An Exploratory Study of Performance Regression Introducing Code Changes

Jinfu Chen
Department of Computer Science and Software Engineering
Concordia University - Montreal, QC, Canada
Email: fu_chen@encs.concordia.ca

Weiyi Shang
Department of Computer Science and Software Engineering
Concordia University - Montreal, QC, Canada
Email: shang@encs.concordia.ca

***Abstract*—Performance is an important aspect of software quality. In fact, large software systems failures are often due to performance issues rather than functional bugs. One of the most important performance issues is performance regression. Examples of performance regressions are response time degradation and increased resource utilization. Although performance regressions are not all bugs, they often have a direct impact on users' experience of the system. Due to the possible large impact of performance regressions, prior research proposes various automated approaches that detect performance regressions. However, the detection of performance regressions is conducted after the fact, i.e., after the system is built and deployed in the field or dedicated performance testing environments. On the other hand, there exists rich software quality research that examines the impact of code changes on software quality; while a majority of prior findings do not use performance regression as a sign of software quality degradation. In this paper, we perform an exploratory study on the source code changes that introduce performance regressions. We conduct a statistically rigorous performance evaluation on 1,126 commits from ten releases of *Hadoop* and 135 commits from five releases of *RxJava*. In particular, we repetitively run tests and performance micro-benchmarks for each commit while measuring response time, CPU usage, Memory usage and I/O traffic. We identify performance regressions in each test or performance micro-benchmark if there exists statistically significant degradation with medium or large effect sizes, in any performance metric. We find that performance regressions widely exist during the development of both subject systems. By manually examining the issue reports that are associated with the identified performance regression introducing commits, we find that the majority of the performance regressions are introduced while fixing other bugs. In addition, we identify six root-causes of performance regressions. 12.5% of the examined performance regressions can be avoided or their impact may be reduced during development. Our findings highlight the need for performance assurance activities during development. Developers should address avoidable performance regressions and be aware of the impact of unavoidable performance regressions.**

I. Introduction

The rise of large-scale software systems (e.g., Amazon.com and Google Gmail) has posed an impact on people's daily lives from mobile devices users to space station operators. The increasing importance and complexity of such systems make their quality a critical, yet extremely difficult issue to address. Failures in such systems are more often associated with performance issues, rather than with feature bugs [1]. Therefore, performance assurance activities are an essential step in the release cycle of large software systems.

Performance assurance activities aim to identify and eliminate performance regressions in each newly released version. A software system is considered to have performance regressions when the performance of the system (or a certain feature of the system) is worse than before. Examples of performance regressions are response time degradation and increased resource utilization. Such regressions may compromise the user experience, increase the operating cost of the system, and cause field failures. We note that performance regression may not be performance bugs, since the performance may still meet the requirement, even though the performance is worse than the previous version. However, detecting performance regressions is an important task since such regressions may have a direct impact on user experience of the software system, leading to significant financial and reputational repercussions. For example, Mozilla has a strict policy on performance regressions [2], which clearly states that unnoticed and unresolved performance regressions are not allowed.

Due to the importance of performance regression, extensive prior research has proposed automated techniques to detect performance regressions [3]–[7]. However, detecting performance regressions remains a task that is conducted after the fact, i.e., after the system is built and deployed in the field or dedicated performance testing environments. Large amounts of resources are required to detect, locate, understand and fix performance regressions at such a late stage in the development cycle; while the amount of required resources would be significantly reduced if developers were notified whether a code change introduces performance regressions during development.

On the other hand, prior software quality research typically focus on functional bugs rather than performance issues. For example, post-release bugs are often used as code quality measurement and are modeled by statistical modeling techniques in order to understand the relationship between different software engineering activities and code quality [8]. In addition, bug prediction techniques are proposed to prioritize software quality assurance efforts [9]–[11] and assess the risk of code changes [12]. However, performance regressions are rarely targeted in spite of their importance.

In this paper, we perform an exploratory study on the performance regression introducing code changes. We conduct a statistically rigorous performance evaluation on 1,126 commits from ten releases of *Hadoop* and 135 commits from five releases of *RxJava*. In particular, the performance evaluation of each code commit with impacted tests or performance

micro-benchmarks is repeated 30 times independently. In total, the performance evaluation lasts for over 2,000 hours. With the performance evaluation results, we identify performance regression introducing changes with statistical tests. To the best of our knowledge, our work is the first **that extensively evaluates and studies performance at the commit level**.

By examining the identified performance regression introducing changes, we find that performance regression introducing changes are prevalent during software development. The identified performance regressions are often associated with complex syndrome, i.e., multiple performance metrics have performance regression. Interestingly, we find that performance regression introducing changes also improve performance at the same time. Such results show that developers may not be aware of the existence of performance regressions, even when they are trying to improve performance. By studying the context and root-causes of performance regression introducing changes, we find that performance regressions are mostly introduced while fixing other functional bugs. Our manual examination on the performance regression introducing code changes identifies six code level root-causes of performance regressions, where some root-causes (such skippable functions) can be avoided. In particular, we find that 12.5% of the examined performance regressions in the two subject systems can be avoided or their impact on performance may be reduced during development.

Our study results shed light on the characteristic of performance regression introducing changes and suggest the lack of awareness and systematic performance regression testing in practice. Based on our findings, performance-aware change impact analysis and designing inexpensive performance tests may help practitioners better mitigate the prevalent performance regression that is introduced during software development.

The rest of this paper is organized as follows: Section II presents our subject systems and our approach to identifying performance regression introducing code changes. Section III presents the results of our case study. Section IV presents the threats to the validity of our study. Section V presents prior research that related to this paper. Finally, Section VI concludes this paper.

II. Case Study Setup

In this section, firstly we discuss our subject systems and our experimental environment. Then we present our approach to identifying performance regression introducing changes.

A. Subject systems

We choose two open-source projects, *Hadoop* and *RxJava* as the subject systems of our case study. *Hadoop* [13] is a distributed system infrastructure developed by the Apache Foundation. *Hadoop* performs data processing in a reliable, efficient, high fault tolerance, low cost, and scalable manner. We choose *Hadoop* since it is highly concerned with its performance and has been studied in prior research in mining performance data [14]. *RxJava* is a library for composing asynchronous and event-based programs by using observable sequences and it carries the JMH benchmarks test options. *RxJava* is a *Java* VM implementation of reactive extensions. *RxJava* provides a slew of performance micro-benchmarks, making it an appropriate subject for our study. We choose the most recent releases of the two subject systems. The overview of the two subject systems is shown in Table I.

TABLE I: Overview of our subject systems.

Subjects	Version	Total lines of code (K)	# files	# tests
Hadoop	2.6.0	1,496	6,086	1,664
	2.6.1	1,504	6,117	1,679
	2.6.2	1,505	6,117	1,679
	2.6.3	1,506	6,120	1,681
	2.6.4	1,508	6,124	1,683
	2.6.5	1,510	6,127	1,685
	2.7.0	1,552	6,413	1,771
	2.7.1	1,556	6,423	1,775
	2.7.2	1,562	6,434	1,784
	2.7.3	1,568	6,439	1,786
RxJava	2.0.0	164	1,107	76
	2.0.1	242	1,513	76
	2.0.2	243	1,524	76
	2.0.3	244	1,524	76
	2.0.4	244	1,526	76

B. Identifying performance regression introducing changes

In this subsection, we present our approach to identifying performance regression introducing changes. In general, we extract every commit from the version control repositories (Git) of our subject systems and identify impacted test cases of each commit. Afterward, we chronologically evaluate the performance of each commit using either the related test cases (for *Hadoop*) or performance micro-benchmarks (for *RxJava*). Finally, we perform statistical analysis on the performance evaluation results to identify performance regression introducing changes. The overview of our approach is shown in Figure 1.

