IoPV: On Inconsistent Option Performance Variations

Jinfu Chen*
jinfuchen@whu.edu.cn
Wuhan University
Wuhan, Hubei, China

Zishuo Ding
zishuo.ding@uwaterloo.ca
University of Waterloo
Waterloo, ON, Canada

Yiming Tang
yxtvse@rit.edu
Rochester Institute of Technology
Rochester, NY, USA

Mohammed Sayagh
Mohammed.Sayagh@etsmtl.ca
ETS (Quebec University)
Montreal, QC, Canada

Heng Li
heng.li@polymtl.ca
Polytechnique Montréal
Montreal, QC, Canada

Bram Adams
bram.adams@queensu.ca
Queen's University
Kingston, ON, Canada

Weiyi Shang
wshang@uwaterloo.ca
University of Waterloo
Waterloo, ON, Canada

ABSTRACT

Maintaining a good performance of a software system is a primordial task when evolving a software system. The performance regression issues are among the dominant problems that large software systems face. In addition, these large systems tend to be highly configurable, which allows users to change the behaviour of these systems by simply altering the values of certain configuration options. However, such flexibility comes with a cost. Such software systems suffer throughout their evolution from what we refer to as "Inconsistent Option Performance Variation" (***IoPV***). An *IoPV* indicates, for a given commit, that the performance regression or improvement of different values of the same configuration option is inconsistent compared to the prior commit. For instance, a new change might not suffer from any performance regression under the default configuration (i.e., when all the options are set to their default values), while altering one option's value manifests a regression, which we refer to as a hidden regression as it is not manifested under the default configuration. Similarly, when developers improve the performance of their systems, performance regression might be manifested under a subset of the existing configurations. Unfortunately, such hidden regressions are harmful as they can go unseen to the production environment. In this paper, we first quantify how prevalent (in)consistent performance regression or improvement is among the values of an option. In particular, we study over 803 *Hadoop* and 502 *Cassandra* commits, for which we execute a total of 4,902 and 4,197 tests, respectively, amounting to 12,536 machine hours of testing. We observe that *IoPV* is a common problem that is difficult to manually predict. 69% and 93% of the *Hadoop* and *Cassandra* commits have at least one configuration that hides a performance regression. Worse, most of the commits have different options or tests leading to *IoPV* and hiding performance regressions. Therefore, we propose a prediction model that identifies whether a given combination of commit, test, and option (***CTO***) manifests an *IoPV*. Our evaluation for different models shows that random forest is the best performing classifier, with a median AUC of 0.91 and 0.82 for *Hadoop* and *Cassandra*, respectively. Our paper defines and provides scientific evidence about the *IoPV* problem and its prevalence, which can be explored by future work. In addition, we provide an initial machine learning model for predicting *IoPV*.

*Corresponding author.

ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA

ACM ISBN 979-8-4007-0327-0/23/12…$15.00
https://doi.org/10.1145/3611643.3616319

CCS CONCEPTS

• **Software and its engineering → Software performance**.

KEYWORDS

Software performance, Performance variation, Configurable software systems

ACM Reference Format:
Jinfu Chen, Zishuo Ding, Yiming Tang, Mohammed Sayagh, Heng Li, Bram Adams, and Weiyi Shang. 2023. IoPV: On Inconsistent Option Performance Variations. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3611643.3616319

1 INTRODUCTION

Modern large-scale software systems tend to have a large number of configuration options, which can hide performance issues. These options are used to customize the behaviour of a software system without changing its source code. Although these options add flexibility to a software system, they make testing software performance a challenging task. For example, in theory, one has to run 2^{10} tests for a software system with just 10 boolean configuration options, while a highly configurable software system such as *Hadoop* can have as many as 365 available options [1]. While there are constraints between configuration options, bringing down the total number of configurations in practice, this still amounts to a too large set of configurations to test exhaustively, especially for (long-running) performance tests.

Prior study [2] found that more than 50% of performance bugs are related to misconfiguration, implying that configuration tuning is crucial for system performance. Most of the configurations directly impact performance, such as the timeout-family configurations like *ipc.client.connect.timeout* that directly affects latency in *Hadoop*. For another example, a prior study [3] found that the configuration option *native_transport_max_threads* in the *Cassandra* project can lead to up to 7.3X latency (i.e., performance regression) under two different configuration settings, i.e., two different values for the option *native_transport_max_threads*.

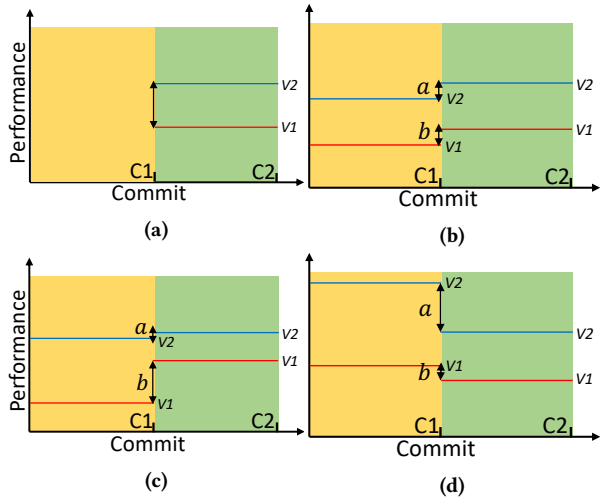


Figure 1: The definition of *IoPV*: (a) Approaches that do not consider the historical evaluation, (b) An option with a consistent performance variation (a=b), (c) An option with an inconsistent performance variation (a!=b) C2 has a performance regression compared to C1, and (d) An option with an inconsistent performance variation (a!=b) C2 has a performance improvement compared to C1. V1 and V2 are two different values of the same configuration option. C1 and C2 are two revisions. A smaller performance metric value (e.g., CPU usage) indicates a better performance.

Performance-related configuration is a non-trivial problem since configuration options are hard to understand and poorly documented [3]. In particular, the default configuration may have no performance regression perceived by the end-user; however, user-specific configuration settings may lead to a significant performance regression. For example, a real-life performance issue is shown in MySQL under the configuration option *query_cache_type* in versions 5.0.44, 5.0.84, and 5.1.38. Such an issue is reported in the MySQL bug tracking system with bug ID MySQL-47529 [4]. The option *query_cache_type* is used to set the query cache type. The possible values of the option *query_cache_type* include OFF, ON, and DEMAND, defaulting to OFF. In general, turning on *query_cache_type* leads to somewhat better performance, since MySQL can cache results in memory. In this bug, we see that with setting *query_cache_type* to OFF, the performance among the three versions remains the same. However, when turning on *query_cache_type*, the performance in version 5.0.84 is worse than in versions 5.0.44 and 5.1.38.

Traditionally, prior work studied the difference in system performance caused by *different* values of the *same* option, without considering how the performance impact of an option evolves due to code changes [1]. For instance, traditional approaches compare different values of a configuration option based on their raw performance values [5, 6, 7], as illustrated in Figure 1a. However, such comparison is subjective as the option's value *V2* with worse performance might not necessarily be problematic, but might, as an example, just enable the execution of some extra features. Conversely, even if an option's value has a good performance compared to other values, those differences in performance might start to vary when comparing to the performance of the same option value in the prior commit. Normally, one would expect to see the situation in Figure 1b, which shows that both option values have a consistent variation in performance, in this case, a similar increase (regression) in the performance metric. In reality, one can observe cases such as in the example in Figure 1c, where *V1* still shows better performance compared to *V2* after commit C2, but it faces a significantly larger performance regression compared to the prior commit than *V2*. In Figure 1d, the *V2* value still has a worse performance after the new commit, but its performance improved much more significantly compared to the prior commit than *V1*.

Therefore, different values of an option can have an inconsistent variation in terms of performance compared to the prior commit, which we refer to as **Inconsistent Option Performance Variation (a.k.a, *IoPV*)**. *IoPV* is a difficult and complicated issue for developers to identify and fix. We find there is a lack of reported *IoPV* issues in the issue tracking systems as developers all too often do not test different values of options across versions. However, the *IoPV* might be problematic as it can hide a performance regression that is manifested only when altering configuration options. Such regressions can unfortunately go as unseen to the production environment. The *IoPV* may directly affect the user experience, increase the resources cost of the system and lead to reputational and financial repercussions.

In this paper, we perform a case study on two large-scale open-source software systems: *Hadoop* and *Cassandra*. We first conduct a preliminary study to quantify the prevalence of *IoPV* in practice. We observe that 81% of the commits have at least one option manifesting an *IoPV* issue. We also observe that manually identifying such issues is challenging, as commits do not share the same options that manifest an *IoPV*. That motivates us to propose an automated model that predicts if the combination of a Commit, a Test, and an Option (***CTO***) would exhibit an *IoPV* issue. We evaluate our prediction model using the following two research questions:

RQ1. Can we accurately learn *IoPV* issues in the studied systems? Our prediction model reaches an area under the receiver operating characteristic curve (AUC) up to 0.93 and 0.90 for predicting *IoPV* for *Hadoop* and *Cassandra*, respectively. AUC measures our models' ability to discriminate the *CTO* cases into *IoPV* and non-*IoPV* cases. We observe that random forest is the most performing model for four and three out of five performance measures (i.e., response time, CPU, memory, I/O Read, and I/O write) for *Cassandra* and *Hadoop*, respectively.

