Egyptian Informatics Journal 21 (2020) 219–229

Contents lists available at ScienceDirect

Egyptian Informatics Journal

journal homepage: www.sciencedirect.com

Detecting software performance problems using source code analysis techniques

Salma Eid [a,*], Soha Makady [b], Manal Ismail [a]

[a] Faculty of Computer and Information Technology, National Egyptian E-Learning University, Egypt
[b] Computer Science Department, Faculty of Computers and Artificial Intelligence Cairo University, Egypt

ARTICLE INFO

Article history:
Received 20 September 2019
Revised 2 February 2020
Accepted 9 February 2020
Available online 20 February 2020

Keywords:
Performance regression
Performance aware unit test
Version control system
Source code analysis

ABSTRACT

Software is evolving rapidly. Many software systems release new versions in short iterations. Code changes within such versions may be enhancements, bug fixes, or new features. While preserving some of those changes, the functionality of software may accidentally degrade its performance within a new version when compared to a previous version thus introducing performance regressions. Developers suffer from finding code changes that cause performance regressions especially with a large number of code changes. The cost of detecting performance regressions increases massively as the size of the changes increases. In this paper, we propose a novel approach for automatically identifying potential code changes that cause performance regression from one system version to a subsequent one using source code analysis techniques. Such approach is realized through a prototype tool called PerfDetect. PerfDetect retrieves the changed source code across new and previous version of a specific application's source code. PerfDetect automatically: (a) identifies relevant unit test cases for the changed source code within the new version, (b) compares the execution time of these relevant test cases across the new and previous system versions using various loads to detect performance regressions, and (c) analyzes the root causes for such performance regression within the corresponding source code. In case no relevant unit tests are found as per step (a), automatically generated unit tests for the changed code are used instead within step (a). The proposed approach is evaluated on four open-source applications to assess its ability to detect performance regressions and identify their root causes. The evaluation results demonstrate that the proposed approach can automatically detect the root cause of performance regression in a shorter time as compared to alternative performance detection approaches. Furthermore, PerfDetect detects performance regressions, that were missed by other performance regression techniques, due to its reliance on source code analysis techniques.

© 2020 Production and hosting by Elsevier B.V. on behalf of Faculty of Computers and Artificial Intelligence, Cairo University. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

1. Introduction

Performance regression is a situation where the software performs slowly in a new version when compared to a previous version. With a large number of code changes between two system versions, the cost of detecting performance regressions, and identifying their relevant causing code changes increases massively. Such increased cost is attributed to two main reasons. First, the effort needed to find inputs that would properly examine the analyzed system to detect performance regressions increases with the increase in that system's overall code size. Second, the identification of the code changes causing (or contributing to) any identified performance regressions, is challenging due to the possibly large set of code changes across two system versions, and the absence of previous performance records for newly added code within the new system version. Some approaches for identifying performance regressions across two system versions demand the presence of performance aware unit tests [1]; a requirement that is challenging to find within applications. Developers are ordinally reluctant to writing unit tests in general [2]. Other performance

* Corresponding author.
E-mail addresses: seid@eelu.edu.eg (S. Eid), s.makady@fci-cu.edu.eg (S. Makady), mismaeel@eelu.edu.eg (M. Ismail).

Peer review under responsibility of Faculty of Computers and Information, Cairo University.

Production and hosting by Elsevier

https://doi.org/10.1016/j.eij.2020.02.002
1110-8665/© 2020 Production and hosting by Elsevier B.V. on behalf of Faculty of Computers and Artificial Intelligence, Cairo University.
This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

regression detection techniques: (i) rely on applying random inputs to a complete system, in order to detect performance regressions [3], (ii) demand the presence of two running version of the analyzed applications [1,3]. Such approaches do not consider utilizing the source code of the applications of interest to identify performance regressions.

We propose a new approach for detecting performance regression within two versions of software using source code analysis. The proposed approach uses code differences across the two versions to identify relevant unit tests that directly/indirectly exercise the modified code parts. Such tests identification process is done using both static and dynamic source code analysis techniques. The identified relevant tests are executed on both the new system version and on the previous system version. The execution time of those test cases is compared across the previous and new system versions to identify performance regressions. In the absence of manually written relevant unit tests, automatically generated unit tests are used instead. The proposed approach is realized through a prototype tool named PerfDetect. The proposed approach supersedes alternative performance regression detection techniques in various aspects as follows: (i) does not demand a completely running application, (ii) does not exploring non-changed parts of the application if interest, hence decreasing the performance regression detection time, (iii) relies on unit tests, which are highly guaranteed to access specific parts of the application's source code, as compared to randomly generated test inputs which can easily miss specific parts of an application's source code [3], and (iv) utilizes source code analysis techniques contrary to alternative approaches.

This paper makes the following contributions: (1) A novel approach to detect performance regressions across two system versions, while utilizing unit tests using source code analysis techniques. (2) An empirical evaluation of the proposed approach is conducted on real performance regressions, across two different versions, for four open source applications, namely: AgileFant, Apache Commons Math, Apache Commons IO, and Xalan. The results show that our proposed approach could successfully identify performance regressions and improvements across the four systems. Even more, our proposed approach relies on finer granularity unit tests, rather than randomly generated test input. Accordingly, PerfDetect could detect additional performance regressions that were missed by alternative performance regression detection approaches.

This paper is structured as follows. We start with explaining our problem statement in Section II. Section III introduces our proposed approach, whereas Section IV shows our experimental setup. Section V shows the results our study, whereas section VI discusses the remaining issue Section VII concludes the paper, while pointing out future research directions.

2. Problem statement

Current approaches that aim to identify performance regression issues, and analyze their root causes within the source code suffer from limitations in several directions as follows. (1) Such approaches demand running a complete test suite to compare and detect performance problems across two releases, which is an extremely time-consuming process for large test suites. (2) Existing approaches rely on the presence of execution traces for the analyzed system. Such execution traces can hardly exist, especially if the system is undergoing a major change that hinders having a completely running version. Furthermore, the quality of the analysis relies on the strength and code coverage of those traces. (3) Such approaches treat the analyzed system as a black-box executable program, while ignoring the rich wealth of information present within the modified source code across the different releases, (4) Existing approaches ignore newly added features that may be one of the causes of performance regression that effect indirectly on another piece of code.