1) Filtering commits: As the first step of our approach, we start off by filtering commits in order to focus on commits that are more likely to introduce performance regressions. In particular, we use the *git log* command to list all the files that are changed in each commit. We only extract the commits that have source code changes, i.e., changes to *.java* files.

In practice, there may exist multiple commits that are made to accomplish one task, making some of the commits temporary. We would like to avoid considering the performance regressions that are introduced in such temporary commits. Since *Hadoop* uses JIRA as their issue tracking system and *RxJava* uses the internal issue tracking system in Github, we use the issue id that is mentioned in each commit message to identify the task of each commit. If multiple commits are associated with the same issue, we only consider the snapshot of the source code after the last commit.

2) Preparing impacted tests: Identifying impacted tests. In order to evaluate performance of each code commit, we use the tests and performance micro-benchmarks that are readily available in the source code of our subject systems. As mature software projects, each subject system consists of a large amount of test cases. For example, *Hadoop* release 2.7.3 contains 1,786 test cases in total. Exercising all test cases may cause two issues to our performance evaluation: 1) the test cases that are not impacted by the code change would dilute the performance impact from the code changes and introduce

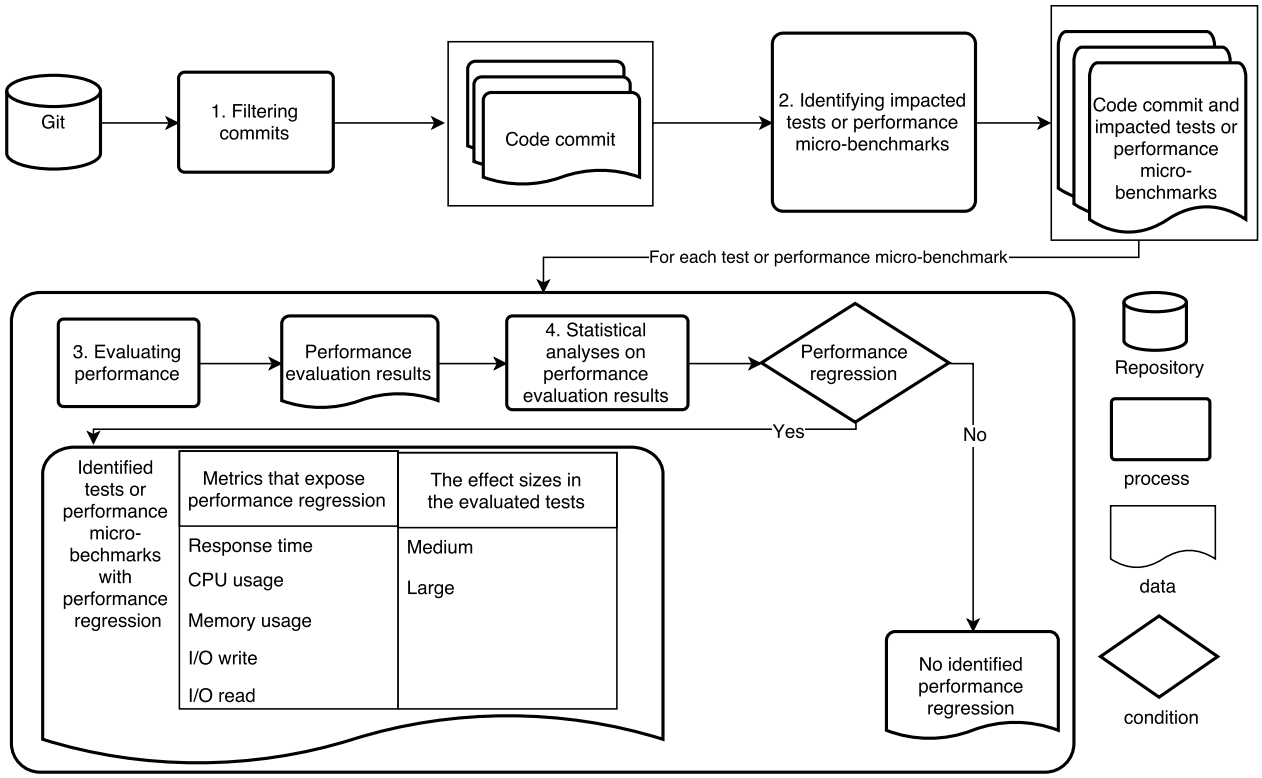


Fig. 1: An overview of our approach that identifies performance regression introducing changes.

noise in the performance evaluation and 2) the large amounts of un-impacted test cases would require extra resources for performance evaluation (e.g., much longer test running time).

Therefore, in this step, we leverage a heuristic to identify impacted tests for each commit. In particular, we find that *Hadoop* test cases follow a naming convention that the name of the test files contains that same name of the source code files being tested. For example, a test file named *TestFSNamesystem.java* tests the functionality of *FSNamesystem.java*. Hence, for each changed source code file in a commit, we automatically identify the test files. If multiple commits are associated with one issue, we consider all the tests that are impacted by these commits, but will later only evaluate performance on the last one of these commits.

Dealing with changed tests. Some commits may change source code and test code at the same time. Such changed test cases will bias the performance evaluation if ample testing logic is added, removed or modified in the test cases. In order to minimize the impact of changed test cases in performance evaluation, we opt to use the test code before the code change. Since the new version of the test cases may include new features of the system, which is not the major concern of performance regression. However, in the cases where old test cases cannot compile or fail, we use the new test cases, since the failure of the compilation or the tests indicates that the old feature may be outdated. Finally, if both new and old test cases are failed or un-compilable, we do not include this test in the performance evaluation. In total, we have 132 tests with 106 commits that use the new tests to evaluate performance and 21 test with 19 commits not included in our performance evaluation. There exist only six commits that are not included because all of their tests are either un-compilable or failed.

Leveraging micro-benchmarks for RxJava. Fortunately, *RxJava* provides a slew of micro-benchmarks with the goal of easing performance evaluation. Micro-benchmark is used to evaluate different small units' performance in *RxJava*.We opt to run all 76 micro-benchmarks from *RxJava*. In the rest of this paper, we also refer these micro-benchmarks as test cases to ease the description of our results.

3) Evaluating performance: In this step, we exercise the prepared test cases and the performance micro-benchmarks to evaluate performance of each considered commit. Our performance evaluation environment is based on Azure node type Standard F8s (8 cores, 16 GB memory). We do not create a cluster of machines to evaluate performance for *Hadoop* since the exercised tests all run on local machine. We do not opt to build *Hadoop*, deploy on a cluster and run *Hadoop* jobs on the cluster to evaluate performance for the following reasons: 1) not all commits can be built into a deployable version of *Hadoop*; 2) running *Hadoop* jobs to evaluate performance may not cover the changed code in each commit; 3) it is challenging to locate the cause of performance regressions from deployed *Hadoop* on clusters. In addition, prior research [15] has leveraged repeated local tests to evaluate performance.

