed historical performance regressions by varying a parameter that normally should not impact the benchmark result,

and decide to keep the parameter in the test to identify future similar regressions. (3) Finally, having a parameter with only a single parameter value can still be pragmatically useful for performance testers, as it allows overriding this value from the commandline without having to re-compile the benchmark class.

Power Law Correlation. For metric parameters that actually impact the result, the most common relation between parameter and measured value in our sample are power law correlations, which we have observed in 42 benchmark / parameter combinations (26%). An example from `EclipseCollections` is given in Figure 5.

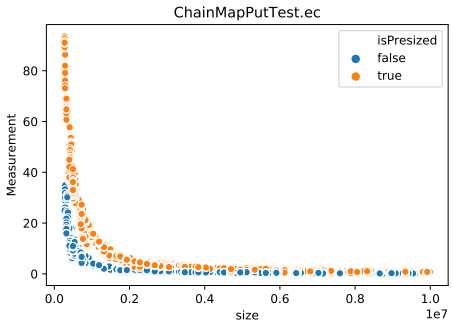

Figure 5: A power law correlation between metric parameter and benchmark result (example benchmark from `EclipseCollections`, *size* is a metric parameter, and *isPresized* is categorical)

A power law distribution can be mathematically described as $1/X$, i.e., after a rapid initial decline in the measurement the curve flattens out quickly, and further increasing the parameter value adds little new information. Once developers have established that a given parameter follows a power law, a focus should be put on benchmarking the short (initial) part of the curve with rapid decline. After the initial decline, further increasing the parameter value does not relevantly impact the measurement outcome. Developers of our study objects frequently show awareness of this fact, and often benchmark small values extensively and only sparsely cover larger, more extreme, benchmark parameters.

Linear Correlation. Besides power law, a second type of relation that we have observed with some regularity are linear correlations (22 combinations, or 13%). An example from the `Log4J2` project is given in Figure 6.

In many ways, a linear correlation is easy to handle for developers as a linear curve can theoretically be estimated from as little as two measurements. However, in practice, even linear correlations sometimes exhibit irregularities. For an example, observe the distribution of a different benchmark in the same class as the example above (Figure 7): while the relationship between the `count` parameter and the measured value for *threadContextMapAlias=Default* is still linear, there is a significant discontinuity around a parameter value of 800.

Logarithmic Correlation. Interestingly, logarithmic correlations are considerably more rare in our data. We have only observed

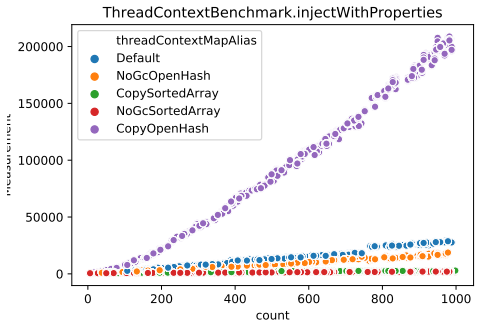

Figure 6: A linear correlation between metric parameter and benchmark result (example benchmark from `Log4J2`, *count* is a metric parameter, and *threadContextMapAlias* is categorical)

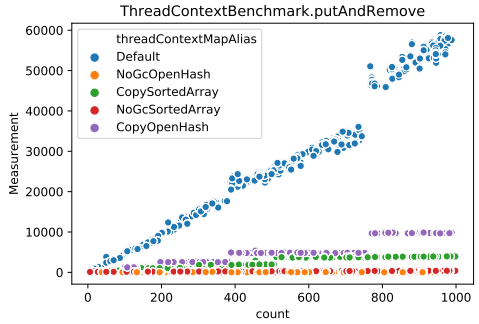

Figure 7: A linear correlation between metric parameter and benchmark result, with a discontinuity around *count=800* for one categorical parameter value (example benchmark from `Log4J2`, *count* is a metric parameter, and *threadContextMapAlias* is categorical)

2 instances (1%) of this pattern, both in the `SortedArrayVsHashMapBenchmark` benchmark class of the `Log4J2` project. An example is given in Figure 8. Note that this plot shows the same parameter in than Figure 6, but in a different concrete benchmark. This illustrates a technical limitation of JMH – parameters are defined on class-level, but the same parameter may be more important to some benchmarks than to others.