Currently, developers must run a complete set of test cases for all the system which is an extremely time-consuming process with huge test suites. Developers must have a completely running version. Test suites used must be accurate to reveal performance regressions.

As an example of a performance regression, consider that we have two system versions vi (current version) and vi-1 (previous version). The current version has some modified code portions that differ from the previous one. As shown in Fig. 1, the method called retrieveByName() is modified within current version Vi and became called method named getByName() in class settingDAO instead of get() method in class settingCashe. When software engineers execute current version, they notice that the execution time of the current version v_i is increased comparing to the previous version v_{i-1}. With all code changes that have been submitted in the current version, developers may not be aware that this specific change has been responsible for the increased execution time of the overall application. Developers need to review the source code of the current changes to check if there is a performance regression and detect which code portion is responsible for this performance regression.

Depending only on the black-box technique for finding performance regression may fail in detecting that such method causes performance regression. Such failure is attributed to the fact that such specific code change might not get executed at all with randomized inputs. Randomized inputs may not cover all branches of code in this version.

Alternatively, if some developers decide to exhaustively run all the source code of an application to detect performance regressions, such task would be hectic. That task would demand covering all branches of the given application's source code; an excessively time-consuming activity. Besides being time consuming, executing all the application's source code would result in executing code portions that did not change at all. Such execution of unchanged code portions is a redundant activity that would delay detecting the performance regressions.

3. Approach

3.1. Overview

Our approach identifies performance regressions, and their root causes across two system versions. Such approach should alleviate (1) running a complete test suite across two system versions to identify performance problem which is an extremely time-consuming process for large test suites; (2) relying on the presence of execution traces for the analyzed system, which is hard to exist especially if the system is undergoing a major change that hinders having a completely running version; (3) relying on the quality of the used inputs to create the execution traces. If the execution traces result from inputs that exercise complete system scenarios, such inputs may ignore specific modified source code elements whose modification resulted in performance degradation. Accordingly, our approach utilizes different information to avoid executing the whole system to detect performance problems. Such information is (a) the version control information of the two system versions, (b) the added/modified code across the two system versions, (c) the dependencies between the modified source code and its corresponding test suites.

The proposed approach assumes the presence of (a) an existing system version with at least one previous version (to be used as

V_{i-1}

```
public Setting retrieveByName(String name) {
    return this.settingCache.get(name);
}
```

V_i

```
public Setting retrieveByName(String name) {
    return this.settingDAO.getByName(name);
}
```

Fig. 1. An example of changes between two versions V_{i-1} and V_i.

our reference for acceptable performance), and a current changed version to be analyzed for performance issues, (b) the complete source code of both versions, (c) JUnit tests for both versions, and (d) version control history for both versions like the projects' history present within Git repository [4].

As shown within Fig. 2, the approach follows these steps: (1) select the changed code portions between two versions, (2) identify the relevant test cases to the changed code within the current version (if any), (3) If no relevant JUnit test cases, the approach generates JUnit test cases and compares their performance against a reference version, (4) analyze the performance problems as per the results of the previous step, and use such information to identify the source code changes responsible for such performance degradation.

3.2. Determination of added/changed code portions

To apply the proposed approach, a prototype tool named PerfDetect has been built. The proposed approach starts its first step by comparing two versions of source code to determine which parts of code are changed or added. The methods affected by such changes are then identified to serve as inputs used for analyzing and detecting performance regressions.

Algorithm 1: DETECT-CHANGED-CODE
Input: Two software versions (v_{i-1}, v_i)
Output: List of Changed Methods in v_i (M)

1: $F \leftarrow$ Compare two versions source code
2: **for all** $f_j \in F$ **do**
3: Compare the source code in v_{i-1} and v_i
4: Mark all statements removed or added inside methods
5: Define method m of each changed statement
6: $M \leftarrow$ Add the method m
7: **end for**
8: return M

That first step is represented in the algorithm Alg. 1 DETECT-CHANGED-CODE which takes the source code of the current version (v_i) and the previous one (v_{i-1}) as inputs. PerfDetect compares the source code of both two versions through some version control system (e.g. Git) that holds each version and its change history. PerfDetect outputs all the changed files within the current version v_i. PerfDetect compares the source code of each changed file (f_j) across the two versions to identify the added/changed code using GitHub API [5]. Fig. 3 show an example of one of the changed code portions inside (f_j), the statement in v_{i-1} (highlighted in red) was changed in v_i (highlighted in green).

PerfDetect analyzes each changed statement like the previous example to get the direct method which was affected by the change (m). Consequently, PerfDetect adds the method (m) to list (M) as shown in Fig. 4. Fig. 4 shows an accumulative list of the identified changed methods, across two system versions, like retrieveByName() method. The direct methods will be the input method to be used for detecting performance regressions in the following steps. The proposed approach depends on running test cases that are relevant to only changed methods instead of running