In order to generate statistically rigorous performance results, we adopt the practice of repetitive measurements [16] to evaluate performance. In particular, each test or performance micro-benchmark is executed 30 times independently. We collect both domain level and physical level performance metrics during the tests. We measure the response time of each test case as domain level performance metric. A shorter response time indicates better performance from the users' perspective. Sometimes performance regressions may not cause impact on

response time but rather cause a higher resource utilization. The high resource utilization, may not directly impact user experience, however, it may cause extra cost when deploying, operating and maintaining the system, with lower scalability and reliability. For example, systems that are deployed on cloud providers (like Microsoft Azure) may need to choose virtual machines with higher specification for higher resource utilization. Moreover, a software release with a higher memory usage is more prone to crashes from memory leaks. Therefore, physical level performance metrics are also an important measurement for performance regressions. We use a performance monitoring software named *psutil* [17] to monitor physical level performance metrics, i.e., the CPU usage, Memory usage, I/O read, and I/O write of the software, during the test.

4) Statistical analyses on performance evaluation: Statistical tests have been used in prior research and in practice to detect whether performance metric values from two tests reveal performance regressions [18]. After having the performance evolution results, we perform statistical analyses to determine the existence and the magnitude of performance regression in a statistically rigorous manner. We use Student's t-test to examine if there exists statistically significant difference (i.e., p-value < 0.05) between the means of the performance metrics. A p-value < 0.05 means that the difference is likely not by chance. A t-test assumes that the population distribution is normally distributed. Our performance measures should be approximately normally distributed given the sample size is large enough according to the central limit theorem [19].

T-test would only tell us if the differences of the mean between the performance metrics from two commits are statistically significant. On the other hand, effect sizes quantify such differences. 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 be small even if the difference is trivial). We use *Cohen's d* to quantify the effects [20]. *Cohen's d* measures the effect size statistically and has been used in prior engineering studies [21], [22]. *Cohen's d* is defined as:

$$Cohen's\ d = \frac{mean(x1) - mean(x2)}{s}$$

where *mean(x1)* and *mean(x2)* are the mean of two populations, and s is the pooled standard deviation [23].

$$effect\ size = \begin{cases} trivial & \text{if } Cohen's\ d \leqslant 0.2 \\ small & \text{if } 0.2 < Cohen's\ d \leqslant 0.5 \\ medium & \text{if } 0.5 < Cohen's\ d \leqslant 0.8 \\ large & \text{if } 0.8 < Cohen's\ d \end{cases}$$

III. Case Study Result

In this section, we perform an exploratory study on the extracted performance regressions from our subject systems (*Hadoop* and *RxJava*). Our study aims to answer two research questions. For each research question, we present the motivation of the question, the approach that we use to answer the question, the results of the question and we discuss the results.

RQ1: How prevalent are performance regression introducing changes?

Motivation: Prior research has conducted empirical studies on performance bugs [24]–[27], using the reported performance bugs in issue reports (like JIRA issues). However, there may exist many more performance issues, such as performance regressions, that are not reported as JIRA issues. On the other hand, we evaluate performance on each code commit instead of depending on JIRA issues. Intuitively, we may uncover instances of performance regressions that are not reported, and hence are not be able to be investigated using the approach of prior studies. Therefore, in this research question, we start off by examining how prevalent are detected performance regression introducing changes.

Approach: With the approach presented in Section II, we obtain the results of performance evaluation of the impacted tests in every commit of our subject systems. Since one commit may impact multiple tests, where there may exist both performance regression and performance improvement. However, from users' perspective, having worse performance in one feature may result in bad experiences, despite the performance improvement in other features. Therefore, we examine the existence of performance regression by each impacted test separately, instead of considering a commit as a whole.

We use both domain level performance metrics, i.e., response time, and physical level performance metrics, i.e., CPU usage, Memory usage, I/O read and I/O write, as measurements of performance regressions. As explained in Section II, we examine whether a commit would cause any test case to complete with a statistically significantly longer response time, or utilizing statistically significantly more resources. To ensure the identified performance regressions are not negligible, we only consider a test having performance regression if the effect size is medium or large.

In order to understand whether each performance metric can provide complementary information to others, we also calculate the Pearson correlation between the effect sizes of performance regressions calculated using different metrics. Therefore, we would understand whether we can use a smaller set of metrics to identify performance regressions.

Results: **Performance regressions are not rare instances and are often with large effects.** We find 243 and 91 commits that contain at least one test with performance regression in at least one performance metric for *Hadoop* and *RxJava*, respectively. In particular, we find 93 commits from *Hadoop* and 91 commits from *RxJava* that contain at least one test case with performance regression in response time. In fact, some commits may have multiple test cases that demonstrate statistically significantly slower response time as performance regression. In the total of 1,270 executed tests from *Hadoop* and 7,600 executed tests from *RxJava*, 129 and 1,410 have statistically significantly slower response time with medium or large effect sizes, respectively. The performance regressions on response time may have a direct impact on users' experiences, making these regressions higher priority to be examined by developers. When examining the effect sizes of the detected performance regressions, we find that there exist more performance-regression-prone tests with large effect sizes than medium (see Table II). Such results imply that developers may not ignore these performance regressions since they may have large impact on system performance. In addition, we detect more tests with performance regressions in CPU and Memory usage, than other performance metrics. Since CPU

and Memory usage both have large impact on the capacity of the software systems, these regressions may impact reliability or financial cost of the software system.

Physical performance metrics are important complementary indicators of performance regressions. We use the four physical performance metrics, i.e., CPU usage, Memory usage, I/O read and I/O write to measure performance regression. We find that with the physical performance metrics, we can identify more commits and tests with performance regressions that are not identified with response time. In fact, we find the low correlation between the effect sizes calculated with response time and the physical metrics (see Table III). On the other hand, the effect sizes calculated with the physical metrics may have medium or large correlations with each other. We believe that these correlations are often due to the nature of the software itself (e.g., database accessing software may have its CPU usage highly correlated with I/O).

Discussions: **Performance regressions are often with complex syndromes.** One of the approaches in practice of resolving performance regression is to examine syndrome of the regression, i.e., which physical performance metrics contain performance regression, with the consideration of code changes. However, we find that such an approach can be inefficient due to the complexity of performance regressions. In our case study results, we find that 154 commits and 203 tests from *Hadoop*, 91 commits and 777 tests from *RxJava*, have multiple physical performance metrics associated with performance regressions. Moreover, there even exist one test in one commits and one test in three commits of *Hadoop* and *RxJava* having all four physical metrics with performance regressions. Resolving these performance regressions may be challenging and time-consuming.

Performance regression and performance improvement co-exist. We find that although many commits contain performance regressions, most of these commits also have performance improvements at the same time. Figure 2 shows the number of tests that have performance regression and improvement for each commit measured using response time[1]. In *Hadoop*, 70 commits contain both performance improvement and regression in different tests. Among these commits, 82 tests executed with these commits have performance regression while 117 tests have performance improvement. In *RxJava*, all 91 performance-regression-prone commits have performance improvement. Among the tests executed for these commits, 1,410 tests have performance regression and 1,545 tests have performance improvement. It may be the case that such performance regressions are side-effect of performance improvement activities. However, our results suggest the possible trade-off between improving performance and introducing performance regression at the same time. Therefore, in-depth analysis on these code changes is needed to further understand the context and reason of introducing performance regressions (see RQ2). In addition, our results illustrate the need for performance regression testing in order to increase the awareness of introducing performance regressions during other development activities.