RQ2. What are the most important metrics for predicting *IoPV* issues? We observe that all four dimensions of metrics considered in our study, namely the code structure, code change, code token, and configuration options metrics, have a statistically significant impact in predicting *IoPV*. The dimensions that are related to the configuration options and the tokens of the changed code are the most important dimensions for both case studies.

2 BACKGROUND

Software configuration is a mechanism used to customize the behaviour of a software system without changing the source code. The configuration **options** are often stored in configuration files as a set of key - value pairs, where the key represents an option's name and the **value** represents a default or user-chosen value for that

option. We define a **configuration** as one particular assignment of a value to all existing options. Table 1 lists the definition of these terms. For example, *A=1* and *B=2* is one possible configuration for a software system with the two integer options *A* and *B*. Configuration options enable users to adapt the execution of their software systems by simply modifying the values of certain configuration options, without re-compilation. For example, a user can change the directory that stores the cache for *Cassandra* by changing the value of the *saved_caches_directory* configuration option.

Table 1: Our definition of configuration, option, and value.

Term	Definition	Example
Option	A typed, configurable item that allows users to set different values.	A
Value	A specific assignment of a value for an option.	$A = 1$
Configuration	An assignment of values to all options by a user.	$A = 1; B = 2$

Although configuration introduces large flexibility for users, considering all the possible configurations during testing is impossible. A software system with 10 boolean configuration options requires testing 2^{10} configurations. In fact, configuration problems are among the dominant problems in software engineering [1, 8].

In particular, a software system can suffer from what we refer to as **Inconsistent Option Performance Variation** (a.k.a, *IoPV*). This occurs when, for a given commit *C*, the performance of a subset of an option's values evolved differently relative to their performance in the commit prior to *C*. Considering the example in Figure 1, when comparing the raw performance of the two option values *V1* and *V2* (Figure 1a), we observe that *V1* shows a better performance than *V2*. However, that might not be problematic as *V2* might just enable an extra feature, such as logging a transaction. In fact, Figure 1c shows that even if *V2* does not show any significant performance variation from the prior commit, *V1* suffers from a performance regression. Similarly, in Figure 1d the performance of *V2* is improved compared to the prior commit, while that improvement does not manifest under option value *V1*. The *IoPV* may directly affect the user experience, increase the resources cost of the system and lead to reputational repercussions. A performance variation is calculated as the difference between the performance variation of each option's value after and before each commit, which is illustrated in Figure 1 by "$a - b$".

3 DATA COLLECTION

In this section, we present our subject systems and our approach to collect performance regressions and configuration data.

3.1 Subject Systems

In this paper, we consider *Hadoop* [9] and *Cassandra* as two subject systems. We choose these two subject systems due to the following reasons: (1) their performance is critical for the users, (2) the two systems are highly configurable, (3) the two systems have been studied in prior research on software performance [10], and (4) we are familiar with these two systems. The overview of our subject systems is shown in Table 2.

3.2 Data Gathering

We follow the approach summarized in Figure 2 to collect data.

Table 2: Our studied dataset.

Subjects	# Studied Releases	Last release	# Commits	# Configuration Options	# Tests
Hadoop	7	2.7.3	803	365	1,853
Cassandra	5	3.0.15	502	162	369

3.2.1 Filtering Commits. Since we study performance variation across different versions of a software system, we only consider source code related changes. *Hadoop* and *Cassandra* are both Java systems. Therefore, we filter out commits without any *java* source code changes. Furthermore, developers can commit multiple changes toward fixing the same issue, which is defined in the issue tracking system. As *Hadoop* and *Cassandra* use JIRA as their issue tracking system and have an explicit mapping between commits and issues, we use the issue ID mentioned in the commit messages to identify the commits that belong to the same issue. If multiple commits are associated with the same issue, we only consider the last commit. This is important as developers can initially introduce a regression but then fix it before releasing the code changes related to the issue.

3.2.2 Extracting Options. In the second step, we extract configuration options and their corresponding values for each subject system (i.e., 365 and 162 configuration options in the last studied releases of *Hadoop* and *Cassandra*, respectively). We obtain option names and default values by crawling the documentation of *Hadoop* [11] and *Cassandra* [12], by extracting the configuration file that is shipped with the project's releases. Finally, we manually classified the extracted options based on their expected data types (e.g., Boolean when the default value is *TRUE* or *FALSE*).

3.2.3 Identifying Impacted Tests. We automatically create a mapping between the changed source code in each commit and the existing unit tests. We derive such commit-test mapping based on the automatically generated method-level code coverage results, similar to a prior study by Chen et al. [13]. For each commit, we use Eclipse JDT to automatically add logging instrumentation to each method that will print log messages that indicate the execution of the method at runtime. We then run each test for the commit. A test is considered impacted by the commit if any instrumented logging is output. Afterwards, we only run the tests that execute the changed source code for a given commit since executing all the existing tests of a software system for each commit and each possible configuration is practically infeasible. In addition, running those tests that are not impacted by the code change of a commit is not likely to detect performance variations (regressions or improvements under some values of an option).

3.2.4 Identifying Impacted Options. Similar to identifying impacted tests, we would like to identify which configuration options are impacted while running the tests. The configuration option values are accessed using *getters*, based on our research of two well-known projects, *Hadoop* and *Cassandra* (e.g., *DatabaseDescriptor.java* [14] to access the *Cassandra*'s options). We keep track of method invocations to the *getters* that involve configuration options. If such method invocations occur during test execution, the relevant options are considered impacted by the commit. Note that we only execute the tests that cover the changed methods in each commit.

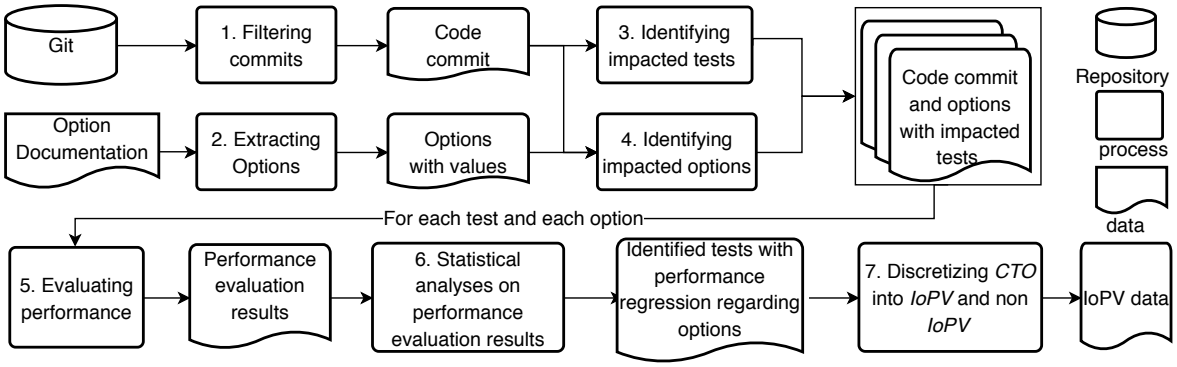


Figure 2: An overview of our approach to collect data. *CTO* is a combination of a Commit, a Test, and an Option.

3.2.5 Evaluating Performance. After obtaining which tests and which options are impacted by each commit, we exercise the test on each commit and its parent commit (i.e., the previous commit) to evaluate their respective performances. We first execute each test with all the configuration options set at their default values. Then, we alter the value of one configuration option at a time. For the configuration options with boolean values, we alter the configuration option to the value that is not the default. For example, if the default value is *TRUE*, we would alter the value to be *FALSE*. For the numeric type option, we alter the configuration option once to the value that is double the default value and once to half of the default value. For example, if a configuration option has a default value of 100, we would run the test altering the value to 200, then run the same test altering the value to 50. For enumeration-typed options, we alter to each of the possible values.

Our performance evaluation environment uses the Google Compute Engine with 8GB memory and 16 cores CPU. In order to generate statistically rigorous performance results, we adopt the practice of repetitive measurements [15] to evaluate performance. Conservatively, we executed each test 30 times independently, which is larger than prior work that repeats a test only 5 to 20 times [16, 17, 18]. To measure the performance that is associated with each test, we use a performance monitoring tool named *psutil* [19] (Python system and process utilities). *Psutil* can capture detailed performance metrics and has been widely used in prior research [20, 21]. We collect both domain level and physical level performance metrics. In our execution, we collect five performance metrics during the execution, i.e., response time, CPU usage, memory usage, I/O read and I/O write. To minimize the performance noise, we first control the execution environment strictly in each instance, i.e, every instance is with the same hardware setup and every instance only runs the small-scale test process. Second, all the performance measures related to a given version of a given project were done in a limited time period, so the variation within that scope should not have been impacted by VM provisioning/contention. Finally, we repeat the unit test 31 times. The first time of execution is to warm up the junit process. The remaining 30 times of execution are taken into consideration in our statistical analysis.

3.2.6 Statistical Analyses on Performance Evaluation Results. To identify the *IoPV*, we statistically compare the performance of a given test and a configuration option value before and after each commit using the Mann-Whitney U test [22] (i.e., $\alpha = 0.05$) and Cliff's delta [23], which measures the magnitude of performance regressions. We choose Mann-Whitney U test since it does not have any assumption on the distribution of the data. Researchers have shown that reporting only the statistical significance may lead to erroneous results (i.e., if the sample size is very large, p-value can indicate statistical significance even if the difference is trivial). Thus, we also use Cliff's delta to quantify the magnitude of the differences (a.k.a., effect sizes). Cliff's delta measures the effect size statistically and has been used in prior engineering studies [24, 25, 26]. Cliff's delta ranges from -1 to +1, where a value of 0 indicates two identical distributions.