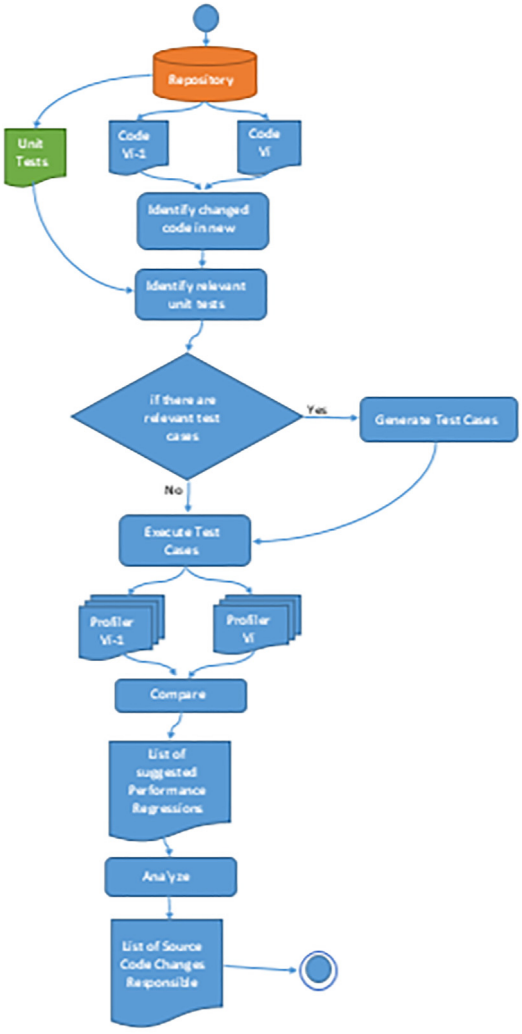

Fig. 2. The workflow of our approach.

```
@Transactional(readOnly = true)
public Setting retrieveByName(String name) {
V-1 -       return this.settingCache.get(name);
V  +        return this.settingDAO.getByName(name);
}
```

Fig. 3. Statement changed in current version and previous one.

the complete test suite to minimize time-consuming process for large test suites. Accordingly, the identified list (M) will be used to locate such relevant test cases within the following step (See Section C).

3.3. Identification of relevant test cases

PerfDetect identifies the relevant test cases to the changed methods in an automated way using Eclipse Call Hierarchy plugin

1) business/impl/MenuBusinessImpl/constructBacklogMenuData()
2) business/impl/PortfolioBusinessImpl/getPortfolioData()
3) business/impl/SearchBusinessImpl/searchByReference()
4) business/impl/SettingBusinessImpl/retrieveByName()
5) business/impl/SettingBusinessImpl/storeSetting()
6) model/WidgetCollection/getWidgets()
7) web/TeamAction/retrieveAll()
8) model/Project/setRank()
9) business/impl/DailyWorkBusinessImpl/setStoryRanks()
10) business/impl/SearchBusinessImpl/taskListSearchResult()
11) business/impl/SearchBusinessImpl/checkAccess()
.....

Fig. 4. Examples of changed methods affected in v_i.

[6]. Eclipse Call Hierarchy plugin is used to find the references of a method whether such reference represents the callers of such method, or the callees of such method. This step is represented in Alg. 2 IDENTIFY-RELEVANT-TEST-CASES which takes as input the list of changed methods (M), all the test cases in the two system versions (T). In each changed method (m_j), PerfDetect identifies all caller methods of (m_j) and detects if each caller method (tdirect) is test case or not by one of these ways: (1) The caller method has the JUnit 4 @Test annotation, (2) The parent class of the caller method is TestCase class.

Algorithm 2: IDENTIFY-RELEVANT-TEST-CASES
Input: List of changed methods ***(M)***, All test cases ***(T)***
Output: List of relevant test cases ***($T_{relevant}$)***

1: ***For all mj ϵ M do***
2: Get all caller methods that call this ***mj***
3: ***While*** relatd test case not found ***do***
4: ***if*** caller method is test case (Direct Test case) (***t_{direct}***)
5: Execute test case (***t_{direct}***)
6: ***if t_{direct}*** call the target method
7: ***$T_{relevant} \leftarrow t_{direct}$***
8: Break
9: ***End if***
10: ***End if***
11: ***End while***
12: ***If $T_{relevant}$*** of ***mj*** is empty
13: ***$t_{indirect}$*** ← identirfy indirect relevant test case(***mj***)
14: ***$T_{relevant} \leftarrow t_{indirect}$***
15: ***end if***
16: ***End for***
17: return ***$T_{relevant}$***

After detecting the relevant JUnit test cases of each modified method for the two system versions, the identified relevant test cases fall in one of two categories. (1) Direct test cases (t_{direct}) that directly call the modified methods, and (2) in-direct test cases ($t_{indirect}$) that call other source code methods, which call the modified methods. PerfDetect starts by checking if the changed method (m_j) has relevant direct test cases. If no relevant direct test cases were found, PerfDetect searches for relevant in-direct test case. PerfDetect outputs all the relevant test cases, whether direct or in-direct test case, in the list (***$T_{relevant}$***). Calculating the complexity of the algorithm depends on the complexity of the selected relevant test cases (see line 5 in Algorithm 2). The complexity of such relevant test cases varies based on the content of the selected test case. As we cannot determine such test cases' complexity till we run the algorithm and execute those tests. Hence, we assume that the complexity of each executed test case to be CT. Accordingly, the complexity of algorithm would be $O(n^2) + O(n^2*CT)$ as $O(n^2)$ is the complexity of nested two loops and $O(n^2*CT)$ is the complexity of test cases multiplied by times of nested two loop [40].

Sometimes modified methods do not have relevant test cases for many reasons: (a) no caller method calls them whether directly or indirectly, (b) the methods that call them do not have test cases, or (c) the caller method is located within a third-party or standard libraries.

Eclipse Call Hierarchy plugin depends on static analysis for finding the caller methods [15]. This plugin involves examining source code and making inferences about the future from it based on data gathered about the code to help you make decisions. One of the limitations of static analysis is thatautomated tools, like Eclipse Call Hierarchy, produce false positive caller methods (i.e., candidate caller methods that do not actually get executed at runtime). Such limitation causes PerfDetect to detect false positive relevant test cases to modified methods at sometimes. For instance, if a modified method *m* exists in a Java Interface *I*, that method will be modified in all subclasses of that interface (e.g., *C1*, *C2*, and *C3*). For that method, PerfDetect detects all the relevant test cases of all that method's occurrences within the *C1*, *C2*, and *C3*, even if method *m* in *C2* is the only one that gets executed at runtime (dynamically). To remove such false positives, PerfDetect needs to further ensure if the identified relevant test case *actually/dynamically* calls the modified method in the correct subclass or another subclass by using dynamic analysis [7–10]. Dynamic analysis is based on running the test cases to avoid reporting wrong relevant test cases not related to the changed method (false positive) and validate the static code analysis findings. PerfDetect uses both static and dynamic analysis to make sure that it selects the correct relevant tests case of each modified method separately.