Non-Continuous Correlation. In 22 combinations (13%), we observed that the relationship between parameter and measured value cannot be described through a single function, as there are one or multiple discontinuities where the distribution changes. Most frequently, this takes the form of a step function (see Figure 9 from `Log4J2`).

For developers, awareness of this situation is crucial, as there is little to be gained from executing the benchmark multiple times with parameter values in the same range. However, identifying the "cutoff points" and covering all important ranges may be crucial, as we observe that the size of the steps generally does not follow any apparent pattern.

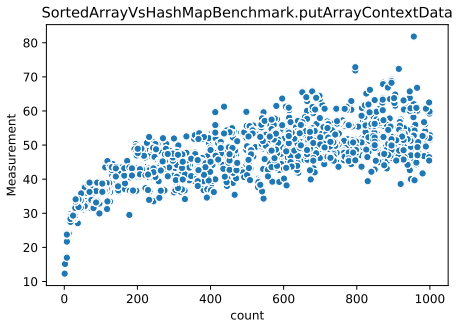

Figure 8: A logarithmic correlation between metric parameter and benchmark result (example benchmark from `Log4J2`, *count* is a metric parameter)

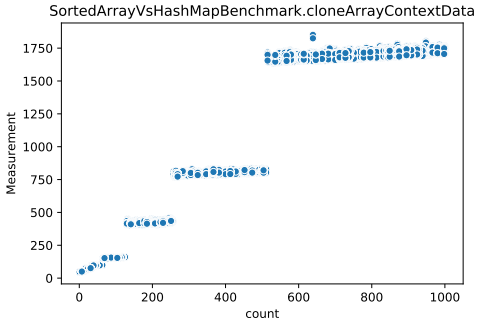

Figure 9: The correlation between metric parameter and benchmark result forms a step function with clearly different ranges (example benchmark from `Log4J2`, *count* is a metric parameter)

A special case of step functions are benchmarks where a single special parameter value leads to different behavior than all other values. For example, the `Eclipse-Collections` project uses the metric parameter `megamorphicWarmupLevel` in some benchmarks. Setting this value to 0 (equivalent to "no warmup") leads to different behavior than any positive value, but which warmup level is chosen concretely has no further impact on the result (see Figure 10).

However, not all non-continuous correlations are as regular. We have also observed some combinations of benchmarks and parameters with highly irregular correlation shapes, which do not adhere to any easily described statistical shape. An example, again from `Eclipse-Collections`, is given in Figure 11. Note that this case is different from the examples given in figures 3 and 4 (no correlation between parameter and measurement value): here, the parameter clearly impacts the measurement result strongly, but different values lead to completely different measurements without following a clear pattern. Without knowing the implementation details of the benchmarked code it is not easy to speculate what the technical reason behind this peculiar behavior is.

Special Cases. Two individual benchmarks in our study have parameters that led to special distributions. An example from the

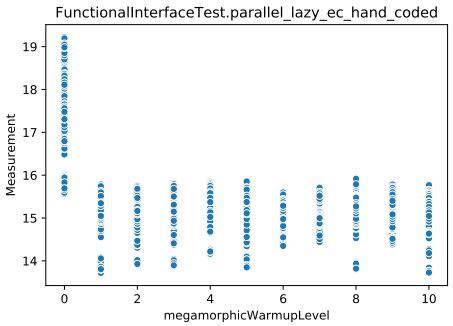

Figure 10: One specific parameter value leads to different benchmark results than all other values (example benchmark from `Eclipse-Collections`, *megamorphicWarmupLevel* is a metric parameter)

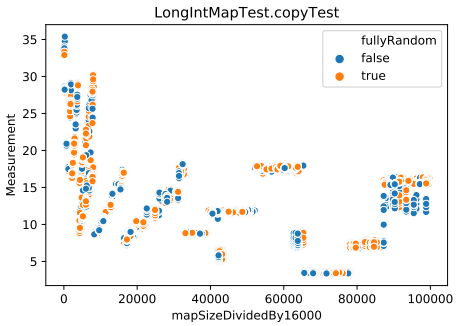

Figure 11: A highly irregular correlation between metric parameter and benchmark result (example benchmark from `Eclipse-Collections`, *mapSizeDividedBy16000* is a metric parameter)

`jdk-microbenchmarks` project is given in Figure 12. In this benchmark, the measured performance has an upper bound that linearly depends on the parameter value, but concrete measurements are uniformly distributed between 0 and this upper bound. Similar to the example in Figure 11, the reason behind this on first glance peculiar behavior is the impact of a second metric parameter.