For each combination of commit, test and option value, we obtain a Cliff's delta value. We then calculate the differences between the maximum and minimum Cliff's delta for each option's different values, which the next subsection uses to categorize a combination of commit, test and option as *IoPV* or non-*IoPV*.

We also consider a test to be a performance regression when the value of the effect size is positive and has either medium ($0.33 < \textit{Cliff's delta} \leqslant 0.474$) or large ($0.474 < \textit{Cliff's delta}$) magnitude. On the other hand, we consider a test to manifest a performance improvement if the value of the effect size is negative and has a medium ($-0.33 > \textit{Cliff's delta} \geqslant -0.474$) or large ($-0.474 > \textit{Cliff's delta}$) magnitude. Note that we perform this statistical analysis for each performance metric (i.e., response time, CPU usage, memory usage, I/O read and I/O write) separately. For example, a commit may show a CPU regression or improvement, but not show any difference for the response time.

3.2.7 Discretizing CTO *into* IoPV *and non-*IoPV. In the final step, we categorize each commit, test, and option (*CTO*) into *IoPV* or non-*IoPV* based on an automatically determined threshold. Our intuition is that the maximum difference values (a-b in Figure 1) would be concentrated in either small values (i.e., when adjusting an option does not make a difference) or large values (i.e., when adjusting an option does make a difference), which is demonstrated in Figure 1. Specifically, we use Ckmeans.1d.dp [27], a one-dimensional clustering algorithm, to find a threshold that separates the maximum difference values of all *CTO*s into two groups, i.e., *IoPV* and non-*IoPV*. Note that the option variation ranges between 0, when there is no variation, and 2, when the effect size (cf. Section 3.2.6) is 1 for one option value and -1 for another value of the same option.

To further investigate our collected data, we manually examine the issue reports related to each commit with an *IoPV* issue (based on our quantitative analysis of our preliminary analysis). In particular, we first extract the issue id from the commit message. Then we write a script to search in the issue report for the name of each configuration option as the keywords. Finally, if there are commits or issue reports containing the searched keywords, we manually examine whether the issue is reporting any performance regression. We observe that out of 1,155 and 2,275 CTOs without regression under the default option value but with regression under other option values, only 9 and 1 *CTO* in *Hadoop* and *Cassandra*, respectively, have reported configuration-related performance regression in the commit messages or issue reports. Such results show that performance regression may be hidden from developers since performance regression such as CPU or memory regression do not have a direct impact on developers, but rather on users. In addition, we infer that developers do not test other configuration option values except the default ones.

4 PRELIMINARY STUDY

Through this preliminary study, we quantify the existence of the *IoPV* problem in large and highly configurable software systems, as well as how difficult it is to identify the *IoPV*. This preliminary analysis will also motivate the need for an approach that automatically identifies the *IoPV*.

PQ1. Are *IoPV* issues common in the studied systems?

Motivation. The goal of this preliminary research question is to quantify and provide scientific evidence on how often a configuration option can suffer from instances of the *IoPV* issue. While a new code change might not show any performance regression under the default configuration, another configuration can hide a performance regression that can go as unseen to the production environment. This is an important problem as performance issues often lead to serious monetary losses [28]. Similarly, a configuration improvement might not be manifested under all the configurations. One may only compare different values of a given configuration option rather than identifying the *IoPV* problem. However, only comparing different values of a given option cannot know whether the performance variation is due to the configuration error or other reasons, such as a new feature.

Approach. We first collect performance measurements for each ***CTO*** (combination of a commit, test and option) and label each *CTO* as *IoPV* or a non-*IoPV*. Then, we identify for each commit and unit test the number of configurations under which the performance is statistically significantly worse (a.k.a., performance regression) or better (a.k.a., performance improvement) than the performance of the same test and configuration in the prior commit. Finally, we quantify for each commit the number of tests that show a performance regression or a performance improvement under just a subset of the existing configurations. In the studied *Hadoop* and *Cassandra* releases, there are 4,902 and 4,197 *CTO*, respectively.

We also evaluate whether the interactions between the combinations of configuration options would influence the manifestation of *IoPV* issues. Unfortunately, measuring all the possible interaction of configuration options is practically unfeasible. An estimation of what would be the cost of evaluating all the combination of just 2-wise configuration options sums up to 377,556 and 963,788 test executions for *Hadoop* and *Cassandra*, respectively, which would take more than 16 machine years to finish the experiment. Instead, we select a statistically representative (95% confidence level and 5% confidence interval) random sample of 384 t-wise *CTO* from the population of all possible t-wise (t ranges from two to five in our study) combinations of options, which comprises 32,536,088 and 91,864,800 t-wise combinations for *Hadoop* and *Cassandra*, respectively. Then, we measure the performance of each of our t-wise based *CTO* similar to single-option based *CTO*.

Intuitively, if each occurrence of an *IoPV* issue for a combination of options coincides with the occurrence of an *IoPV* for at least one of the individual options, in practice one would be able to rely on only the analysis of *IoPV* for individual options. In other words, if none of the individual options would manifest *IoPV* issues, the combination of options would not manifest *IoPV* issues either; and if the combination of options manifests *IoPV* issues, at least one individual option would manifest *IoPV* issue. Therefore, we also evaluate the number of t-wise *CTO* that have results contradicting with the single option results.

Table 3: Number of *CTO* collected from the subject systems.

No regression under the default option value, but with regression under other option values.

subject	#CTO	Any	Response time		CPU		Memory		I/O read		I/O write	
		metric	large	med	large	med	large	med	large	med	large	med
Hadoop	4,902	1,155	24	18	517	84	208	214	216	52	526	100
Cassandra	4,197	2,275	600	423	1,094	352	788	404	1,033	363	921	326

Improvement under the default option value but with regression under other option values.

subject	#CTO	Any	Response time		CPU		Memory		I/O read		I/O write	
		metric	large	med	large	med	large	med	large	med	large	med
Hadoop	4,902	668	4	3	425	14	102	36	170	30	426	46
Cassandra	4,197	842	122	53	450	95	220	52	412	93	327	74

Regression in default value and non-regression/improvement in other values.

subject	#CTO	Any	Response time		CPU		Memory		I/O read		I/O write	
		metric	large	med	large	med	large	med	large	med	large	med
Hadoop	4,902	1,022	17	9	431	60	128	200	228	64	441	4
Cassandra	4,197	1,408	236	222	592	298	314	229	553	264	439	229

Any metric means the union #*CTO* of five performance metrics.
Med|large means the effect size *Cliff's delta* of performance regression is medium|large.

Result. The *IoPV* is a common problem, as 61% and 91% of our studied *CTO* in *Hadoop* and *Cassandra* suffer from the *IoPV* problem in at least one performance metric. In addition, each *Hadoop* and *Cassandra* commit has a median percentage of 43% and 96% of the pairs of tests and options that manifest at least one *IoPV* across releases. Although a small percentage of *Hadoop* tests and options suffers from an *IoPV* in each performance metric, e.g. response time, there is a large percentage (61%) of *CTO* suffering from *IoPV* when considering five performance metrics. On the other hand, the percentage of pairs of tests and options that suffer from an *IoPV* is larger than the percentage of pairs of tests and options that do not face an *IoPV* for *Cassandra* across all the performance metrics. The result of high percentage is relative to the number of values of each *CTO*. For instance, *Cassandra* has a larger percentage of *CTO* that suffer from *IoPV*. We find that there are a lot of values of each option in *Cassandra*.

Noted from Table 3, 1,155 out of 4,902 (24%) *CTO* in *Hadoop* and 2,275 out of 4,197 (54%) *CTO* in *Cassandra*, show a performance regression on at least one performance metric when the default configuration does not show any performance regression. For instance, in terms of response time, we observe a performance regression on 42 and 1,023 out of 4,902 and 4,197 *Hadoop* and *Cassandra CTO* respectively, when the default configuration does not show any regression, as shown in Table 3. As shown in the same Table, the performance metric that suffers the most from the *IoPV* problem are the I/O write in *Hadoop* and CPU usage in *Cassandra*. In addition, these are not minor regression differences, as 78% of the regressions are large based on our effect size analysis.

In almost all cases, an *IoPV* result for a t-wise combination of options correlates with an *IoPV* for at least one of the individual options, for all evaluated performance metrics. For instance, 383 and 374 (out of a total of 384) t-wise *CTO* show such a correlation in terms of response time for *Hadoop* and *Cassandra*. Similarly, the *IoPV* of 376, 383, 376, and 382 *Hadoop* t-wise *CTO* correlate with *IoPV* of their constituent individual options in terms of CPU, Memory, I/O read, and I/O write, respectively. We find similar numbers for *Cassandra*. Such results imply that if there is an *IoPV* issue manifested by a combination of options (i.e., a t-wise combination), in practice the same *IoPV* issue will be exhibited by one of the individual options. On the other hand, if none of the individual option manifests an *IoPV* issue, their combination is not likely to manifest any *IoPV* issue either. For this reason, the rest of the paper focuses on the individual *CTO*.