[1]Due to limited space, we do not show such results for other metrics. Those results will be shared with our data online. https://jinfuchen.github.io/icsmeData

Finding: *We find that performance regression introducing changes are prevalent phenomenon with complex syndromes, yet lacking in-depth understanding from the current and prior research. Moreover, these code changes mostly introduce performance regressions while improving performance at the same time.*
Actionable implication: *The findings suggest the need of more frequent performance assurance activities (like performance testing) in practice.*

RQ2: What are the root-causes of performance regressions?

Motivation: In RQ1, we find that there exist prevalent performance regressions that are introduced by code changes. If we can understand what causes the introduction of these performance regressions, we may provide guidance or automated tooling support for developer to prevent the regressions during code change.

Approach: We follow two steps in our approach to discover the reasons for introducing performance regressions. First, we investigate the high-level context when these performance regressions are introduced. We use the issue id in a commit message to identify the issue report (JIRA issue report for *Hadoop* and Github issue report for *RxJava*) that is associated with a performance regression introducing code change. We use the type of issues as the context (fixing a bug or developing new features) that are related to the performance regression introducing changes. Sometimes, in *Hadoop*, an issue may be labeled as "subtask". We manually look for the related issue field in the issue report to for the issue type. However, there still exist "subtask" issues for which we cannot identify an issue type.

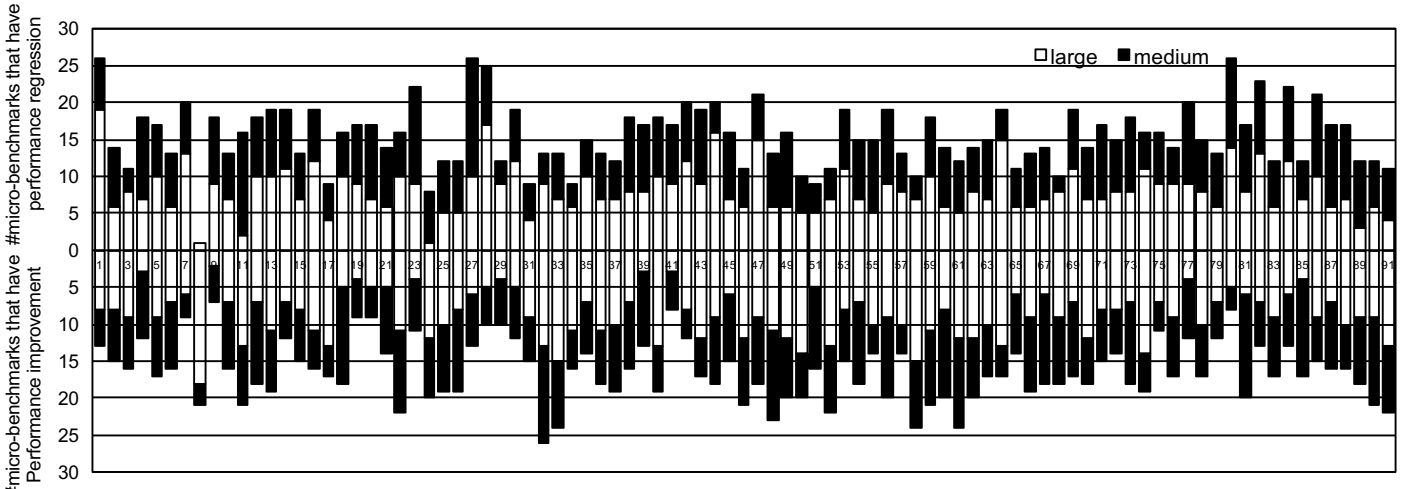
The information in issue reports is about the entire commit rather than the code change that impacts the tests with regression. Therefore, in the second step, we would like to know the code level root-causes (e.g., which code change) of the performance regressions in each identified test from RQ1. Shown in Table II, there exist 338 and 3,100 tests that have at least one metric with performance regression, in *Hadoop* and *RxJava*, respectively. For *Hadoop*, we manually examine all the code changes that are associated with the corresponding tests where performance regression are identified. For *RxJava*, we take a statistically significant random sample (95% confidence level and 5% confidence interval) from the 3,100 tests from *RxJava*. Our random sample consists of 342 tests in total. We follow an iterative approach to identify the root-causes that the code change introduces performance regression, until we could not find any new reasons. Based on our manual study, we also try to examine whether the introduced performance regression can be avoided or whether the perforamnce impact from these regressions may be reduced.

Results: **A majority of the performance regressions are introduced with bug fixing, rarely with new features.** Figure 3 presents the number of performance regression introducing commits that are associated with different issue types. We find that 176 out of 243 (72%) of the commits from *Hadoop* and 48 out of 91 commits (53%) from *RxJava* are associated with issue type *bug*. Such results show that these performance regressions are often introduced during bug fixing tasks. But manually examining all these issues, we find that

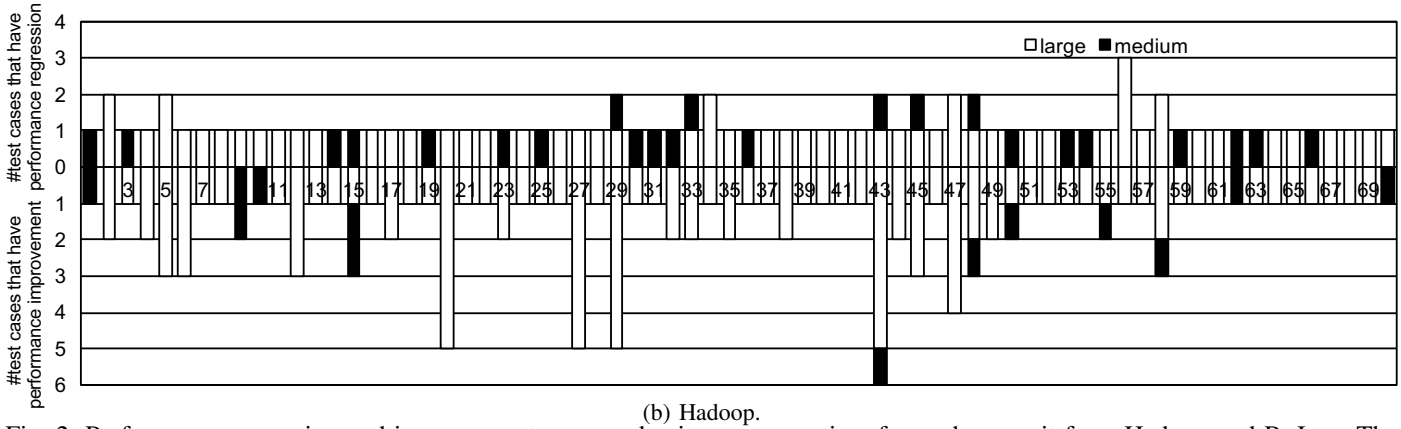
TABLE II: Results of identifying performance regression introducing changes.