4.3 Parameter Importance

After analyzing the impact of individual parameters in isolation, we now assess the relative importance of parameters in benchmark classes with multiple parameters. As discussed in Section 3, we use a machine learning based feature selection (FS) approach based on Random Forest ensembles. Hence, we train a machine learning model and derive weights for the different features (parameters) from there. Table 3 presents the relative importance of parameters (features in the trained machine learning model) for each benchmark class. As parameters are defined on class level, we report aggregate results for each class. Further note that the table omits classes with only a single parameter (as the results would be trivial).

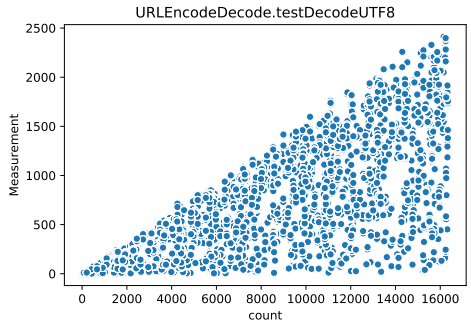

Figure 12: The measurement value has a linear upper bound given by a metric parameter (example benchmark from `jdk-microbenchmarks`, *count* is a metric parameter)

We observe that for most benchmark classes, a single parameter dominates the measurement result (e.g., *count* for `SortedArrayVsHashMapBenchmark` or *qType* for `QueueThroughputBackoffNone`). More generally, workload-related parameters (*count*, *size*, *mapSizeDividedBy64*, etc.) are often dominant. Parameters that relate more to the configuration of the system (e.g., *threshold*, *workers*, *parallelism*, *qCapacity*, etc.) often only have a minor impact on the benchmark result. The benchmarks `ChannelThroughputBackoffNone`, `IntIntMapTest`, `ParallelPerf`, and `QueueOfferPoll` are interesting because two parameters in these classes have high importance to the benchmark result, i.e., multiple parameters where neither clearly dominates the other.

	Class	Parameter	Score
eclipse-collections	LongIntMapTest	**mapSizeDividedBy16000**	**0.8923**
		fullyRandom	0.1077
	IntIntMapTest	**mapSizeDividedBy64**	**0.6841**
		fullyRandom	0.3159
	TroveMapPutTest	**size**	**0.7892**
		isPresized	0.2045
		loadFactor	0.0063
	ChainMapPutTest	**size**	**0.8312**
		isPresized	0.157
		loadFactor	0.0118
RxJava	ParallelPerf	**count**	**0.4711**
		compute	**0.4558**
		parallelism	0.073
	FlowableFlatMapCompletableSyncPerf	**items**	**0.8283**
		maxConcurrency	0.1717
JCTools	QueueThroughputBackoffNone	**qType**	**0.9835**
		qCapacity	0.0462
	SetOps	**occupancy**	**0.5759**
		size	0.3476
		type	0.0766
	QueueOfferPoll	**burstSize**	**0.5257**
		qType	**0.4741**
		qCapacity	0.0002
	ChannelThroughputBackoffNone	**capacity**	**0.6206**
		type	0.3794
Log4j2	ThreadContextBenchmark	**count**	**0.688**
		threadContextMapAlias	0.3123
	ConcurrentAsyncLoggerToFileBenchmark	**queueFullPolicy**	**0.8235**
		asyncLoggerType	0.1766
	SortedArrayVsHashMapBenchmark	**count**	**0.8517**
		length	0.1483
jdk-microbenchmarks	ForkJoinPoolForking	**size**	**0.7916**
		threshold	0.1319
		workers	0.0765
	ForkJoinPoolThresholdAutoSurplus	**size**	**0.9016**
		threshold	0.0814
		workers	0.017
	URLEncodeDecode	**count**	**0.4924**
		maxLength	**0.487**
		mySeed	0.0206

Table 3: Relative importance of parameters on benchmark class level for all benchmarks with two or more parameters.