3.4. Analyzing changed code caused performance regression

Unit tests focus only on functional testing with usually the minimum set of test cases to test the functionality of the components. In order to detect performance regressions properly and achieve the best results, our proposed method requires performance-aware unit tests which are unit tests developed for performance requirements not only for functionality to test cases that can potentially be critical for the performance of the code in the target application context [1]. So we modify the relevant test cases by tools like ContiPerf [11] or JUnitPerf [12] to evaluate the performance. Both tools used by software engineers who need to evaluate the performance of JUnit tests. That was done by building these tools to each JUnit test to increase the workloads (number of users) that execute the unit test cases.

For measuring the execution time of each modified method, we use a tool like JProfiler [13]. JProfiler is a full-featured Java profiling tool (profiler) dedicated to analyzing J2SE and J2EE applications combining CPU, Memory and Thread profiling in one application.

Algorithm 3: EXECUTE-RELEVANT-TEST-CASE
Input: List of related test cases ***($T_{relevant}$)***
Output: List of changes cause performance regression (***Result***)

1: ***For all tj ϵ T do***
2: ***$t_j^i \leftarrow$*** run ***t_j*** in ***v_i***
3: ***$t_j^{i+1} \leftarrow$*** run ***t_j*** in ***v_i***
4: ***$td_j \leftarrow t_j^{i+1} - t_j^i$*** where ***$td_j$ ϵ*** List of Difference execution time (***TD***)
5: ***end for***
6: ***Result*** ← order ***TD***
7: return ***Result***

In the final stage Alg. 3 EXECUTE-RELEVANT-TEST-CASE takes the list of relevant test cases as input and outputs the list of changes causing the performance regressions. Through the relevant unit test case of each modified method, PerfDetect profiles the execution time of the unit test case across the two versions (t_{ij}, t_{i+1j}) with different workloads. The differences in execution time (td_j) between the two versions is then calculated. Finally, PerfDetect orders the differences in execution time for all changed methods according to the execution time differences; higher difference are the ones cause performance regression [1].

3.5. Generating test cases for the changed code portions

For modified methods that have no relevant test cases, PerfDetect generates new test cases for changed code which have no relevant test cases whether directly or indirectly. PerfDetect utilizes a genetic algorithm, through Evosuite [14], for generating test cases to reveal any performance regressions across the two system versions. Evosuite applies a novel hybrid approach that generates and optimizes test cases with different coverage criteria, like individual statements, branches, outputs and mutation testing. The target of Evosuite is generating test cases with assertions to help developers to detect deviations from expected behavior. That target varies from PerfDetect which targets detecting performance variation behaviour. PerfDetect depends on Evosuite for generating JUnit test cases for classes written in Java code. Once those tests are generated by EvoSuite, PerfDetect applies the above mentioned step (See section D) Such step is applied to make those generated test cases to be performance-aware tests, in order to enable them to detect performance regressions.

4. Evaluation

4.1. Research questions

RQ1: Can PerfDetect identify the changed code portions which are responsible for performance regressions?
RQ2: How good is PerfDetect in recommending changed code portions causing performance regressions in comparison to other approaches?

To answer **RQ1,** PerfDetect is applied to four open source systems with different performance regression problems, to evaluate its ability to detect performance regressions (See Sections V.A, V. B, V.C, and V.D). Three steps are followed within that evaluation. First, all the changed source code across the two system versions is identified. Second, the relevant test case to each changed code through the two releases is identified. Third, each test case is executed with different workloads (5, 10, 15, 20 and 25 users) to analyze the identified changes' impacts on performance regressions [3]. Each workload runs five times, then we calculate the average of five times for each workload to get stable results of each workload in each test case.

Within the evaluation, the four open source systems were utilized along with their manually written unit tests, to detect performance regressions using our approach (See Sections V.A, V.B, V.C, and V.D). Additionally, one of those systems (Commons Math) was used another time, along with automatically generated unit tests, to detect performance regressions using our approach (See Sections V.E). For that system, Apache Commons Math, the evaluation showed that we could detect the same performance regressions using both manually written unit tests, and automatically generated unit tests.

To answer **RQ2**, the detected performance regressions by PerfDetect are compared to the performance regressions detected by the other approaches, to assess the quality of PerfDetect results (See Section V.F). That comparison is done for two of the four initially selected open source systems.

4.2. Subject AUTs

We evaluated PerfDetect on four real applications, Agilefant (V3.2[1], V3.3[2]), Apache Commons Mathematics Library (Commons Math V2.1[3], V2.2[4]), Apache Commons IO (V2.0[5],V2.1[6]), and Xalan (V2.7.1[7], V2.7.2[8]).

Agilefant is an open-source web application for helping software engineer to manage agile software development faster. Agilefant is deployed in server core i5 using Tomcat 7.0.47 as a web server environment, and MySQL as the backend database. Commons Math is a lightweight library of mathematics and statistics components addressing common problems not available in the Java programming language.

Commons IO is a library of utilities to assist developers with input and output functionalities. Xalan is an XSLT processor which is provided with different languages including Java. All applications are written in Java and have their test cases. Agilefants' test cases are written using JUnit and EasyMock Frameworks [16]. Test cases of Commons Maths and Commons IO are written using JUnit. Xalan has performance test suite in some versions.