Number of commits that have at least one test with performance regressions in different metrics.						
	Any metric	Response time	CPU	Memory	I/O read	I/O write
Hadoop	243	93	175	138	90	82
RxJava	91	91	91	91	91	90

Total number of tests with performance regressions in different metrics.												
	Total executed tests	Any metric	Response time		CPU		Memory		I/O read		I/O write	
			large effect	medium effect	large effect	medium effect	large effect	medium effect	large effect	medium effect	large effect	medium effect
Hadoop	1,270	338	87	42	202	97	167	74	75	28	75	17
RxJava	7,600	3,100	745	665	659	487	919	489	657	449	38	0



(a) RxJava.



(b) Hadoop.

Fig. 2: Performance regression and improvement measured using response time for each commit from Hadoop and RxJava. The commits are ordered chronologically from left to right.

all such issues are fixing functional bugs. For example, issue HADOOP-11252 is associated with type *bug*. The goal of the issue is to control the timeout of RPC client. While performing functional bug fixing, developers may introduce performance issues at the same time.

We only observe three commits that are associated with having new features. We think the reason is that when having new features, developers typically would create a new test, or modify existing test to include the new feature. However, in our approach to identifying performance regressions (see Section II), we ensure to use the exact same test cases with prioritizing on using the old test to eliminate the chance of detecting new feature as performance regression.

Performance regressions may be introduced by tasks with typically low impact. Another interesting finding is that performance regressions can also be introduced during tasks that are not generally considered with high impact. For example, 12 commits are associated with issue type *cleanup* and 6 commits are associated with issue type *documentation*. We manually investigate these commits and find that, all too often developers label the issue as *documentation* or *cleanup* while committing code changes for other small tasks (like bug fixes) on the side. In *RxJava*, issue #4987 is labeled as *documentation* but developers fix additional input sources problems on this issue. Issue #4706 is labeled as *cleanup* but fix minor mistakes for operators. Such small tasks may

TABLE III: Pearson correlation between effect sizes measured using different performance metrics.

Hadoop					
	Response time	CPU	Memory	I/O read	I/O write
Response time	1	-0.02	0.04	0.01	0.06
CPU	-0.02	1	0.60	0.32	-0.02
Memory	0.04	0.60	1	0.27	0.04
I/O read	0.01	0.32	0.27	1	0.06
I/O write	0.06	-0.02	0.04	0.06	1

RxJava					
	Response time	CPU	Memory	I/O read	I/O write
Response time	1	-0.01	0.01	0.01	-0.01
CPU	-0.01	1	0.39	0.46	0.06
Memory	0.01	0.39	1	0.57	-0.26
I/O read	0.01	0.46	0.57	1	0.06
I/O write	-0.01	0.06	-0.26	0.06	1

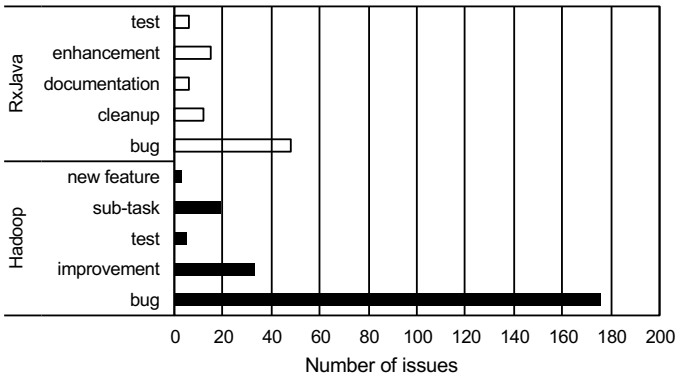


Fig. 3: Number of performance regression introducing commits associated with different issue types.

introduce unexpected performance regressions. Therefore, developers should not ignore such commits that are labeled with low-impact task when evaluating performance regressions.

We identify six code level root-causes of introducing performance regressions. Based our manual analysis on all commits that are identified with performance regressions, we discover six code level root-causes. The distribution of each root-cause is presented in Table IV.

Changing function calls. The regression may be introduced by changed function calls in the source code. For example, in class *Mover.java* of commit #c927f938 in *Hadoop*. Developers added an API call *shuffle* of *Collections* inside a loop, leading to the performance regression. In particular, we identify four patterns of changing function calls that may introduce performance regression: 1) new functionality, 2) external impact, 3) changing algorithm and 4) skippable function. In particular, the performance regressions that are introduced by changing algorithm and skippable function should be avoided or resolved by developers. On the other hand, some code changes that introduce new functionality and depend on external resources may not be avoidable. In total, among the 334 identified regression in this root-cause, we find 36 of them are avoidable. Developers should still be aware of the un-avoidable regression and consider possible alternatives if they have large impact on users.

Changing parameters. The regression may be introduced by changing parameters. For example, in commit #81a445ed in *Hadoop*. The developer added new parameter *conf* into the function *doPreUpgrade* in file *FSImage.java*. The parameter *conf* contains a large number of variables such that initialization can cause memory overhead when calling function *doPreUpgrade*. Making it worse, the function is called inside a loop, leading to large performance regression. We identify three patterns of changing parameters that may introduce performance regressions: 1) using more expensive parameter, 2) changing condition parameter and 3) changing configuration parameter. In particular, using more expensive parameter is the case when a parameter is more expensive than before. If all the data in the new parameter is indeed needed, developer may not be able to resolve this regression, while on the other hand, if developers can identify unneeded data in the parameter, the performance impact from the identified regression may be reduced. Similarly, changing condition parameter and changing configuration parameter may both be due to the need of other code change. Among the 100 identified regression in this root-cause, we find 18 of them are avoidable or reducible.

Changing conditions. Changing condition can change the code that is executed and may cause more operation eventually executed by the software, leading to performance regressions. For example, in class *AccessControlList.java* of commit #3c1b25b5 in *Hadoop*, developers changed the *else* condition to *else if*. So every time the function *isEmpty* inside the condition has to execute. More execution inside the *else if* will execute and it will cause more operations. We identified 13 avoidable or reducible regression out of 85 regressions this root-cause.

Having extra loops. Changing loops may significantly slow down performance. For example, in commit #94a9a5 in *Hadoop*, developers added a *for* loop into the file *LeafQueue.java*, which adds an item to a queue repetitively, while such action can be done as a batch process. If the functionality of a loop can be achieved by batch process, developers may consider implementing batch to minimize the regression. Such solutions can make this regression avoidable or may reduce the performance impact from the loop. All the identified regressions in this root-cause are un-avoidable.

Using more expensive variables. Some variables are more expensive to be held in memory and need more resources to visit or operate. For example, it is recommended to use local variables and to avoid *static* variables. If the keyword *static* is used to declare a variable, the lifetime of the variable will be longer, costing more memory. Developer should avoid performance regressions that are introduced by unnecessary expensive variables in the code. We find 9 out of 48 regressions avoidable or reducible in this root-cause.