PQ2. How difficult is it to manually identify *IoPV* issues?

Motivation. The goal of this preliminary question is to understand how difficult the manual prediction of *IoPV* (i.e., identification of *IoPV* without running the tests) is. For instance, the higher the number of options that manifest an *IoPV* in a large number of pairs of commits and tests, the more difficult the identification of *IoPV* is, as it indicates that an *IoPV* can occur in an unexpected way and any option can be responsible for such a problem. The lower the number of options that suffer from an *IoPV*, the easier it is to test all of these *IoPV* responsible options.

Approach. To investigate the difficulty of identifying an *IoPV*, we first study the prevalence of *IoPV* in different granularity, i.e., commit, test, and option. Second, we calculate the intersection of the <test, option, *IoPV* > triplets between each pair of commits using the Jaccard similarity defined as follows:

$$J(C1, C2) = \frac{|CTO_{C1} \cap CTO_{C2}|}{|CTO_{C1} \cup CTO_{C2}|} \quad (1)$$

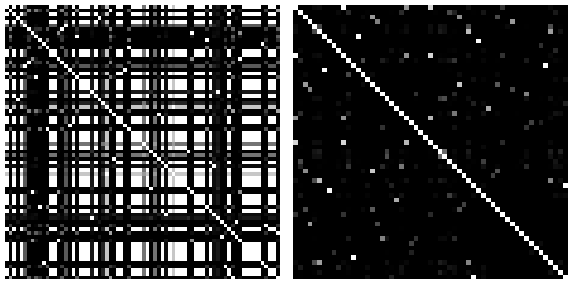
where $C1$ and $C2$ refer to every pair of commits (both consecutive and non-consecutive commits). $|CTO_{C1} \cap CTO_{C2}|$ is the number of *CTO* that share the same <test, option, *IoPV* > (i.e., the intersection). $|CTO_{C1} \cup CTO_{C2}|$ is the total number of unique <test, option, *IoPV* > in commits $C1$ and $C2$ (i.e., the union). The Jaccard distance ranges between 0 and 1, where a value of 1 means that the pair of commits share the same <test, option, *IoPV* > triplets, while 0 indicates that the pair of commits does not share any <test, option, *IoPV* > triplet.

Result. *IoPV* problems are hard to manually predict. The results of the prevalence of *IoPV* in different granularity are shown in Table 4. In particular, 60 out of 74 (81%) commits in *Hadoop* and 56 out of 57 (98%) commits in *Cassandra* show at least one *CTO* with an *IoPV* in at least one performance metric. Similarly, 117 out of 122 (96%) options in *Hadoop* and 50 out of 54 (93%) options in *Cassandra* suffer at least once from an *IoPV* through the studied commits. Table 4 shows more details about how common are *IoPV* for the studied commits, tests, and options. In summary, our results indicate that the *IoPV* problem is not limited to a small set of commits, tests or options, which makes it challenging to predict which *CTO* would have an *IoPV*.

Table 4: Number of unique commits, tests and options with *IoPV* problems.

		Commit		Test		Option	
		Total	IoPV	Total	IoPV	Total	IoPV
Hadoop	Res. time	74	27	74	13	122	74
	CPU	74	47	74	62	122	113
	Memory	74	38	74	59	122	113
	I/O read	74	45	74	53	122	108
	I/O write	74	47	74	57	122	117
	Any metric	74	60	74	67	122	117
Cassandra	Res. time	57	55	216	189	54	43
	CPU	57	55	216	204	54	49
	Memory	57	53	216	202	54	43
	I/O read	57	56	216	202	54	44
	I/O write	57	53	216	192	54	39
	Any metric	57	56	216	208	54	50

Even if most of the commits show at least one *IoPV*, it is not easy to predict which test and option may suffer from the *IoPV*. Figure 3 shows the pairwise Jaccard distance between the <test, option, *IoPV* > triplets of the studied commits in the *Hadoop* and *Cassandra* systems, respectively. The figures indicate that most of the commits do not share any <test, option, *IoPV* > (i.e., with dark cells), especially for the *Cassandra* system (i.e., more dark cells). Therefore, it is difficult for developers to manually identify which tests and options that they need to run and configure to verify the existence of *IoPV*.



(a) Hadoop response time **(b) Cassandra response time**

Figure 3: Pairwise Jaccard distance between the <test, option, *IoPV* > triplets of the studied commits of the *Hadoop* and *Cassandra* system. The x-axis and y-axis show the studied commits. Each cell refers to the Jaccard distance of any pair of commits: the darker the color is, the larger the distance is.

To understand why different commits show inconsistent <test, option, *IoPV* > triplets, we manually analyze some commits that show the largest Jaccard distance from other commits. In particular, there are two and six commits with large Jaccard distance (> 0.8) to all other commits in *Hadoop* and *Cassandra*, respectively. For *Hadoop*, we pick up one [29] out of the two commits of *Hadoop* to manually examine the impacted tests, configuration option and the commit changes. We find that the studied options that cause *IoPV* are related to connection time, such as the options *dfs.ha.fencing.ssh.connect-timeout* and *fs.s3a.connection.timeout*. By examining the code in the test *TestKMS.java*, we find that *TestKMS.java* loads the connection timeout configuration options. And the code changes in this commit trigger the test case in the test *TestKMS.java*. Thus, the commits that impact such connection time-related options and the test may lead to *IoPV* problems while other commits may not lead to the same *IoPV*. For *Cassandra*, we select one commit [30] with the largest Jaccard distance to other commits. Our results show that two tests named *EmbeddedCassandraServiceTest* and *DebuggableScheduledThreadPoolExecutorTest* manifest the largest performance regression regarding options *max_hints_file_size_in_mb* and *memtable_heap_space_in_mb*, respectively. By manually examining the commit changes covered by the tests, we find that there exist code changes in the method *start* within the Java file *EmbeddedCassandraService.java* [31]. Such code changes trigger the test cases to load and initialize options in the impacted tests *EmbeddedCassandraServiceTest* and *DebuggableScheduledThreadPoolExecutorTest*, which lead to performance regression. In particular, 69% of commits in *Hadoop* and 96% of commits in *Cassandra* have a Jaccard distance more than 0.5. Such results imply that different commits may lead to different options and tests that exhibit *IoPV* problems.

Summary of Preliminary Study

IoPV is a common problem in our studied systems and it is difficult to manually identify *IoPV* without exhaustively running the tests. Our results suggest the need for an approach that identifies which *CTO* manifests an *IoPV*.

5 PREDICTING *IOPV* PROBLEMS

RQ1. Can we accurately learn *IoPV* issues in the studied systems?

Motivation. This research question is to evaluate different classification approaches on predicting for which *CTO* one has to check multiple option's values. In our preliminary study, we observe that the *IoPV* is common and hard to manually predict, which indicates that developers need to test different values for each option. However, as there are typically a large number of configuration options (e.g., *Hadoop* version 2.7.3 has 355 configuration options) with different possible values, exhaustively experimenting with all different options for each test in performance testing is time- and resource-consuming. In this RQ, we aim to reduce the effort of conducting configuration-aware performance testing by predicting the need for testing with different values for a given configuration option when a code change is made (i.e., for a *CTO*). Specifically, our approach predicts whether a *CTO* manifests an *IoPV*, such that developers can make an informed decision on whether they should consider different values for that option in their performance testing.

Approach. In this RQ, we follow the detailed steps to build ML models to predict whether a *CTO* manifests an *IoPV*.

Step 1. Data preparation. Our target variable is a binary variable that indicates whether a *CTO* manifests an *IoPV*, which we obtained following the approach discussed in Section 3. We consider four dimensions of software metrics that are related to the likelihood of a configuration option impacting the performance testing of a code commit for each test (i.e., of a *CTO*). Table 5 lists the detailed metrics used in our models. Chen et al. [13] find that code structure, and code change dimensions are important for predicting performance regressions, however they did not consider the impact of different configurations on the manifestation of performance regressions. Therefore, we use the prior dimensions as well as an additional dimension about the configuration options.

Next, we pre-process the features. The code token metrics include thousands of unique code tokens. Thus, we need to pre-process such metrics into a numeric representation. We consider three different approaches to pre-process the code token metrics. **(1) Term frequency-inverse document frequency (tf-idf)**: Tf-idf [38] generates a feature for each unique token. The value of a feature for a commit is the term frequency of the corresponding token (i.e., $tf(t,c) = f_{t,c}$, where $f_{t,c}$ is the number of times a token t appears in commit c) times the inverse frequency of the commits that contain the token ($idf(t) = \log(N/N_t)$, where N is the total number of commits while N_t is the number of commits containing the token t.) **(2) Principal component analysis (PCA)**: Using tf-idf generates a large number of features that may lead to very complex models. Therefore, we apply PCA [39] on the features resulting from tf-idf to reduce the number of features. **(3) Word embeddings**: We use word2vec [40, 41] to code each token into a vector of 128 numerical values. Specifically, we pre-train the embeddings from a large code base [42], then apply the pre-trained embeddings on the tokens in our data.