5 INFERRING THE PERFORMANCE OF UNTESTED PARAMETER VALUES

As we have observed in Section 4, most parameters either do not have an impact on the observed benchmark result at all, or the correlation between parameter value and measurement follows fairly predictable curves. Hence, we speculate that, given sufficient training data, it is possible to infer the performance of untested parameter combinations with high accuracy, without having to actually run the benchmark for this specific parameter combinations. This supports what-if analysis, and allows developers to reason about the performance of their system in specific situations (e.g., when defining low-level throughput or response time guarantees).

To build this predictive model, we use a machine learning model based on Random Forest regression. We are dealing with a regression problem because the model will predict outcome of performance tests given specific JMH parameters, which is a continuous numerical value. Moreover, since each class has its own set of parameters and benchmarks, we will build one model for each benchmark class. We now discuss data preprocessing, the design and training of our model, as well as concrete results demonstrating the feasibility of this approach. As training data, we again use the dataset introduced in Section 3.

5.1 Data Preprocessing and Splitting

In general, the machine learning model accepts only continues and numerical values. Thus, an important preprocessing step involves dealing with the categorical parameters discussed in Section 4.1. On that basis, we perform feature engineering steps before using the features as input to the model. The features are the parameters discussed in Table 3, besides the benchmarks included in each class. Figure 13 shows the preprocessing phases for the class *SetOps*. Those phases can be generalized for all the discussed classes.

Concretely, we apply one-hot encoding [12] to all categorical features. These include all categorical parameters, but also the feature `Benchmark`, which contains the name of the concrete benchmark. In one-hot encoding, if the categorical feature contains n values, the feature is exploded into n new (binary) features. For example, by applying the approach to the `type` parameter (a categorical parameter of `JCTools' SetOps` benchmark), we get two new features (one for each possible value of this parameter) as in the yellow highlighted in Figure 13. The values of the new features will be either 1 if the feature appears in the sample, and 0 otherwise, leading to a binary vector. The same is true for Benchmark which will be replaced by 4 new binary features were constitutes the Benchmark values before the features engineering.

To train and test the model, we follow common machine learning practice and split the dataset for each class into a training set (80% of data) and a test set (20% of data). The test set will be unavailable to the model during training, and will be used to measure the accuracy of the model.

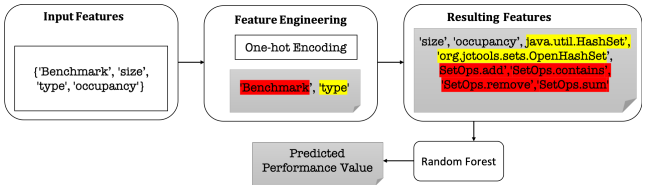

Figure 13: Data preprocessing pipeline to train the machine learning model. This pipeline presents an example for one specific example class (`SetOps`, from the `JCTools` project.

5.2 Predictive Model

After extensive experimentation with different models like Support Vector regression and Polynomial regression,we found that the Random Forest model gives us the best performance for the majority of classes. The Random Forest model is a well-understood machine learning algorithm, and has been applied to various problems in classification and regression [4]. It provides, in general, a good predictive performance, low overfitting, and easy interpretability. Interpretability, in particular, is crucial to our problem domain, as it allows performance engineers to not only predict performance values but also understand how this prediction has been generated. Interpretability is given by the fact that it is straightforward to derive the importance of each feature in a tree decision, i.e., it is easy to see how much each feature contributes to the decision.

Random Forest is an ensemble technique that combines bagging and random feature sub-spacing. It constructs a large collection of regression ensembles of independent decision trees (forests) [4]. Predicting the output performance values involves obtaining an output from each decision tree and then aggregating the results. This is done by taking the average of their predictions to form the final output as shown in Equation 1, where $F_B(x)$ is the output value of sample x in forest B, and $T_b(x)$ is the output value of sample x in tree b.