4.3. Setup environment

Versions of Agilefant [17], Commons Math [18], Commons IO [19], and Xalan [20] applications are located in the version control system (Git Repository) [4]. The approach is applicable to any Java-based software systems and pure JUnit 4 tests without any additional framework like EasyMock. EasyMock is used to provide mock objects for interfaces by generating them on the fly so that a dummy functionality can be added to a mock object that can be used in unit testing. PerfDetect needs to test a real scenario for each component without any dummy data. Hence, PerfDetect tests the performance of unit test cases and their impact set rather than those tests functionality only. This is done to get the real execution time of each method in the impact set, and hence identify the cases that can potentially be critical for the performance of the code in the target application context. So EasyMock is inconsistent with our approach.

5. Empirical results

This section analyzes the results of four real Java applications that validate the approach presented in this paper.

5.1. Detecting performance regression in Agilefant

This section analyzes the empirical results of the modified code across two system versions (v3.2 and v3.3) after testing. The results are analyzed for forty-nine changed methods including ten added methods, three deleted methods, and thirty-six modified methods whether from the old version or the new version. That is done by (1) identify the source code differences that have changed between the two system versions, (2) find the related test cases of code differences using the caller hierarchy of eclipse, (3) and

[1] https://github.com/Agilefant/agilefant/tree/v3.2.
[2] https://github.com/Agilefant/agilefant/tree/v3.3.
[3] https://github.com/apache/commons-math/tree/MATH_2_1.
[4] https://github.com/apache/commons-math/tree/MATH_2_2.
[5] https://github.com/apache/commons-io/blob/commons-io-2.0.
[6] https://github.com/apache/commons-io/blob/commons-io-2.1.
[7] https://github.com/apache/xalan-j/tree/xalan-j_2_7_1.
[8] https://github.com/apache/xalan-j/tree/xalan-j_2_7_2

Table 1
Changed methods which have direct relevant test cases in Agilefant.

	Method	Added Lines	Deleted Lines	Test case
1	business/impl/MenuBusinessImpl/constructBacklogMenuData()	1	0	2
2	business/impl/PortfolioBusinessImpl/getPortfolioData()	2	22	2
3	business/impl/SearchBusinessImpl/searchByReference()	1	1	8
4	business/impl/SettingBusinessImpl/retrieveByName()	1	1	1
5	business/impl/SettingBusinessImpl/storeSetting()	0	1	1
6	model/WidgetCollection/getWidgets()	1	1	1
7	web/TeamAction/retrieveAll()	10	2	1
8	model/Project/setRank()	1	1	3
9	model/Task/setRank()	1	1	16
10	model/WhatsNextEntry/setRank()	1	1	2

Table 2
Changed methods which have in-direct relevant test cases in Agilefant.

	Method	Added Lines	Deleted Lines	Test case
11	business/impl/DailyWorkBusinessImpl/setStoryRanks()	27	2	5
12	business/impl/SearchBusinessImpl/taskListSearchResult()	2	1	4
13	business/impl/SearchBusinessImpl/checkAccess()	1	45	9
14	business/impl/TransferObjectBusinessImpl/getBacklogDataRecurseNames()	1	1	8
15	web/UserAction/getTeamMembers()	1	6	4
16	transfer/PortfolioTO/getRankedProjects()	1	16	2
17	transfer/PortfolioTO/getUnrankedProjects()	1	16	2

measure the execution time for each related test cases using JProfiler and ContiPerf.

PerfDetect identifies the added/changed code across two versions using Github API. The initial results are forty-nine changed methods including ten added methods, three deleted methods, and thirty-six modified methods whether from the old version or the new version.

The results of detecting relevant test cases of each modified method for the two system versions indicate that most JUnit test cases in Agilefant are using EasyMock framework which is not helpful for detecting performance regressions. We convert the test cases to unit test cases without using Mocking [21,22] to remove any dummy data and get real results. Some of the modified methods do not have test cases directly but have caller methods that have a related test case that we use it as related test cases indirectly. The results of identifying related test cases are 17 methods that have their test cases including 10 test cases directly as shown in Table 1 and 7 test cases indirectly as shown in Table 2, and 18 methods have not test cases that's because no caller method calls them or the methods that call them do not have test cases indirectly or the caller method is in a library. Table 1 and Table 2 represent the list of methods that have changed in version 3.3 with indicating number of lines added and removed in each changed method and number of related test cases for each one.

After converting any test case to JUnit test case without EasyMock, we add ContiPerf tool for each unit test case to increase the workload by increasing the number of users that execute the unit test cases. We use JProfiler to measure the execution time of each modified method through its unit test case with different workload across the two system versions.

To detect the changed methods caused performance regressions, we obtain the differences of average total execution times for the relevant test case of each change with different workloads (number of users) in two releases. In general, one change with longer total execution times in vi is more likely to trigger performance degradation and that causes longer difference between the execution time in the two releases.

Fig. 5 shows the differences between the average execution time of each changed method for the two versions in different workloads. The x-axis represents the different workload (number of users) and the y-axis represents the variance in execution time in milliseconds of two releases. As the results show, only those with red text are the suspects for performance regression (1,4,5,13). Those changed methods' differences in avg execution time of two versions are increased in noticeable variance more than others changed methods. Other changed methods with green text in Fig. 5 are close to zero and that indicates that the execution time of vi-1 is close to vi-1, so these methods are considered to be less classified as causing performance regressions like methods (1,4,5,13). All changed methods mentioned in this figure have positive values in their variance. The variance of methods (2,3,6,8,15) in Table 1 and Table 2 have negative values, that indicates that the execution time vi is less than the execution time of vi-1 so these methods have performance improvements from vi to 1 to vi.

5.2. Detecting performance regression in apache commons math

Apache Commons Math is the second application we used to evaluate our approach. In this section, we analyze the results of detecting performance regression problems through two versions (v2.1 and v2.2). In 2010, Commons Math developers submitted a new code version (v2.2) with hidden code portions that caused performance regressions. This problem was discovered and reported[9] in 2011. The problem took a long time around one year additional (2012) until developers can detect the code portions caused performance regression and solve it. Our proposed approach in this paper targets helping developers for detecting performance regression faster and earlier.