Step 2. Model construction. We build machine learning models to predict whether a configuration option suffers from an *IoPV* on a given *CTO*. For the generalization of our results, we consider five different types of models, including random forest (RF), logistic regression (LR), XGBoost (XG), neural network (NN), and convolutional neural network (CNN). A random forest is a classifier consisting of a collection of decision tree classifiers [43]. Logistic regression is a statistical model that uses a logit function to model a binary variable as a linear combination of the independent variables [44], which is widely used in software analytics [45, 46]. XGBoost is an efficient and accurate implementation of the gradient boosting algorithm [47, 25]. The neural network model [48] used in our study consists of four layers and is trained with batch size 100, and 10 epochs. The CNN model [49] in our study consists of five layers, and are trained with batch size 100, and 10 epochs.

Step 3. Model evaluation. We use 10-fold cross-validation to evaluate the performance of our models. In each repetition of the 10-fold cross-validation, the whole data set is randomly partitioned into 10 sets of roughly equal size. One subset is used as the testing set (i.e., the held-out set) and the other nine subsets are used as the training set. We train our models using the training set and evaluate the performance of our models on the held-out set. In each fold

Table 5: Overview of our selected metrics.

Dimension	Metric	Rationale
Code change	Number of modified subsystems	The more subsystems are changed, the higher risk the change may be [32].
	Number of modified directories	Changing more directories may more likely introduce performance regressions [32].
	Number of modified files	Changing many source files are more likely to cause performance regressions [33].
	Modified code across files	Scattered changes are more possible to cause performance regressions [34].
	Number of modified methods	Changes altering many methods are more likely to introduce performance regressions [35].
	Number of lines SOC in tests	Program with more lines is more likely to suffer from performance regressions [36].
	Lines of code added	The more lines of code added, the higher risk that the program will suffer from performance regressions [37, 35].
	Lines of code deleted	The more lines of code deleted, the higher risk of performance regression is introduced [37, 35].
Code structure	Number of methods in impacted test	Program with a large number of methods is more likely to suffer from performance regressions.
	McCabe Cyclomatic complexity	Program with higher complexity is more likely to suffer from performance regressions [34].
	Number of called subprograms	Large called subprograms will amplify the regressions if there exist regressions in the called program [33].
	Number of calling subprograms	Large calling subprograms will amplify the regressions if there exist regressions in the called program [33].
Code token	Code tokens of the changed code	Some code tokens may be more related to performance than other tokens.
Option token	Split configuration option names	The name components of a configuration option may be related to a specific performance metric.

of the cross-validation, we use precision, recall and Area Under the receiver operating characteristic Curve (AUC) to measure the performance of our models.

Result. Our models can effectively predict when a *CTO* is manifesting an *IoPV* for all of our five studied performance measures (as shown in Table 6). Our best models (i.e., as indicated by the ***bold-italic*** values) achieve an AUC of 0.85 to 0.94 on the *Hadoop* project and 0.79 to 0.90 on the *Cassandra* project, for different performance metrics. For the *Hadoop* project, RF is the best model for four out of the five performance metrics, achieving an AUC of 0.85 to 0.93. Grebhahn et al. [50] find that random forest performs better compared to other prediction models. Even if XG shows the best AUC performance for the fifth performance metric (i.e., Response time), the difference between RF and XG is only 0.01. For the *Cassandra* project, RF shows the best performance on three out of five performance metrics. NN shows the best performance on also three performance metrics (Memory and I/O read have the same performance as the RF model). The average AUC of the best NN model is 0.83, while the average AUC of the best RF model is 0.82. Note that NN, on the other side, requires a large amount of resources to train and test a model, while the improvements it shows over RF is trivial. CNN shows the best performance on only one performance metric (i.e., with an AUC of 0.79 for the Response time). However, the average AUC of the best CNN model is 0.09 lower than that of RF. In summary, we suggest that developers consider the RF model for predicting when a *CTO* has an *IoPV* problem.

The choice of representation of the code tokens significantly impacts the performance of our models. For the traditional models (RF, LR, and XG), using code embeddings to represent the code tokens often achieves the best performance, while using PCA usually results in the worst performance. For example, for the *Hadoop* project, the RF model achieves an AUC of 0.85 to 0.93 using code embeddings, 0.82 to 0.93 using tf-idf, and only 0.59 to 0.76 using PCA. The reason for the poor performance of the models using PCA might be that PCA significantly reduced the information in the tokens through dimension reduction, even though we considered the principal components that account for 95% of the variance in the original variables. In contrast, for the deep neural network models (NN and CNN), using PCA to represent the code tokens may achieve better results than the other two representations. For example, for the *Cassandra* project, the CNN model combined with PCA achieves the best AUC for two out of the five performance metrics, across all different models. The reason might be that there are a larger number options in our studied systems in the deep neural network models, while using PCA could significantly reduce the number of options to be trained.

Table 6: Results of modeling whether a *CTO* manifests *IoPV*.

Hadoop

	RF with tf-idf			RF with PCA			RF with code embedding			XG with tf-idf
	Pre.	Recall	AUC	Pre.	Recall	AUC	Pre.	Recall	AUC	AUC
Res. time	0.68	0.39	0.93	0.68	0.39	0.66	0.73	0.33	0.93	**0.94**
CPU	0.70	0.51	0.90	0.55	0.02	0.71	0.77	0.60	**0.92**	0.88
Memory	0.64	0.36	0.87	0.48	0.04	0.69	0.75	0.41	**0.91**	0.87
I/O Read	0.68	0.54	0.91	0.58	0.02	0.76	0.79	0.56	**0.93**	0.91
I/O Write	0.63	0.44	0.82	0.44	0.02	0.59	0.72	0.49	**0.85**	0.82
Average	0.67	0.45	0.89	0.55	0.10	0.68	0.75	0.48	**0.91**	0.88

Cassandra

	RF with tf-idf			RF with PCA			RF with code embedding			NN with PCA	CNN with PCA
	Pre.	Recall	AUC	Pre.	Recall	AUC	Pre.	Recall	AUC	AUC	AUC
Res. time	0.74	0.37	0.74	0.45	0.13	0.62	0.67	0.46	0.75	0.73	**0.79**
CPU	0.68	0.39	0.76	0.46	0.15	0.61	0.73	0.59	0.82	**0.90**	0.75
Memory	0.71	0.37	0.78	0.35	0.04	0.61	0.71	0.58	**0.84**	**0.84**	0.77
I/O Read	0.74	0.48	0.79	0.54	0.32	0.67	0.74	0.63	**0.83**	**0.83**	0.68
I/O Write	0.76	0.50	0.82	0.58	0.32	0.68	0.77	0.65	**0.86**	0.84	0.67
Average	0.73	0.42	0.78	0.47	0.19	0.64	0.72	0.58	0.82	**0.83**	0.73

Only full results of the random forest models and the AUC values of models with at least one best result are presented.

Summary of RQ1

Our models can effectively predict whether a *CTO* manifests an *IoPV* problem. Random forest based on code embedding shows the best performance on predicting *IoPV* for most of the performance measures.

RQ2. What are the most important metrics for predicting *IoPV* issues?

Motivation. The goal of this research question is to analyze the models (of RQ1) that predict the *IoPV* to understand the factors that play the most important role in determining whether an option could manifest an *IoPV*. In particular, we focus on the random forest model with code embeddings, as it shows the best performance in predicting *IoPV*. Our results can help practitioners understand and identify the scenarios where they need to adjust their configuration parameters during their performance tests.

Approach. To analyze the most important metrics for predicting *IoPV*, we analyze the impact of each dimension. Different projects have different features as the code tokens and configuration options are different. To make our results more generalizable, we measure the important features at the dimension level instead of feature level. In particular, we consider the following experiments:

Measuring the importance of each dimension of metrics by removing the dimension from the model. In order to study the importance of each dimension of metrics, we build a model with all dimensions and compare it to a model with one dropped dimension at a time. That comparison consists of statistically comparing both models' AUC values. The larger the difference is for a dimension, the more important that dimension is.

Measuring the importance of each dimension of metrics by only keeping the dimension in the model. Since metrics from different dimensions can be correlated, we also consider comparing models that are built using one dimension at a time. For example, some tokens from the code token dimension can be correlated with tokens from the configuration dimension. Therefore, we build a model using one dimension at a time, which results in four models.

Finally, for each model with different dimensions of metrics, we have ten values of AUC since we use 10-fold cross validation. We compare all these models based on their respective AUC values. In particular, we use the Mann-Whitney U test to examine whether there is a statistically significant difference between the original model with all dimensions of metrics and other models with partial metrics.

Result. Every dimension of metrics plays a statistically significant role in predicting *IoPV* cases. Table 7 shows the results of using the Mann-Whitney U test to compare the complete RF model with the RF model that uses only one dimension of metrics or that excludes one dimension of metrics. A p-value that is smaller than 0.05 indicates a statistically significant difference. Table 7 shows that, when only keeping one dimension of metrics, all the resulting models show a statistically significant different (worse) performance. When excluding each dimension of metrics, the resulting models show a statistically significant different (worse) performance in most of the cases (in 16 out of the 20 combinations of the four metric dimensions and the five performance measures for *Hadoop*, and in 14 out of the 20 combinations for *Cassandra*). Our results highlight that one should consider all the four dimensions of metrics together when building a model to predict which *CTO* manifests an *IoPV*.