$$F_B(x) = \frac{1}{B}\sum_{b=1}^{B} T_b(x) \tag{1}$$

5.3 Model Evaluation

For model evaluation, the validation data (a randomly sampled 20% of data) will be used. Hence, to test the trained model quality, the input vector of all data in the validation set will be passed to the model, and the predicted outcome will be compared with the true (measured) performance. Concretely, we use the R-squared (R^2) metric to measure the overall model performance. R^2 is a statistical measure that represents the proportion of the variance for a dependent variable (output performance value) that is explained by an independent variable or variables in a regression model (input features). The R^2 value is always between 0 and 100%. The higher the R^2 is, the better the model fits the data. A formal definition is provided in Equation 2, where SS_{RES} is the residual sum of squared errors of our regression model, SS_{TOT}) is the total sum of squared errors, y_i is the true output value,$\hat{y}_i$ the predicted output, and $\bar{y}$ the average across all the output.

$$R^2 = 1 - \frac{SS_{RES}}{SS_{TOT}} = 1 - \frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - \bar{y})^2} \tag{2}$$

In Table 4, the R^2 scores for all models of all tested benchmark classes are depicted. Three classes in `Log4J2` (`ConcurrentAsyncLoggerToFileBenchmark`, `AsyncAppenderLog4j2Benchmark`, and `MDCFilterBenchmark`) were skipped, as the ranges of all parameters in these classes were below a threshold of 50 different samples (different parameter values). Hence, no Random Forest model could be trained for these classes.

Project	Class	R^2
eclipse-collections	LongIntMapTest	96%
	FunctionalInterfaceTest	94%
	IntIntMapTest	93%
	TroveMapPutTest	97%
	ChainMapPutTest	96%
RxJava	ParallelPerf	97%
	RangePerf	98%
	FlowableFlatMapCompletableSyncPerf	99%
	ObservableFlatMapMaybePerf	99%
	ObservableConcatMapSinglePerf	98%
	FlattenRangePerf	97%
JCTools	QueueThroughputBackoffNone	92%
	SetOps	98%
	SpscChannelThroughputTest	99%
	QueueOfferPoll	97%
	ChannelThroughputBackoffNone	11%
Log4j2	ThreadContextBenchmark	98%
	SortedArrayVsHashMapBenchmark	99%
jdk-microbenchmarks	ForkJoinPoolForking	97%
	ForkJoinPoolThresholdAutoSurplus	98%
	ArrayCopyUnalignedBoth	98%
	Limit	99%
	URLEncodeDecode	99%

Table 4: Accuracy of the trained models for all benchmark classes in the experiment.

We observe that the trained models perform very well for the majority of classes with R^2 scores ranging between 92% and 99%. However, the model for one example (`JCTools' ChannelThroughputBackoffNone` class) has a very low accuracy of 11%. This class only has two parameters (one metric and one categorical), which both constitute asymptotic bounds. Due to this problem, the model struggles to detect patterns to learn from the training data.

5.4 Discussion

Our experiment demonstrates that given sufficient training data, it is very feasible to predict the benchmarking outcome of a JMH benchmark using an untested combination of parameter values with very high accuracy. However, it should be noted that our models have been trained on a significant amount of data collected from a single version of the study subjects. While it is likely that the trained models can be re-used for other (future) versions of the system, the prediction accuracy will degrade depending on how much code has changed between the training version and the version the model is being used on. Our proposed model can be extended to support online learning [13], for instance by explicitly including the version that a specific training data item refers to as an additional feature and collecting new data on an ongoing basis. However, if new versions include changes to the benchmarks

themselves (e.g., adding or removing benchmarks or parameters), a complete re-training of the models for the changed benchmark classes will be required.

It should be noted that this prediction approach is not *per se* suitable or intended to replace actually testing performance: training a model on one version of a software system and using the predictor instead of executing the test for future version would evidently counteract the reason why we benchmark in the first place. However, we envision that a predictor such as the one demonstrated here can be useful when developers are actually interested in the specific test outcome for concrete combinations parameters (e.g., in what-if analysis, or when defining low-level throughput or response time guarantees). Especially if a wide range of (combinations of) parameter values should be assessed, or if this analysis needs to be done frequently for different values, training and using a predictor may be faster than running the benchmark suite on demand for all combinations that are of interest. However, of course this needs to be judged on a case-by-case basis.