Our setup here is different; we increase the size of data set used in each test unit test case in addition to increasing the workloads. Assuming that performance-aware unit tests are available and the data set size is large, we applied our approach to the commons math code repository every new code portions committed to the repository. Our approach compares between the last version 2.2 and the previous one 2.1 and detect 25 methods including 9 added methods and 16 methods have changed as shown in Table 3 and Table 4.

The two tables represent list of methods that have changed in version 2.2 with indicating number of added and removed lines

[9] https://issues.apache.org/jira/browse/MATH-578.

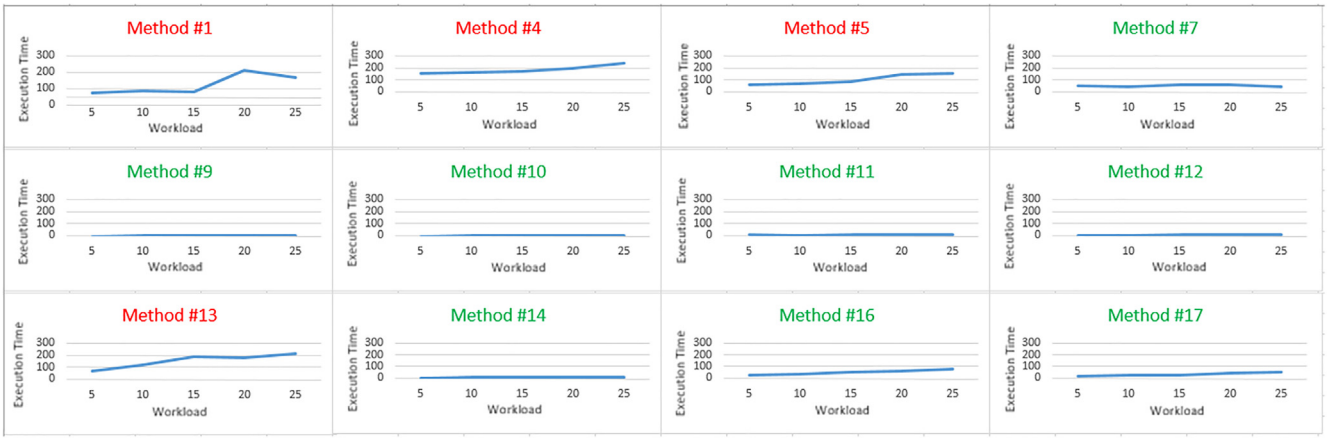

Fig. 5. Difference of execution time for change methods in Agilefant which has positive values as represented in y-axis through different workload (number of users) as represented in x-axis. The execution time reported for the two version v3.2 and v3.3.

Table 3
Changed methods which have relevant related test cases in Commons Math.

	Method	Added Lines	Deleted Lines	Test case
1	src/main/org/apache/commons/math/stat/descriptive/rank/Percentile/evaluate(double[], int, int, double)	16	8	3

Table 4
Changed methods which have in-direct relevant test cases in Commons Math.

	Method	Added Lines	Deleted Lines	Test case
1	src/main/org/apache/commons/math/stat/descriptive/moment/FirstMoment/copy	1	0	2
2	src/main/org/apache/commons/math/stat/descriptive/moment/GeometricMean/copy	1	0	3
3	src/main/org/apache/commons/math/stat/descriptive/moment/Kurtosis/copy	1	0	2
4	src/main/org/apache/commons/math/stat/descriptive/moment/Mean/copy	1	0	3
5	src/main/org/apache/commons/math/stat/descriptive/moment/SemiVariance/copy	1	0	2
6	src/main/org/apache/commons/math/stat/descriptive/moment/Skewness/copy	1	0	2
7	src/main/org/apache/commons/math/stat/descriptive/moment/StandardDeviation/copy	1	0	2
8	src/main/org/apache/commons/math/stat/descriptive/moment/Variance/copy	5	0	3
9	src/main/org/apache/commons/math/stat/descriptive/rank/Max/copy	1	0	3
10	src/main/org/apache/commons/math/stat/descriptive/rank/Min/copy	1	0	3
11	src/main/org/apache/commons/math/stat/descriptive/rank/Percentile/copy	4	0	2
12	src/main/org/apache/commons/math/stat/descriptive/summary/Product/copy	1	0	2
13	src/main/org/apache/commons/math/stat/descriptive/summary/Sum/copy	1	0	3
14	src/main/org/apache/commons/math/stat/descriptive/summary/SumOfLogs/copy	1	0	3
15	src/main/org/apache/commons/math/stat/descriptive/summary/SumOfSquares/copy	1	0	3
16	src/main/org/apache/commons/math/stat/descriptive/moment/FirstMoment/copy	1	0	2

in each changed method, and number of related test cases for each one. Table 3 shows that there is only one changed method which has 3 related test case call it directly (direct relevant test case). Table 4 shows 15 methods changed and each one of them does not have related test case call it directly but there is a test case call it through another method (in-direct relevant test case).

Our approach observes that the following method called **Percentile/evaluate (double[], int, int, double)** is one of the changed methods in v2.2 with 9 deleted lines and 17 added lines. There are 3 direct test cases call this method, one of them is in Fig. 6(a). We observe after execute the test cases that the difference in execution time between the two versions across the different workload is very small nearly 0 as shown in Fig. 7(a). We notice that the dataset value is small size as line marked red in Fig. 6(a), when we modify the input test case with large dataset values as line 4 (green line) in Fig. 6(b), the difference of execution time was increased as shown in Fig. 7(b). This large variance in execution time indicates a performance regression with larger data sets. These results show that the size of dataset and analysis of the source code is effective in detecting performance regressions.