The code token and configuration dimensions show the best performance among the four dimensions of metrics. For both *Hadoop* and *Cassandra*, for all the performance measures, using only the code token metrics or the configuration metrics in the model achieves a better AUC than using other single dimension of metrics, except that the configuration dimension leads to a relatively worse performance for the I/O write measure of *Cassandra*. The results indicate that the context of the change as well as the goal of configuration options expressed through their tokens are the most important predictors for *IoPV*. However, when we exclude one dimension of metrics from the model , the resulting differences are less significant, and removing the code tokens and the configuration dimensions in fact does not lead to the worst performance. For example, removing the code change dimension from the model for the response time measure of *Hadoop* actually leads to worse performance than removing the code tokens dimension. This is because the different dimensions of metrics are correlated even after correlation analysis; thus the impact of removing one dimension of metrics may be partially mitigated by other dimensions of metrics.

Table 7: The results (p-values) of using the Mann-Whitney U test to statistically compare the AUC of RF with the complete set of metrics vs. with a subset of metrics.

Hadoop				
	Without CC	Without CS	Without CT	Without CON
Res. time	$\ll$0.0001	0.001	0.001	$\ll$0.0001
CPU	0.002	0.052	$\ll$0.0001	$\ll$0.0001
Memory	0.016	0.396	0.019	$\ll$0.0001
I/O Read	0.052	0.154	0.027	0.002
I/O Write	$\ll$0.0001	0.001	0.005	$\ll$0.0001
	Only CC	Only CS	Only CT	Only CON
Res. time	$\ll$0.0001	$\ll$0.0001	$\ll$0.0001	$\ll$0.0001
CPU	$\ll$0.0001	$\ll$0.0001	$\ll$0.0001	$\ll$0.0001
Memory	$\ll$0.0001	$\ll$0.0001	$\ll$0.0001	0.001
I/O Read	$\ll$0.0001	$\ll$0.0001	$\ll$0.0001	0.005
I/O Write	$\ll$0.0001	$\ll$0.0001	$\ll$0.0001	$\ll$0.0001
Cassandra				
	Without CC	Without CS	Without CT	Without CON
Res. time	0.093	0.061	0.052	0.038
CPU	$\ll$0.0001	0.001	$\ll$0.0001	$\ll$0.0001
Memory	$\ll$0.0001	0.009	0.093	0.013
I/O Read	0.001	0.016	$\ll$0.0001	0.006
I/O Write	0.001	0.312	$\ll$0.0001	0.192
	Only CC	Only CS	Only CT	Only CON
Res. time	$\ll$0.0001	$\ll$0.0001	0.019	$\ll$0.0001
CPU	$\ll$0.0001	$\ll$0.0001	0.011	$\ll$0.0001
Memory	$\ll$0.0001	$\ll$0.0001	0.002	0.011
I/O Read	$\ll$0.0001	$\ll$0.0001	$\ll$0.0001	0.005
I/O Write	$\ll$0.0001	$\ll$0.0001	$\ll$0.0001	$\ll$0.0001

CC is Code Change, CS is Code Structure, CT is Code Token, and CON is Configuration.

Summary of RQ2

Every dimension of metrics plays a statistically significant role in predicting whether a *CTO* manifests an *IoPV* problem. The most important dimensions are related to code tokens and configurations.

6 DISCUSSION

In this section, we discuss the learned lessons during the implementation of our approaches, the generalizability of our study and the application of our approach for predicting *IoPV* in interaction of configuration options.

6.1 Generalizability of our study

Adding more case studies can benefit the generalizability, but it still may not address the generalizability issue. Below, we discuss some aspects that may impact the generalizability of our study.

(1) Option quantity: Our approach aims to predict the inconsistent option performance variation issue automatically. We find that there are 365 and 162 configuration options in *Hadoop* and *Cassandra*, respectively. If the number of configuration options in a system is small, e.g., less than ten options, our approach may not have a practical impact for practitioners, as practitioners can examine the small limited number of configuration options manually.

(2) Test and Option coverage: Test and Option coverage. Our approach depends on the readily available small-scale tests in the software systems. If the tests cannot cover the source changes and impacted options, our approach may fail to predict *IoPV*. Since our approach works at the commit level, only the changed methods need to be covered by the test. We find that for all the changed methods in all commits, 68% and 53% are covered by the tests in *Hadoop* and *Cassandra*, respectively. Such changed methods cover a total of 122 and 55 options in *Hadoop* and *Cassandra*, respectively. Such high coverage ensures the success of our approach. This also implies that, in order to adopt our approach, practitioners may first evaluate whether the source code that is likely to be changed is covered by tests.

(3) Test quality: We use the existing available small-scale tests to evaluate the performance variations. Prior research [51, 52] study the use of performance unit tests to increase performance awareness. If the existing test is written with a sub-optimal quality, the performance results may be biased. For example, the test failures in the flaky test may introduce noise and require extra running time to achieve stable performance results. Recent research [52] discusses the reasons for tests not suitable for performance evaluation, which can be leveraged to know how well other projects can adopt our approach.

6.2 The application of our approach for predicting *IoPV* in interaction of configuration options

Based on the findings from the study [2], performance bugs are often related to configurations. The results from study [2] show that the majority (72%) of parameter configuration bugs is related to only one option; about 28% of studied configuration bugs involve two or more configuration options. Therefore, on the one hand, our approach can be directly used to predict the majority of configuration-aware performance issues. On the other hand, our measured data and our approach can be also partially toned to predict a combination of configuration-aware performance issues. We have executed 61,860 *CTO* instances. Even if our measured data cannot represent all the interactions, our measured performance data covers part of interactions of options, like two-way, and N-way options. For example, we assume that there are two options O1 and O2. The possible values of O1 are 0, 4, 8, defaulting to 0, and the possible values of O2 are True and False, defaulting to False. Our performance data covers the following pairwise option values between O1 and O2: 1) $< 0, False >$, 2) $< 4, False >$, 3) $< 8, False >$, 4) $< 0, True >$. On the other hand, our existing performance data misses the following pairwise option values: 1) $< 4, True >$ and 2) $< 8, True >$. N-way testing is a kind of combinational test that requires that every combination of any N options in the software must be tested at least once.

7 THREATS TO VALIDITY

In this section, we discuss the threats to the validity of our study.

External validity. Due to the expensive computing resources needed (we spent around 12,536 machine hours collecting performance data), we conducted our evaluation on two open-source software systems, i.e., *Hadoop* and *Cassandra*. Our findings may not generalize to other software systems. However, we found motivating results on the prevalence of *IoPV* and the performance of our prediction model, which can be replicated by future studies on other software systems.

Internal validity. In our approach, we collect five popular performance metrics, i.e., Response time, CPU, Memory, I/O read and write, while other performance metrics such as throughput can still be explored by future research. We do not consider the combination of configuration options as that will require a huge cost and the goal of our study is to identify and define the *IoPV* issues. On the one hand, prior work [2] mentions that 72% of the performance issues are due to a single option, so our paper covers the most common cause of performance issues. On the other hand, one can use covering arrays to conduct N-way testing for functional tests with very low number of cases [53]. However, for performance testing, the approach likely does not work since we want to isolate each combination's performance impact from others. We encourage future studies to extend our work by considering the interaction of configuration options.

Construct validity. The stability of the cloud-based testing environment may cause testing noise. To minimize the noise, we capture the performance of the corresponding Linux processes of the running tests. Furthermore, for each test, we repeat the execution 30 times independently. Finally, we run all of our experiments in the same environments. There may still exist extreme values as outliers that should not be considered by our approach. To mitigate this threat, we remove the outlier data using the mean ± 3 × standard deviation (STD) as an indicator of outliers.

8 RELATED WORK

In this section, we discuss prior works along three dimensions: performance regression detection, performance model for configurable system, and identifying optimal configuration for performance.

8.1 Performance regression detection

Performance regression detection techniques can be divided into two categories: measurement-based and model-based detection. Measurement-based approaches compare performance metrics (e.g.,

CPU usage) between two consecutive versions to detect performance regressions. For example, Nguyen et al. [54, 55, 56] leveraged control charts to identify performance regressions. Foo et al. [57] proposed an approach that compares a test's performance metrics to historical performance metrics. Model-based approach builds a machine learning model with a set of performance metrics to detect performance regressions. Bodik et al. [58] leveraged a logistic regression model to model system users' behavior to improve Cohen et al.'s model [59]. Foo et al. [60] proposed an approach that uses ensembles of models to detect performance regressions in heterogeneous environments.

Our work complements this line of research in the sense that we consider the configuration aspect of configurable software systems. This paper sheds light on the *IoPV* problem by first quantifying the existence of inconsistent performance variations, then proposing a prediction model that identifies the commits, tests, and options that exhibit the *IoPV* problem.

8.2 Performance model for configurable system

Many prior research has been conducted on predicting performance for configurable software systems. Mühlbauer et al. [61, 62] build performance model to identify performance changes in software performance evolution. Unlike the latter work [62], which analyzes a software system as a whole, our work considers the impact of configurability on individual test cases, implying that our work is more fine-grained. Because our focus is on test cases, it may aid in attributing performance variation not only to configuration options but also to tested functionality. Such prior study provides an evidence that performance changes during software evolution, which motivates our study. Jamshidi et al. [63, 64, 65] employ transfer learning to learn performance model across environments. DeepPerf [66] uses a deep feed-forward neural network to model configurable software systems. The existing studies use the historical revisions' performance, or the sampled configuration's performance to build a model, to estimate the performance of future revisions. Different from prior studies, our prediction model does not require training on the historical performance data, but rather identifies inconsistent option performance variations as software evolves. In general, our study is orthogonal to the above approaches and our measured performance data can be used in future research on performance.