In our study, we have observed that collecting sufficient data to train high-accuracy models required between multiple hours and one day on a single cloud instance for a single benchmark class. Hence, we argue that collecting sufficient data in an industrial or open source project is computationally expensive but feasible, as long as it is sufficient to train models for a selection of benchmark classes. However, practitioners need to take care to take existing guidelines on robust Java performance testing [10, 17, 26] into account, otherwise noise from the execution environment (e.g., performance variations in cloud instances) is likely to negatively impact prediction accuracy.

6 RELATED WORK

This research is based on and related to existing work in the areas of performance engineering, performance testing, system benchmarks and workload selection, and machine learning in performance engineering. **Performance Engineering (PE)** represents an entire span of software engineering activities and related analyses used throughout the software development cycle, which are directed to meeting performance requirements [30]. There are two main approach for PE. *i) Measurement-based* PE applies testing, diagnosis and tuning late in the development cycle, when the system-under-test can be run and measured [29]. Contrary, *ii) model-based* PE creates performance models early in the development cycle and uses quantitative results from these models to adjust the architecture and design with the purpose of meeting performance requirements. Our work falls under the umbrella of measurement-based PE.

Performance testing is one concrete form of measurement-based performance engineering [30]. Researchers as well as practitioners commonly use benchmarks developed by either the research community or standardization bodies, such as SPEC, to evaluate and compare the performance of systems. The correct interpretation of performance test results requires knowledge of the synthetic workload to determine how well it represents the envisioned real-world workload of diverse applications. Factors that influence these results include the characteristics of the system-under-test, the procedures used to carry out the tests, and the performance metrics the benchmark generates. A well-known example of a widely used (albeit now outdated) benchmark is TPC-W [9], a benchmark for e-commerce systems. A more recent well-known example is YCSB [6] (the Yahoo Cloud Serving Benchmark), which has the goal of facilitating performance comparison of cloud data serving systems. Hence, a core benchmark is defined and then executed against a variety of cloud storage systems, including Cassandra, HBase, Yahoo!'s PNUTS, and a simple sharded MySQL database. Finally, the *Standard Performance Evaluation Corporation (SPEC)* CPU2017 benchmark suite has recently been analyzed by Limaye and Adegbija with respect to different metrics such as execution performance, in addition a comparison between CPU 2006 and CPU 2017 workloads [22]. Additionally, a robust analysis has been conducted that enables researchers to intelligently choose from a subset of the CPU2017 suite that accurately represents the whole suite, in order to reduce execution times.

Code-level performance tests, or microbenchmarks, have only recently started to gain attention in the PE research community. Leitner and Bezemer have studied the performance test for 111 open source java projects [19], and report that performance tests form only a small portion of the test suite, are rarely updated, and are usually maintained by a small group of core project developers. Georges et al. present a survey of existing Java performance evaluation methodologies and discuss the importance of statistically rigorous data analysis for dealing with non-determinism in evaluation methodologies [10]. Recent work by Stefan et al. indicates that microbenchmarks often do not find useful regressions, and are primarily used to evaluate design decisions [28]. One common problem in microbenchmarking is that writing robust microbenchmarks is complex and requires in-depth knowledge from developers [18]. To reduce this friction, Costa et al. provide tooling to identify bad microbenchmarking practices in through static analyses [7]. Laaber et al. proposed an approach to dynamically adapt software microbenchmark configurations to stop their execution once their result is stable [18]. Another set of problems in microbenchmarking are long execution times [16] and unreliable results [17]. One of the goals of our study is to examine one of the key culprits of long execution times (extensive parameterizations), hence reducing the time spent on executing microbenchmarks.

Our work also uses concepts of **artificial intelligence (AI)** applied to PE. This follows a general trend of applying machine learning (ML) and AI concepts to various software engineering topics. For instance, a novel hybrid algorithm based on optimization and machine-learning approaches was used by Kaur et al. to efficiently detect code smells [14]. Code smells are sub-optimal implementation choices applied by developers that have a negative effect on, among others, the change-proneness of the affected classes [5]. Another area of software engineering that has seen significant interest in ML and AI is defect prediction. Defect prediction approaches can be divided into approaches that use supervised learning, and ones based on unsupervised learning. Supervised defect prediction models need training data and test data. Most commonly, training data is collected from the same or very similar projects within the same organization. Cross-project defect prediction is a subfield that attempts to learn characteristics of defect-prone classes