5.3. Detecting performance improvement in apache commons IO

Apache Commons IO is the third application we used to evaluate our approach. In this section, we analyze the results of comparing the performance of two versions (v2.0 and v2.1). Before version 2.1 especially from version v1.1, developers of Commons IO created a new method called readFileToByteArray() in class FileUtils which is called another method called toByteArray(InputStream input) in class IOUtils. The problem is the method readFileToByteArray() consumed a long time from the time of creation (v1.1) in 2005 until v2.1 in 2011. The reason for this long time is the calling of method toByteArray(InputStream input). In 2010, the problem was discovered and reported[10] with a new solution which is adding a new method toByteArray(InputStream input, int size) for toByteArray. After this modification method readFileToByteArray() now called the new method for toByteArray(). This reported issue was taken around 6 years to resolve.

[10] https://issues.apache.org/jira/browse/IO-251.

(a)

```
public void testPercentile() {
    double[] d = new double[] {1, 3, 2, 4};
    Percentile p = new Percentile(30);
    assertEquals(1.5, p.evaluate(d), 1.0e-5);
    p.setQuantile(25);
    assertEquals(1.25, p.evaluate(d), 1.0e-5);
    p.setQuantile(75);
    assertEquals(3.75, p.evaluate(d), 1.0e-5);
    p.setQuantile(50);
    assertEquals(2.5, p.evaluate(d), 1.0e-5);

    // invalid percentiles
    try {
        p.evaluate(d, 0, d.length, -1.0);
        fail();
    } catch (IllegalArgumentException ex) {
        // success
    }
    try {
        p.evaluate(d, 0, d.length, 101.0);
        fail();
    } catch (IllegalArgumentException ex) {
        // success
    }
}
```

(b)

```
public void newTestCase(){
    final double CONST_NUMBER = 18.;

    DescriptiveStatistics ds = new DescriptiveStatistics();
    for(int i = 0; i<100000; i++) { //100000  values
    ds.addValue(CONST_NUMBER);
    }

    ds.getPercentile(50);
}
```

Fig. 6. Different test case for the same changed method evaluate(). (a) show the test case with small dataset. (b) show new test case with large dataset.

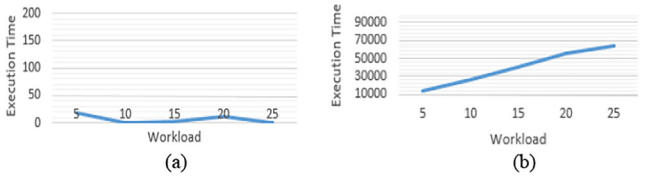

Fig. 7. Difference of execution time for the same changed method with different data set size as represented in y-axis through different workload (number of users) as represented in x-axis.

Our approach identifies the changed methods and their relations in v2.1. Among these changed methods, two methods mentioned in Table 5, the changed method readFileToByteArray() and the new added method toByteArray(InputStream input, int size). The two methods have direct relevant test cases in the new version v2.1 but the new added method does not exist in the previous version v2.0. So we cannot detect the performance of the new added method directly but we can detect its performance through the methods which is called it. Among these caller methods readFileToByteArray() which is one of changed method we measure its performance also. As shown in Fig. 8, the difference of execution time through different workload was negative values because the execution time of the new version v2.1 is less than the previous version v2.0.

5.4. Detecting performance improvement in Xalan

Xalan is one of the open source applications we used to detect performance problems. Some versions of Xalan have their own performance test suite. We didn't depend on those performance test suites in evaluating our approach because they have two problems. Firstly, those test suites are not provided for all versions and each commits. Secondly, these performance test cases do not cover all branches of each changed code portions. In our paper we depend on commits or versions which are set of commits and need to cover all changed code. So we generate test cases for changed code.

In this section, we analyze the results of comparing the performance of two versions (v2.7.1 and v2.7.2). Some of changed code portions depends on external files as inputs and the test cases we generate cannot cover this kind of code. So we generate manual test cases for them. One of these manual generated test case that we used in Xalan is a changed method called compose() in class ElemLiteralResult. The source code of this changed method rely on composing a template for stylesheet with some attributes to cover all branches of changed code. We generate manual test case to cover all branches of source code for testing the performance of changed code.

As shown in Fig. 9, the line of the difference execution time for the two versions is negative though different workload. That indicate that the performance of the changed method ElemLiteralResult.compose() has improved in v2.7.2 because the execution time of v2.7.2 is less than the execution time of v2.7.1.

5.5. Detecting performance regression in apache commons math using generated test case

Referring to (see section B) which is detecting performance regression cause by the changed method evaluate() in Percentile class. The method has JUnit test case already but the results of this test case do not define the performance regression. So we write manual test cases for defining the performance regression. We generate test case for this changed method using our approach and evosuite as shown in Fig. 10. The results of the generated test cases using large workloads as shown in Fig. 11 defined that the changed method caused performance regression.

5.6. Validating PerfDetect

We picked two existing approaches [1,3] for performance regression and evaluated our approach against them. Those approaches were selected due to utilizing the same real applications (Agilefant and Commons Math) that we utilized in our evaluation. We discuss our results against their results for each application.

5.6.1. AgileFant

Results of detecting performance regression for Agilefant between two versions (v3.2 and v3.3) using another approach as shown in Table 6, they compare between two changed methods. We validate their results with our results. Their results conclude that method (a) has relatively shorter average total execution times and has less contribution to performance regressions compared to method (b) which have much larger averages of the total execution times in v_i as compared to the ones in v_{i-1}.

When comparing these results with our results, firstly method (a and b) in Table 6 are detected in the results of our changed methods in Table 1 (4) and Table 2 (12). Secondly, Result of method (a) in another approach classified as not causing performance regression and our approach detects that it is not causing performance regression. Table 7 shows that another approach detected one method (b) caused performance regression, while our approach detected 4 methods that may cause performance regression including the same method (b) which is classified by another approach as causing performance regression. These results show that PerfDetect can detect changed methods causing performance

Table 5
Changed methods which have relevant related test cases in Commons IO.