Introducing locks and synchronization. Locks are expensive actions for software performance. Introducing locks and synchronization can suspend threads waiting on a lock until released, causing performance degradation on response time. For example, in commit #2946621a in *Hadoop*, developers added *synchronized* operation to lock the block in class *MetricsSourceAdapter.java*, in order to protest the shared resources used by the two functions inside the block. *synchronized*

operation introduces performance regression to the software. Indeed, it is often necessary to have locks in the source code to protect shared resources. On the other hand, developers should always only lock the necessary resource to minimize performance regression. All the identified regressions in this root-cause are un-avoidable.

Discussion: **A considerable amount of performance regressions are avoidable.** Based on our manual study on the root-causes of performance regression, even though only few regressions are due to having new features, many regressions cannot be avoided. For example, if new function calls or more data is needed to fix a bug, such performance regressions may not be avoidable. However, there also exist performance regressions that should be entirely avoided by developers, such as having skippable functions. Such performance regressions may eventually become performance bugs. Nevertheless, developers should still be aware of the impact from un-avoidable performance regressions, and minimize the impact on users' experiences by either providing more hardware recourses or considering alternative solutions. Among the total 680 manually examined performance regressions, we find 85 of them are avoidable or their impact may be reduced. With more frequent and thorough performance assurance activity, the impact from these performance regressions can be minimized.

Functional bugs and performance regressions. We find that performance regressions are mostly introduced during fixing functional bugs. Such findings may be due to two reasons. First of all, performance regression testing is not enforced during development. After developers fix a functional bug, there is no systematic mechanism to prevent the introduction of performance regression at the same time. Although thorough performance regression testing can help avoid such performance regressions, these tests are often resource intensive. Designing inexpensive performance testing can help developers better avoid performance regressions in practice. Second, the performance regressions can be ignored due to the pressure or trade-off of fixing functional bug. Developers may consider the high importance of fixing a functional bug while choose to sacrifice performance for the ease of bug fixes. It is important to make such choices wisely based on the impact of functional bugs and performance regressions. In-depth user studies and field data analysis may help developers minimize the impact from these choices.

Finding: *We find that the majority of performance regressions are introduced with fixing functional bugs, while surprisingly, tasks that are typically considered with low impact also may introduce performance regression. In addition, we identify six root-causes of performance regressions, some of which are avoidable.*
Actionable implication: *Developers should always be aware of the possible performance regression from their code change, in order to address avoidable regressions or minimize the impact from un-avoidable regressions.*

IV. Threat to Validity

A. External Validity

Generalizing our results. In our case study, we only focus on fifteen releases from two open source systems, i.e., *Hadoop* and *RxJava*. Both of the subject systems are mainly written in *Java* languages. Some of the findings might not be generalizable to other systems or other programming languages. Future studies may consider more releases from more systems and even different programming languages (such as C#, C++).

B. Internal Validity

Subjective bias of manual analysis. The manual analysis for root-causes of performance regression is subjective by definition, and it is very difficult, if not impossible, to ensure the correctness of all the inferred root-causes. We classified the root-causes into six categories; however, there may be different categorizations. Combining our manual analysis with controlled user studies on these performance regressions can further address this threat.

Causality between code changes and performance regressions. By manually examining the code changes in each commit, we identify the root-causes of each performance regression. However, the performance regression may be not caused by the particular code change but due to unknown factors. Furthermore, the performance regression may not be introduced by one change to the source code but a combination of confounding factors. In order to address this threat, future work can leverage more sophisticated causality analysis based on code mutation can be leveraged to confirm the root-cause of the performance regression.

Selection of performance metrics. Our approach requires performance metrics to measuring performance. In particular, we pick one commonly used domain level and four commonly used physical level performance metrics based on the nature of the subject systems. There exist a large number of other performance metrics. However, practitioners may require system-specific expertise to select an appropriate set of performance metrics that are important to their specific software. Future work can include more performance metrics based on the characteristic of the subject systems.

C. Construct Validity

Monitoring performance of subject systems. Our study is based on the ability to accurately monitor performance of our subject systems. This is based on the assumption that the performance monitoring library, i.e. *psutil* can successfully and accurately providing performance metrics. This tool monitoring library is widely used in performance engineering research [16], [28]. To further validate our findings, other performance monitoring platforms (such as PerfMon [29]) can be used.

Noise in performance monitoring results. There always exists noise when monitoring performance [30]. For example, the CPU usage of the same software under the same load can be different in two executions. In order to minimize such noise, for each test or performance micro-benchmark, we repeat the execution 30 times independently. Then we use a statistically rigorous approach to measuring performance regressions. Further studies may opt to increase the number of repeated executions to further minimize the threat based on their time and resource budget.

Issue report types. We depend on the types of issues that are associated with each performance regression introducing

TABLE IV: Number of tests with different root-causes of performance regressions with the number of avoidable or reducible ones in brackets.

	Changing function call				Changing parameters			Using more expensive variables	Having extra loops	Changing condition	Introducing locks and synchronization	Others
	New functionality	External impact	Changing algorithm	Skippable function	Expensive parameter	Condition parameter	Configuration parameter					
Hadoop	125 (11)	32 (6)	15 (2)	6 (6)	12 (4)	15 (2)	22 (5)	22 (5)	6 (0)	37 (8)	6 (0)	40 (5)
Rxjava	119 (6)	18 (3)	19 (2)	0	22 (3)	21 (3)	8 (1)	26 (4)	8 (0)	48 (5)	0	53 (4)

commit. The issue report type may not be entirely accurate. For example, developers include extra code changes in issue reports with type *documentation*. Firehouse-style user studies [31] can be adopted to better understand the context of performance regression introducing changes.

The effectiveness of the tests. In our case study, we leverage test cases and performance micro-benchmarks to evaluate performance of each commit. In particular, for *Hadoop* our heuristic of identifying impacted tests are based on naming conventions between source code files and test files. In addition, we also rely on the readily available performance micro-benchmarks in *RxJava*. Our heuristic and the performance micro-benchmarks both may not cover all the performance impacts from code changes. However, the goal of our paper is not to detect all performance regression in the history of our subject systems, but rather collect a sample of performance regression introducing commits for our further investigation. Future work may consider using more sophisticated analysis to identify the impacted tests [32] or manually adapting the tests to address this threat. Moreover, conducting systematic long-lasting performance tests may minimize this threat, the long-lasting time of these test (often more than eight hours) make it almost impossible for every commit. It is still an open research challenge of how to design in-expensive yet representative performance tests, which our case study signifies the importance of breakthrough in such research area.

In-house performance evaluation. We evaluate the performance of our subject systems with our in-house performance evaluation environment. Although we minimize the noise in the environment to avoid bias, such an environment is not exactly the same as in-field environment of the users. There is a threat that the performance regressions identified in our case study may not be noticeable in the field. To minimize the threat, we only consider the performance regressions that have non-trivial (turn out to be mostly large in our experiment) effect sizes. In addition, with the advancing of DevOps, more operational data will become available for future mining software repository research. Research based on field data from the real users can address this threat.

V. Related Work

In this section, we present the related work to this paper.

A. Performance regression detection

A great amount of research has been proposed to detect performance regression. Ad hoc analysis selects a limited number of target performance counters (e.g., CPU and memory) and performs simple analysis to compare the target counters. Heger *et al*. [5] present an approach to support software engineers with root cause analysis of the problems. Their approach combines the concepts of regression testing, bisection and calls tree analysis to detect performance regression root cause analysis as early as possible.