8.3 Identifying optimal configuration for performance

A large body of research has been conducted on performance optimization by finding optimal configurations [5, 6, 67, 68, 7, 69]. Siegmund et al. [5] build mathematical models that describe the impact of a configuration on software performance based on each option's value. Raghavachari et al. [6] propose an iterative approach to identify an optimal configuration in terms of performance. Guo et al. [68] leverage non-linear regression to suggest an optimal configuration. Nair et al. [70, 71] conduct several studies to find well-performing configurations using rank-based and sequential model-based approach. Oh et al. [72] propose a truly random sampling to search configurations recursively to find near-optimal configurations without building a performance model. Kim et al. [73] present a lightweight tool to prune unnecessary configurations for test execution, but they only take into account boolean options. Other efforts identified the optimal configuration options in terms of performance by leveraging existing optimization approaches, i.e., iterative search [74], multi-objective optimization [75], and smart hill climbing [76].

Our goal is neither to identify optimal configurations nor to debug configuration-related performance issues. In particular, we focus on understanding whether a performance improvement or regression is consistent through all the values of an option. That is important, as one can improve the performance of a software system or release new changes that do not impact the performance under one configuration when other configurations hide a performance regression. Furthermore, prior work on this line of research compares the absolute performance between two values for the same option, while this can be subjective, as discussed earlier. One option's value can naturally consume performance as it enables the execution of additional features. However, performance comparison need also considers historical performance data.

9 CONCLUSION

The performance improvement or regression of a software change might not be equally manifested through all the possible configuration options' values, which we refer to as the problem of Inconsistent Option Performance Variation (*IoPV*). In this paper, we observe that *IoPV* is a common problem, which is difficult to manually identify without running exhaustive tests, because most of the commits do not share similar options or tests that may lead to *IoPV* and hide performance regressions. We also observed that predictive models (e.g., RF) can effectively predict the *IoPV* problems using four dimension of metrics that are related to code changes, code structures, code tokens, and configurations. Our findings highlight the importance of considering different configurations when performing performance regression detection, and that leveraging predictive models can mitigate the difficulty of exhaustively considering all configurations of a system during such a process. We expect that our study inspires a wide spectrum of future studies on configuration-aware performance regression detection.

10 DATA AVAILABILITY

The link to our replication package can be found here [77].

REFERENCES

[1] M. Sayagh, N. Kerzazi, B. Adams, and F. Petrillo. 2018. Software configuration engineering in practice: interviews, survey, and systematic literature review. *IEEE Transactions on Software Engineering*. ISSN: 0098-5589. DOI: 10.1109/TSE.2018.2867847.

[2] Xue Han and Tingting Yu. 2016. An empirical study on performance bugs for highly configurable software systems. In *Proceedings of the 10th International Symposium on Empirical Software Engineering and Measurement* (ESEM'16).

[3] Chi Li, Shu Wang, Henry Hoffmann, and Shan Lu. 2020. Statically inferring performance properties of software configurations. In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*. ACM, 10:1–10:16.

[4] [n. d.] Mysql bugs: #47529: query cache performance is bad on multi-core servers. https://bugs.mysql.com/bug.php?id=47529. (Accessed on 01/02/2023). ().

[5] Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kastner. 2015. Performance-influence models for highly configurable systems. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'15)*, 284–294.

[6] Mukund Raghavachari, Darrell Reimer, and Robert D Johnson. 2003. The deployer's problem: configuring application servers for performance and reliability. In *Proceedings of the 25th international conference on Software engineering*. IEEE Computer Society, 484–489.

[7] Yixin Diao, Joseph L Hellerstein, Sujay Parekh, and Joseph P Bigus. 2003. Managing web server performance with autotune agents. *IBM Systems Journal*, 42, 1, 136–149.

[8] Dongpu Jin, Xiao Qu, Myra B. Cohen, and Brian Robinson. [n. d.] Configurations everywhere: implications for testing and debugging in practice. In *Proceedings of the 36th International Conference on Software Engineering* (ICSE'14), 215–224.

[9] Tom White. 2009. *Hadoop: The Definitive Guide*. (1st edition). O'Reilly Media, Inc. ISBN: 0596521979, 9780596521974.

[10] Tse-Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. 2014. Detecting performance anti-patterns for applications developed using object-relational mapping. In *Proceedings of the 36th International Conference on Software Engineering* (ICSE 2014). Hyderabad, India, 1001–1012.

[11] [n. d.] Https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/core-default.xml. https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/core-default.xml. (Accessed on 01/05/2023). ().

[12] [n. d.] Configuring cassandra | apache cassandra documentation. https://cassandra.apache.org/doc/latest/cassandra/configuration/. (Accessed on 01/05/2023). ().

[13] Jinfu Chen, Weiyi Shang, and Emad Shihab. 2022. Perfjit: test-level just-in-time prediction for performance regression introducing commits. *IEEE Trans. Software Eng.*, 48, 5, 1529–1544.

[14] [n. d.] Cassandra/databasedescriptor.java. https://github.com/apache/cassandra/blob/trunk/src/java/org/apache/cassandra/config/DatabaseDescriptor.java. (Accessed on 01/02/2023). ().

[15] Tse-Hsun Chen, Weiyi Shang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. 2016. Cacheoptimizer: helping developers configure caching frameworks for hibernate-based database-centric web applications. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (FSE 2016). ACM, Seattle, WA, USA, 666–677.

[16] Christoph Laaber and Philipp Leitner. 2018. An evaluation of open-source software microbenchmark suites for continuous performance assessment. In *Proceedings of the 15th International Conference on Mining Software Repositories* (MSR '18). ACM, Gothenburg, Sweden, 119–130. ISBN: 978-1-4503-5716-6. DOI: 10.1145/3196398.3196407.

[17] Philipp Leitner and Jürgen Cito. 2016. Patterns in the chaos—a study of performance variation and predictability in public iaas clouds. *ACM Trans. Internet Technol.*, 16, 3, Article 15, (April 2016), 15:1–15:23. ISSN: 1533-5399. DOI: 10.1145/2885497.

[18] Christoph Laaber, Joel Scheuner, and Philipp Leitner. 2019. Software microbenchmarking in the cloud. how bad is it really? *Empirical Software Engineering*, 24, 4, 2469–2508.

[19] [n. d.] Psutil documentation — psutil 5.9.5 documentation. https://psutil.readthedocs.io/en/latest/. (Accessed on 01/05/2023). ().

[20] Jinfu Chen and Weiyi Shang. [n. d.] An exploratory study of performance regression introducing code changes. In *2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017*, 341–352. DOI: 10.1109/ICSME.2017.13.

[21] Kundi Yao, Guilherme B. de Pádua, Weiyi Shang, Steve Sporea, Andrei Toma, and Sarah Sajedi. 2018. Log4perf: suggesting logging locations for web-based systems' performance monitoring. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering, ICPE 2018, Berlin, Germany, April 09-13, 2018*. ACM, 127–138.

[22] Nadim Nachar et al. 2008. The mann-whitney u: a test for assessing whether two independent samples come from the same distribution. *Tutorials in Quantitative Methods for Psychology*, 4, 1, 13–20.

[23] Lee A Becker. 2000. Effect size (es). 12, 2006, 155–159.

[24] Barbara A Kitchenham, Shari Lawrence Pfleeger, Lesley M Pickard, Peter W Jones, David C. Hoaglin, Khaled El Emam, and Jarrett Rosenberg. 2002. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on software engineering*, 28, 8, 721–734.

[25] Lizhi Liao, Jinfu Chen, Heng Li, Yi Zeng, Weiyi Shang, Jianmei Guo, Catalin Sporea, Andrei Toma, and Sarah Sajedi. 2020. Using black-box performance models to detect performance regressions under varying workloads: an empirical study. *Empirical Software Engineering*. Accepted.

[26] Heng Li, Tse-Hsun (Peter) Chen, Weiyi Shang, and Ahmed E. Hassan. 2018. Studying software logging using topic models. *Empir. Softw. Eng.*, 23, 5, 2655–2694.

[27] [n. d.] Ckmeans.1d.dp function | r documentation. https://www.rdocumentation.org/packages/Ckmeans.1d.dp/versions/3.4.0-1/topics/Ckmeans.1d.dp. ().

[28] [n. d.] It outages cause businesses $26.5 billion in lost revenue each year, survey. https://www.eweek.com/networking/it-outages-cause-businesses-26.5-billion-in-lost-revenue-each-year-survey/. (Accessed on 01/10/2023). ().

[29] [n. d.] Hdfs-8163. using monotonicnow for block report scheduling causes test. https://github.com/apache/hadoop/commit/b17d365f. (Accessed on 01/09/2023). ().

[30] [n. d.] Merge branch 'cassandra-2.2' into cassandra-3.0. https://github.com/apache/cassandra/commit/0fe82be8. (Accessed on 01/09/2023). ().

[31] [n. d.] Cassandra/embeddedcassandraservice.java at 0fe82be83cceceb12172d63913388678253413bc. https://github.com/apache/cassandra/blob/0fe82be83cceceb12172d63913388678253413bc/src/java/org/apache/cassandra/service/EmbeddedCassandraService.java#L53. (Accessed on 01/02/2023). ().

[32] Audris Mockus and David M Weiss. 2000. Predicting risk of software changes. *Bell Labs Technical Journal*, 5, 2, 169–180.