across entire ecosystems [15]. An example of such an approach has been provided by Deng et al., who employed deep learning to parse abstract syntax trees and learn defect proneness across heterogeneous systems [8]. Another area of software engineering that nowadays heavily relies on AI methods is the estimation of software development efforts. As there rarely is sufficient data available in the initial stage of the project life cycle, AI approaches such as artificial neural networks or the dragonfly algorithm are used to build more accurate estimation models [2]. Finally, other usage includes code search [11, 23] or the usage of AI to predict node failures in cloud computing [21]. In this work, we apply similar principles and algorithms to predict the measurement result of software microbenchmarks with untested parameterizations.

7 CONCLUSIONS

In this paper, we carried out an experimental study to investigate parametrizations of JMH benchmarks. We statistically investigated the impact of the individual parameters on the performance results, and classified the impact into various various classes. We found that 40% of JMH parameters have no observable impact on the measurement results. Furthermore, we performed feature selection to quantify the importance of different parameters for benchmark classes with multiple parameters, and found that in most cases a single parameter dominates the benchmark result. Finally, we developed and discussed a machine learning model based on Random Forest ensembles for each benchmark class to predict the measurement for unseen parameterizations. We showed that, given sufficient data, highly accurate predictions can be generated for all but one benchmark class.

ACKNOWLEDGEMENTS

This work received financial support from the Swedish Research Council VR under grant number 2018-04127 (Developer-Targeted Performance Engineering for Immersed Release and Software Engineers).

REFERENCES

[1] S. Ahmed, K. K. Ghosh, P. K. Singh, Z. W. Geem, and R. Sarkar. Hybrid of harmony search algorithm and ring theory-based evolutionary algorithm for feature selection. *IEEE Access*, 8:102629–102645, 2020.

[2] N. K. L. Alhammad, E. Alzaghoul, F. A. Alzaghoul, and M. Akour. Evolutionary neural network classifiers for software effort estimation. *Int. J. Comput. Aided Eng. Technol.*, 12(4):495–512, 2020.

[3] S. Baltes and P. Ralph. Sampling in software engineering research: A critical review and guidelines, 2020.

[4] L. Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[5] G. Catolino, F. Palomba, F. A. Fontana, A. D. Lucia, A. Zaidman, and F. Ferrucci. Improving change prediction models with code smell-related information. *Empirical Software Engineering*, 25(1):49–95, 2020.

[6] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[7] D. E. Damasceno Costa, C. Bezemer, P. Leitner, and A. Andrzejak. What's wrong with my benchmark results? studying bad practices in JMH benchmarks. *IEEE Transactions on Software Engineering*, 2019. To appear.

[8] J. Deng, L. Lu, S. Qiu, and Y. Ou. A suitable ast node granularity and multi-kernel transfer convolutional neural network for cross-project defect prediction. *IEEE Access*, 8:66647–66661, 2020.

[9] D. F. Garcia and J. Garcia. TPC-W e-commerce benchmark evaluation. *Computer*, 36(2):42–48, Feb 2003.

[10] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous java performance evaluation. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, OOPSLA '07, page 57–76, New York, NY, USA, 2007. Association for Computing Machinery.

[11] X. Gu, H. Zhang, and S. Kim. Deep code search. In M. Chaudron, I. Crnkovic, M. Chechik, and M. Harman, editors, *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 933–944. ACM, 2018.

[12] J. T. Hancock and T. M. Khoshgoftaar. Survey on categorical data for neural networks. *J. Big Data*, 7(1):28, 2020.

[13] S. C. H. Hoi, D. Sahoo, J. Lu, and P. Zhao. Online learning: A comprehensive survey. *CoRR*, abs/1802.02871, 2018.

[14] A. Kaur, S. Jain, and S. Goel. SP-J48: a novel optimization and machine-learning-based approach for solving complex problems: special application in software engineering for detecting code smells. *Neural Computing and Applications*, 32(11):7009–7027, 2020.

[15] M. Kondo, C. Bezemer, Y. Kamei, A. E. Hassan, and O. Mizuno. The impact of feature reduction techniques on defect prediction models. *Empirical Software Engineering*, 24(4):1925–1963, 2019.