Pair-wise analysis compares and analyzes the performance metrics between two consecutive versions of a system to detect the problem. Nguyen *et al*. [6], [33], [7] conduct a series of studies on performance regressions. Nguyen *et al*. propose an approach to detect performance regression by using a statistical process control technique called control charts. They construct the control chart and apply it to detect performance regressions and examine the violation ratio of the same performance counter. Malik *et al*. [34] propose approaches that combine one supervised and three unsupervised algorithms to help performance regression detection. They employ feature selection methods named Principal Component Analysis (PCA) to reduce the dimensionality of the observed performance counter set and validate their approach through a large case study on a real-world industrial software system [35].

Model-based analysis builds a limited number of detected models for a set of target performance counters (e.g., CPU and memory) and leverages the models to detect performance regressions. Xiong *et al*. [36] propose a model-driven framework to diagnose the application performance in cloud condition without manual operation. In the framework, it contains three modules consisting of sensor module, model building module, and model updating module. It can automatically detect the workload changes in cloud environment and lead to root cause of performance problem. Cohen *et al*. [37] propose an approach that builds a promising class of probabilistic models (Tree-Augmented Bayesian Networks or TANs) to correlate system level counters and systems average-case response time. Cohen *et al*. [38] present that performance counters can successfully be used to construct statistical models for system faults and compact signatures of distinct operational problems. Bodik *et al*. [39] employ logistic regression with L1 regularization models to construct signatures to improve Cohen *et al*.'s work.

Multi-models based analysis builds multiple models from performance counters and uses the models to detect performance regressions. Foo *et al*. [40] propose an approach to detect potential performance regression using association rules. They utilize data mining to extract performance signatures by capturing metrics and employ association rules techniques to collect correlations that are frequently observed in the historical data. Then use the change to the association rules to detect performance anomalies. Jiang *et al*. [41] present two diagnosis algorithms to locate faulty components: RatioScore and SigScore based on component dependencies. They identify the strength of relationships between metric pairs by utilizing an information-theoretic measures and track system state based on in-cluster entropy. A significant change in the in-cluster entropy is considered as a sign of a performance fault. Shang *et al*. [4] propose an approach that first clusters performance metric based on correlation. Each cluster of metrics is used to

TABLE V: Comparing this work with the four prior studies [24]–[27]

	Jin *et al*. [25]	Huang *et al*. [24]	Zaman *et al*. [26], [27]	This work
Data source	Issue reports	Issue reports	Issue reports	Code commits
Granularity	Patch	Performance issue	Performance issue	Performance micro-benchmarks or impacted tests
# instances	109	100	100	3,438 (680 for manual study)
Main approach	Dynamic rule-based checker	Static analysis and risk modeling	N/A	Repeated measurement

build a statistical model to detect performance regressions.

The vast amounts of research on performance regression detection signify its importance and motivate our work. Prior research on performance regressions are all designed to be conducted after the system is built and deployed. In this paper, we explore performance regressions at commit level, i.e., when they are introduced.

B. Empirical studies on performance

Empirical studies are conducted in order to study performance issues [24]–[27]. Jin *et al*. [25] study 109 real world performance issues that are reported from five open source software. Based on the studied 109 performance bugs, Jin *et al*. [25] develop an automated tool to detect performance issues. Zaman *et al*. [26], [27] conducted both qualitative and quantitative studies on performance issues. They find that developers and users face problems in reproducing performance bugs. More time is spent on discussing performance bugs than other kinds of bugs. Huang *et al*. [24] studied real world performance issues and based on the findings. They propose an approach called performance risk analysis (PRA), to improve the efficiency of performance regression testing.

Table V summarizes the comparison between this work and the four prior studies [24]–[27]. In particular, prior studies only study a small amount (around 100) of performance issues often due to the lower number of such issues reported. On the contrary, since our study does not depend on the existence of performance issue reports, we observe a much higher prevalence of performance regressions that are introduced during development. Therefore, our results suggest that maintaining the performance of software may be more challenging. Without an efficient feedback channel from the end users, the developers may overlook possible performance regressions. Our findings suggest the importance of the awareness of possible performance regressions introduced during development.

Luo *et al*. [3] propose a recommendation system, called PerfImpact, to automatically identify code changes that may potentially be responsible for performance regression between two releases. Their approach searches for input values that expose performance regressions and compare execution traces between two releases of a software to identify problematic code changes. Hindle et al. [42] present a general methodology to measure the impact of different software metrics on power consumption. They find the effect of software change on power consumption regressions. Hasan *et al*. [43] create energy profiles as a performance measurement for different Java collection classes. They find that the energy consumption can have large difference depending on the operation. Lim *et al*. [44] use performance metrics as a symptom of performance issues and leverage historical data to build the Hidden Markov Random Field clustering model. Such a model has been used to detect both reoccurring and unknown performance issues.

Prior studies on performance typically are based on either limited performance issue reports or release of the software. However, the limit amount of issue reports and releases of the software hides the prevalence of performance regressions. In our paper, we evaluate performance at commit level. Therefore, we are able to identify more performance regressions and are able to observe the prevalence of performance regression introducing changes in development.

VI. Conclusion

Automatically detected performance regressions are often difficult to fix at the late stage. In this paper, we conduct an empirical study on performance regression introducing changes in two open source software *Hadoop* and *RxJava*. We evaluate performance of every commit by executing impacted tests or performance micro-benchmarks. By comparing performance metrics that are measured during the tests or performance micro-benchmarks, we identify and study performance regressions introduced by each commit. In particular, this paper makes the following contributions:

- To the best of our knowledge, our work is the one of the first to evaluate and to study performance regressions at the commit level.
- We propose a statistically rigorous approach to identifying performance regression introducing code changes. Further research can adopt our methodology in studying performance regressions.
- We find that performance regressions widely exist, and often are introduced after bug fixing.
- We find six root-causes of performance regressions that are introduced by code changes. 12.5% of the manually examined regressions can be avoided or their performance impact may be reduced.

Our findings call for the need of frequent performance assurance activities (like performance testing) during software development, especially after fixing bugs. Although such activities are often conducted before release [45], while developers may find it challenging since many performance issues may be introduced during the release cycle. In addition, developers should resolve performance regressions that are avoidable. For the performance regressions that cannot be avoided, developers should evaluate and be aware of their impact on users. If there exist a large impact on users, strategies, such as allocating more computing resources, may be considered. Finally, in-depth user studies and automated change impact on performance are future directions of this work.

VII. Acknowledgment

We thank Microsoft Azure for providing infrastructure support for this research.

References

[1] E. Weyuker and F. Vokolos, "Experience with performance testing of software systems: issues, an approach, and case study," *Transactions on Software Engineering*, vol. 26, no. 12, pp. 1147–1156, Dec 2000.

[2] "Mozilla performance regressions policy." [Online]. Available: https://www.mozilla.org/en-US/about/governance/policies/regressions/

[3] Q. Luo, D. Poshyvanyk, and M. Grechanik, "Mining performance regression inducing code changes in evolving software," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16. New York, NY, USA: ACM, 2016, pp. 25–36.

[4] W. Shang, A. E. Hassan, M. Nasser, and P. Flora, "Automated detection of performance regressions using regression models on clustered performance counters," in *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '15. New York, NY, USA: ACM, 2015, pp. 15–26.