[33] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. 2006. Mining metrics to predict component failures. In *Proceedings of the 28th International Conference on Software Engineering* (ICSE '06). ACM, Shanghai, China, 452–461. ISBN: 1-59593-375-1.

[34] Ahmed E. Hassan. 2009. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering* (ICSE '09). IEEE Computer Society, Washington, DC, USA, 78–88. ISBN: 978-1-4244-3453-4.

[35] T. Zimmermann, R. Premraj, and A. Zeller. 2007. Predicting defects for eclipse. In *Predictor Models in Software Engineering, 2007. PROMISE'07: ICSE Workshops 2007. International Workshop on*. (May 2007), 9–9.

[36] A. G. Koru, D. Zhang, K. El Emam, and H. Liu. 2009. An investigation into the functional form of the size-defect relationship for software modules. *IEEE Transactions on Software Engineering*, 35, 2, (March 2009), 293–304. ISSN: 0098-5589. DOI: 10.1109/TSE.2008.90.

[37] Nachiappan Nagappan and Thomas Ball. 2005. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th International Conference on Software Engineering* (ICSE '05). ACM, St. Louis, MO, USA, 284–292. ISBN: 1-58113-963-2.

[38] Juan Ramos et al. 2003. Using tf-idf to determine word relevance in document queries. In *Proceedings of the first instructional conference on machine learning*. Volume 242. Piscataway, NJ, 133–142.

[39] Svante Wold, Kim Esbensen, and Paul Geladi. 1987. Principal component analysis. *Chemometrics and intelligent laboratory systems*, 2, 1-3, 37–52.

[40] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2* (NIPS'13). Curran Associates Inc., Lake Tahoe, Nevada, 3111–3119.

[41] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. In *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*.

[42] [n. d.] Zenodo.org. https://zenodo.org/record/3801975. (Accessed on 01/02/2023). ().

[43] Leo Breiman. 2001. Random forests. *Machine learning*, 45, 1, 5–32.

[44] David W Hosmer Jr, Stanley Lemeshow, and Rodney X Sturdivant. 2013. *Applied logistic regression*. Volume 398. John Wiley & Sons.

[45] Chakkrit Tantithamthavorn, Ahmed E Hassan, and Kenichi Matsumoto. 2018. The impact of class rebalancing techniques on the performance and interpretation of defect prediction models. *IEEE Transactions on Software Engineering*.

[46] Weiyi Shang, Ahmed E Hassan, Mohamed Nasser, and Parminder Flora. 2015. Automated detection of performance regressions using regression models on clustered performance counters. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, 15–26.

[47] Tianqi Chen and Carlos Guestrin. 2016. Xgboost: a scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, 785–794.

[48] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. 2011. Deep sparse rectifier neural networks. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2011, Fort Lauderdale, USA, April 11-13, 2011* (JMLR Proceedings). Volume 15. JMLR.org, 315–323.

[49] Steve Lawrence, C. Lee Giles, Ah Chung Tsoi, and Andrew D. Back. 1997. Face recognition: a convolutional neural-network approach. *IEEE Trans. Neural Networks*, 8, 1, 98–113.

[50] Alexander Grebhahn, Norbert Siegmund, and Sven Apel. 2019. Predicting performance of software configurations: there is no silver bullet. *arXiv preprint arXiv:1911.12643*.

[51] Vojtech Horký, Peter Libic, Lukás Marek, Antonín 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, Austin, TX, USA, January 31 - February 4, 2015*. ACM, 289–300.

[52] Zishuo Ding, Jinfu Chen, and Weiyi Shang. 2020. Towards the use of the readily available tests from the release pipeline as performance tests. are we there yet? In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 1435–1446.

[53] Charles J Colbourn. 2004. Combinatorial aspects of covering arrays. *Le Matematiche*, 59, 1, 2, 125–172.

[54] Thanh H.D. Nguyen, Bram Adams, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. 2012. Automated detection of performance regressions using statistical process control techniques. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering* (ICPE '12). ACM, Boston, Massachusetts, USA, 299–310. ISBN: 978-1-4503-1202-8.

[55] T. H. D. Nguyen, B. Adams, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora. 2011. Automated verification of load tests using control charts. In *2011 18th Asia-Pacific Software Engineering Conference*. (December 2011), 282–289. DOI: 10.1109/APSEC.2011.59.

[56] 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, Hyderabad, India, 232–241. ISBN: 978-1-4503-2863-0. DOI: 10.1145/2597073.2597092.

[57] King Chun Foo, Zhen Ming Jiang, Bram Adams, Ahmed E Hassan, Ying Zou, and Parminder Flora. 2010. Mining performance regression testing repositories for automated performance analysis. In *Quality Software (QSIC), 2010 10th International Conference on*. IEEE, 32–41.

[58] Peter Bodík, Moises Goldszmidt, and Armando Fox. 2008. Hilighter: automatically building robust signatures of performance behavior for small-and large-scale systems. In *SysML*.

[59] Ira Cohen, Steve Zhang, Moises Goldszmidt, Julie Symons, Terence Kelly, and Armando Fox. 2005. Capturing, indexing, clustering, and retrieving system history. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*. ACM, Brighton, United Kingdom, 105–118. ISBN: 1-59593-079-5. DOI: 10.1145/1095810.1095821.

[60] King Chun Foo, Zhen Ming 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 *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Volume 2. IEEE, 159–168.

[61] Stefan Mühlbauer, Sven Apel, and Norbert Siegmund. 2019. Accurate modeling of performance histories for evolving software systems. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 640–652.

[62] Stefan Mühlbauer, Sven Apel, and Norbert Siegmund. 2020. Identifying software performance changes across variants and versions. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 611–622.

[63] Pooyan Jamshidi, Norbert Siegmund, Miguel Velez, Christian Kästner, Akshay Patel, and Yuvraj Agarwal. 2017. Transfer learning for performance modeling of configurable systems: an exploratory analysis. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*. IEEE Computer Society, 497–508.

[64] Pooyan Jamshidi, Miguel Velez, Christian Kästner, Norbert Siegmund, and Prasad Kawthekar. 2017. Transfer learning for improving model predictions in highly configurable software. In *12th IEEE/ACM International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS@ICSE 2017, Buenos Aires, Argentina, May 22-23, 2017*. IEEE Computer Society, 31–41.

[65] Pooyan Jamshidi, Miguel Velez, Christian Kästner, and Norbert Siegmund. 2018. Learning to sample: exploiting similarities across environments to learn performance models for configurable systems. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*. ACM, 71–82.

[66] Huong Ha and Hongyu Zhang. 2019. Deepperf: performance prediction for configurable software with deep sparse neural network. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. IEEE, 1095–1106.

[67] Heng Li, Tse-Hsun (Peter) Chen, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. 2018. Adopting autonomic computing capabilities in existing large-scale systems: an industrial experience report. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice* (ICSE-SEIP '18), 1–10.

[68] Jianmei Guo, Krzysztof Czarnecki, Sven Apely, Norbert Siegmundy, and Andrzej Wasowski. [n. d.] Variability-aware performance prediction: a statistical learning approach. In *Proceedings of the 28th International Conference on Automated Software Engineering*, 301–311.

[69] Jianmei Guo, Dingyu Yang, Norbert Siegmund, Sven Apel, Atrisha Sarkar, Pavel Valov, Krzysztof Czarnecki, Andrzej Wasowski, and Huiqun Yu. 2018. Data-efficient performance learning for configurable systems. *Empir. Softw. Eng.*, 23, 3, 1826–1867.

[70] Vivek Nair, Tim Menzies, Norbert Siegmund, and Sven Apel. 2017. Using bad learners to find good configurations. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*. ACM, 257–267.

[71] Vivek Nair, Zhe Yu, Tim Menzies, Norbert Siegmund, and Sven Apel. 2020. Finding faster configurations using FLASH. *IEEE Trans. Software Eng.*, 46, 7, 794–811.

[72] Jeho Oh, Don S. Batory, Margaret Myers, and Norbert Siegmund. 2017. Finding near-optimal configurations in product lines by random sampling. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*. ACM, 61–71.

[73] Chang Hwan Peter Kim, Darko Marinov, Sarfraz Khurshid, Don S. Batory, Sabrina Souto, Paulo Barros, and Marcelo d'Amorim. 2013. Splat: lightweight dynamic analysis for reducing combinatorics in testing configurable systems. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*. ACM, 257–267.

[74] Philipp Lengauer and Hanspeter Mössenböck. [n. d.] The taming of the shrew: increasing performance by automatic parameter tuning for java garbage collectors. In *Proceedings of the 5th international conference on Performance engineering*, 111–122.

[75] Ravjot Singh, Cor-Paul Bezemer, Weiyi Shang, and Ahmed E Hassan. 2016. Optimizing the performance-related configurations of object-relational mapping frameworks using a multi-objective genetic algorithm. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*. ACM, 309–320.

[76] Bowei Xi, Zhen Liu, Mukund Raghavachari, Cathy H. Xia, and Li Zhang. 2004. A smart hill-climbing algorithm for application server configuration. In *Proceedings of the Thirteenth International World Wide Web Conference*, 287–296.

[77] IOPV. 2023. Github.com. https://github.com/iopv/iopv. (Accessed on 01/02/2023). (2023).

Received 2023-02-02; accepted 2023-07-27