[16] C. Laaber and P. Leitner. An evaluation of open-source software microbenchmark suites for continuous performance assessment. In A. Zaidman, Y. Kamei, and E. Hill, editors, *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*, pages 119–130. ACM, 2018.

[17] C. Laaber, J. Scheuner, and P. Leitner. Software microbenchmarking in the cloud. how bad is it really? *Empirical Software Engineering*, 24(4):2469–2508, 2019.

[18] C. Laaber, S. Würsten, H. C. Gall, and P. Leitner. Dynamically reconfiguring software microbenchmarks: Reducing execution time without sacrificing result quality. In *Proceedings of the 2020 ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. IEEE, 2020. To appear.

[19] P. Leitner and C.-P. Bezemer. An exploratory study of the state of practice of performance testing in java-based open source projects. In *Proceedings of the 8th ACM/SPEC International Conference on Performance Engineering (ICPE)*, 2017.

[20] P. Leitner and J. Cito. Patterns in the chaos - a study of performance variation and predictability in public iaas clouds. *ACM Transactions on Internet Technology*, 16(3):15:1–15:23, apr 2016.

[21] Y. Li, Z. M. J. Jiang, H. Li, A. E. Hassan, C. He, R. Huang, Z. Zeng, M. Wang, and P. Chen. Predicting node failures in an ultra-large-scale cloud computing platform: An aiops solution. *ACM Trans. Softw. Eng. Methodol.*, 29(2):13:1–13:24, 2020.

[22] A. Limaye and T. Adegbija. A workload characterization of the spec cpu2017 benchmark suite. In *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 149–158, 2018.

[23] C. Liu, X. Xia, D. Lo, Z. Liu, A. E. Hassan, and S. Li. Simplifying deep-learning-based model for code search. *CoRR*, abs/2005.14373, 2020.

[24] H. Liu, H. Motoda, R. Setiono, and Z. Zhao. Feature selection: An ever evolving frontier in data mining. In H. Liu, H. Motoda, R. Setiono, and Z. Zhao, editors, *Proceedings of the Fourth International Workshop on Feature Selection in Data Mining, FSDM, held at PAKDD 2010, Hyderabad, India, June 21st, 2010*, volume 10 of *JMLR Proceedings*, pages 4–13. JMLR.org, 2010.

[25] H. Liu and Z. Zhao. Manipulating data and dimension reduction methods: Feature selection. In R. A. Meyers, editor, *Encyclopedia of Complexity and Systems Science*, pages 5348–5359. Springer, 2009.

[26] A. V. Papadopoulos, L. Versluis, A. Bauer, N. Herbst, J. Von Kistowski, A. Ali-eldin, C. Abad, J. N. Amaral, P. Tůma, and A. Iosup. Methodological principles for reproducible performance evaluation in cloud computing. *IEEE Transactions on Software Engineering*, pages 1–1, 2019.

[27] H. Samoaa and P. Leitner. An Exploratory Study of the Impact of Parameterization on JMH Measurement Results in Open-Source Projects [Replication Package]. Zenodo, Sept. 2020. https://doi.org/10.5281/zenodo.4013943.

[28] P. Stefan, V. Horký, L. Bulej, and P. Tuma. Unit testing performance in java projects: Are we there yet? In W. Binder, V. Cortellessa, A. Koziolek, E. Smirni, and M. Poess, editors, *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering, ICPE 2017, L'Aquila, Italy, April 22-26, 2017*, pages 401–412. ACM, 2017.

[29] E. J. Weyuker and F. I. Vokolos. Experience with performance testing of software systems: Issues, an approach, and case study. *IEEE Trans. Softw. Eng.*, 26(12):1147–1156, Dec. 2000.

[30] C. M. Woodside, G. Franks, and D. C. Petriu. The future of software performance engineering. In L. C. Briand and A. L. Wolf, editors, *International Conference on Software Engineering, ISCE 2007, Workshop on the Future of Software Engineering, FOSE 2007, May 23-25, 2007, Minneapolis, MN, USA*, pages 171–187. IEEE Computer Society, 2007.

[31] B. Xue, M. Zhang, W. N. Browne, and X. Yao. A survey on evolutionary computation approaches to feature selection. *IEEE Trans. Evol. Comput.*, 20(4):606–626, 2016.