[5] C. Heger, J. Happe, and R. Farahbod, "Automated root cause isolation of performance regressions during software development," in *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '13. New York, NY, USA: ACM, 2013, pp. 27–38.

[6] T. H. Nguyen, B. Adams, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora, "Automated detection of performance regressions using statistical process control techniques," in *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '12. New York, NY, USA: ACM, 2012, pp. 299–310.

[7] T. H. D. Nguyen, M. Nagappan, A. E. Hassan, M. Nasser, and P. Flora, "An industrial case study of automatically identifying performance regression-causes," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: ACM, 2014, pp. 232–241.

[8] A. E. Hassan, "Predicting faults using the complexity of code changes," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 78–88.

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

[10] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Proceedings of the 27th International Conference on Software Engineering*, ser. ICSE '05. New York, NY, USA: ACM, 2005, pp. 284–292.

[11] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06. New York, NY, USA: ACM, 2006, pp. 452–461.

[12] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, June 2013.

[13] T. White, *Hadoop: The Definitive Guide*, 1st ed. O'Reilly Media, Inc., 2009.

[14] M. D. Syer, W. Shang, Z. M. Jiang, and A. E. Hassan, "Continuous validation of performance test workloads," *Automated Software Engineering*, vol. 24, no. 1, pp. 189–231, 2017.

[15] R. Singh, C.-P. Bezemer, W. Shang, and A. E. Hassan, "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*, ser. ICPE '16. New York, NY, USA: ACM, 2016, pp. 309–320.

[16] T.-H. Chen, W. Shang, A. E. Hassan, M. Nasser, and P. Flora, "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*, ser. FSE 2016. New York, NY, USA: ACM, 2016, pp. 666–677.

[17] "psutil," 2017. [Online]. Available: https://github.com/giampaolo/psutil

[18] H. M. Alghmadi, M. D. Syer, W. Shang, and A. E. Hassan, "An automated approach for recommending when to stop performance tests," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Oct 2016, pp. 279–289.

[19] T.-H. Chen, W. Shang, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora, "Detecting performance anti-patterns for applications developed using object-relational mapping," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 1001–1012.

[20] L. A. Becker, "Effect size (es)," *Accessed on October*, vol. 12, no. 2006, pp. 155–159, 2000.

[21] V. B. Kampenes, T. Dybå, J. E. Hannay, and D. I. Sjøberg, "A systematic review of effect size in software engineering experiments," *Information and Software Technology*, vol. 49, no. 11, pp. 1073–1086, 2007.

[22] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. El Emam, and J. Rosenberg, "Preliminary guidelines for empirical research in software engineering," *IEEE Transactions on software engineering*, vol. 28, no. 8, pp. 721–734, 2002.

[23] J. Hartung, G. Knapp, and B. K. Sinha, *Statistical meta-analysis with applications*. John Wiley & Sons, 2011, vol. 738.

[24] P. Huang, X. Ma, D. Shen, and Y. Zhou, "Performance regression testing target prioritization via performance risk analysis," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 60–71.

[25] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu, "Understanding and detecting real-world performance bugs," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12. New York, NY, USA: ACM, 2012, pp. 77–88.

[26] S. Zaman, B. Adams, and A. E. Hassan, "Security versus performance bugs: a case study on firefox," in *Proceedings of the 8th working conference on mining software repositories*. ACM, 2011, pp. 93–102.

[27] S. Zaman, B. Adams, and Hassan, "A qualitative study on performance bugs," in *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, ser. MSR '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 199–208.

[28] T. M. Ahmed, C.-P. Bezemer, T.-H. Chen, A. E. Hassan, and W. Shang, "Studying the effectiveness of application performance management (apm) tools for detecting performance regressions for web applications: An experience report," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16. New York, NY, USA: ACM, 2016, pp. 1–12.

[29] R. Enbody, "Perfmon: Performance monitoring tool," *Michigan State University*, 1999.

[30] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney, "Producing wrong data without doing anything obviously wrong!" *SIGPLAN Not.*, vol. 44, no. 3, pp. 265–276, Mar. 2009.

[31] E. Murphy-Hill, T. Zimmermann, C. Bird, and N. Nagappan, "The design of bug fixes," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 332–341.

[32] A. Qusef, G. Bavota, R. Oliveto, A. De Lucia, and D. Binkley, "Recovering test-to-code traceability using slicing and textual analysis," *J. Syst. Softw.*, vol. 88, pp. 147–168, Feb. 2014.

[33] T. H. D. Nguyen, B. Adams, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora, "Automated verification of load tests using control charts," in *2011 18th Asia-Pacific Software Engineering Conference*, Dec 2011, pp. 282–289.

[34] H. Malik, H. Hemmati, and A. E. Hassan, "Automatic detection of performance deviations in the load testing of large scale systems," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 1012–1021.

[35] H. Malik, Z. M. Jiang, B. Adams, A. E. Hassan, P. Flora, and G. Hamann, "Automatic comparison of load tests to support the performance analysis of large enterprise systems," in *Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on*. IEEE, 2010, pp. 222–231.

[36] P. Xiong, C. Pu, X. Zhu, and R. Griffith, "vperfguard: An automated model-driven framework for application performance diagnosis in consolidated cloud environments," in *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '13. New York, NY, USA: ACM, 2013, pp. 271–282.

[37] I. Cohen, J. S. Chase, M. Goldszmidt, T. Kelly, and J. Symons, "Correlating instrumentation data to system states: A building block

for automated diagnosis and control." in *OSDI*, vol. 4, 2004, pp. 16–16.

[38] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox, "Capturing, indexing, clustering, and retrieving system history," in *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, ser. SOSP '05. New York, NY, USA: ACM, 2005, pp. 105–118.

[39] P. Bodík, M. Goldszmidt, and A. Fox, "Hilighter: Automatically building robust signatures of performance behavior for small-and large-scale systems." in *SysML*, 2008.

[40] K. C. Foo, Z. M. Jiang, B. Adams, A. E. Hassan, Y. Zou, and P. Flora, "Mining performance regression testing repositories for automated performance analysis," in *Quality Software (QSIC), 2010 10th International Conference on*. IEEE, 2010, pp. 32–41.

[41] M. Jiang, M. A. Munawar, T. Reidemeister, and P. A. Ward, "Automatic fault detection and diagnosis in complex software systems by information-theoretic monitoring," in *Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on*. IEEE, 2009, pp. 285–294.

[42] A. Hindle, "Green mining: a methodology of relating software change and configuration to power consumption," *Empirical Software Engineering*, vol. 20, no. 2, pp. 374–409, Apr 2015.

[43] S. Hasan, Z. King, M. Hafiz, M. Sayagh, B. Adams, and A. Hindle, "Energy profiles of java collections classes," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 225–236.

[44] M. H. Lim, J. G. Lou, H. Zhang, Q. Fu, A. B. J. Teoh, Q. Lin, R. Ding, and D. Zhang, "Identifying recurrent and unknown performance issues," in *2014 IEEE International Conference on Data Mining*, Dec 2014, pp. 320–329.

[45] Z. M. Jiang and A. E. Hassan, "A survey on load testing of large-scale software systems," *IEEE Transactions on Software Engineering*, vol. 41, no. 11, pp. 1091–1118, Nov 2015.