Method		Added Lines	Deleted Lines	Test Case
1	org/apache/commons/io/IOUtils/toByteArray()	All method	0	3
2	org/apache/commons/io/FileUtils/readFileToByteArray()	1	1	2

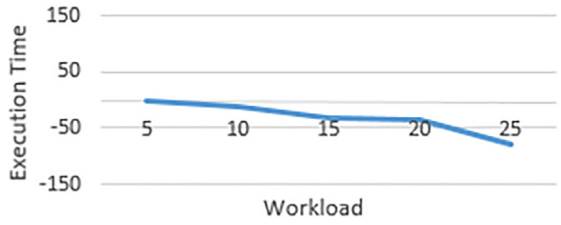

Fig. 8. Difference of execution time for a changed method in Commons-IO as represented in y-axis through different workload (number of users) as represented in x-axis.

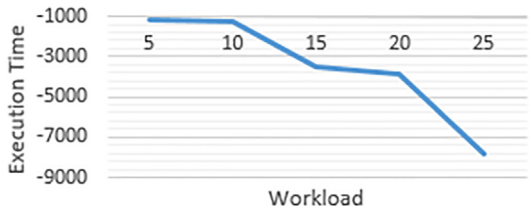

Fig. 9. Difference of execution time for a changed method in Xalan as represented in y-axis through different workload (number of users) as represented in x-axis.

```
public void test35()  throws Throwable  {
    Percentile percentile0 = new Percentile();
    assertEquals(50.0, percentile0.getQuantile(), 0.01);
    assertNotNull(percentile0);

    double[] doubleArray0 = new double[28];
    doubleArray0[2] = 1996.1795937687925;
    doubleArray0[3] = (-1425.59877891);
    doubleArray0[4] = 1996.1795937687925;
    double double0 = percentile0.evaluate(doubleArray0);
    assertEquals(0.0, double0, 0.01);
    assertEquals(28, doubleArray0.length);
    assertEquals(50.0, percentile0.getQuantile(), 0.01);
}
```

Fig. 10. Automatic generated test case for the changed method Percentile.evaluate () in Commons-Math.

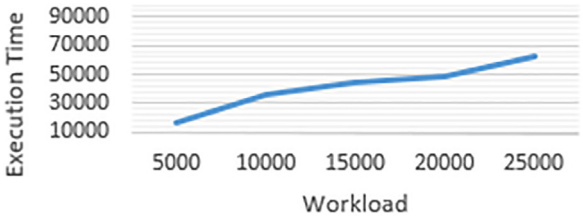

Fig. 11. Difference of execution time for a changed method in Commons-Math as represented in y-axis through different workload (number of users) as represented in x-axis.

regressions more than another approach and that is because we use source code analysis and cover all changed methods.

5.6.2. Apache commons math

Results of Commons Math in another approach between two versions (2.0 and 2.2) as the same as our results because they use source code analysis but the difference is in time of detecting performance regressions. Another approach runs all test cases in the applications and has taken 48 min in v2.2 and 30 min in v2.0. The time of our approach did not exceed 80 s for v2.2 and 11 s for v2.0. That is because our approach depends on running only the test cases related to the changed methods not all test cases in the application. All these results show that PerfDetect can be used to effectively identify the changes that are responsible for performance regressions.

Table 6
Examples of code changes in Agilefant detected by another approach [3].

	Method Name
a	fi.hut.soberit.agilefant.business.impl.SearchBusinessImpl.taskListSearchResult
b	fi.hut.soberit.agilefant.business.impl.SettingBusinessImpl.retrieveByName

Table 7
Comparison between our approach and another approach in detecting performance regression across v3.2 and v3.3 in Agilefant.

Approaches	# identified problems
Luo et al. [1]	2
Our approach	4

In order to avoid manual generated test cases, we generate automatic test case for detecting performance regression. PerfDetect can detect performance regression relying on automatic generated test cases.

5.6.3. Apache commons IO

Results of our proposed approach when comparing the source code between two versions (v2.0, v2.1) observed performance improvement. PerfDect detects the changed method readFileToByteArray() in class FileUtils and observe that the performance of this method has improved more than 50% comparing to the previous version 2.0. The cause of this performance improvement that the changed method call a new added method instead of old one that was taken a long time.

5.6.4. Xalan

Results of our proposed approach when comparing the source code between two versions (v2.7.1, v2.7.2) observed performance improvement. PerfDect detects the changed method ElemLiteralResult.compose() and observes that the performance of this method has improved slightly comparing to the previous version 2.7.1. The cause of this performance improvement that the changed method changed the method used to locate an element of an array. This results prove that PerfDetect can detect the performance whatever that performance is improvement or regression and identify the cause of performance change.

6. Threats to validity

The external validity of our study is threatened by three main issues. (1) We applied our approach on four applications only.

Finding open source applications, with a JUnit test suite of decent coverage is a challenging task. To fit our experimental setup, any candidate open source application needed to satisfy a set of conditions: (i) the presence of performance regressions or performance improvements in one version, that were fixed in a subsequent version; (ii) the presence of an existing test suite that covers the source code changes that introduced such performance regressions; (iii) the absence of any additional test code constructs (like EasyMock code constructs) that would prohibit applying our approach. We had examined a large set of open source applications (120 applications), and only four of those applications matched the three above criteria. Furthermore, one of those four did not match the third criterion (iii), so we had to modify all its relevant tests in order to apply our approach. However, extending those four systems with an additional set of system remains future work. (2) We had a small number of performance regressions within each of the chosen system. While this is true, all of our chosen performance regressions were real faults, rather than seeded ones, hence reflect real performance problems rather than simulated ones. Furthermore, we picked performance regressions that were used by other published approaches, to increase the credibility and trust in our study. (3) Our test case generation step relied on a single test generation tool named Evosuite. However, we mainly picked EvoSuite because it had various advantages as follows: (1) Evosuite depends on genetic algorithms to generate JUnit test cases the selected classes which was useful for our approach to generate test cases for selected changed classes only not all the project to to avoid time consuming. (2) EvoSuite extended with a memetic algorithm enabling a global search algorithm for achieving high coverage with different types (individual statements, branches, outputs and mutation testing). Our approach need high coverage in generating test cases for achieving the changed code that cause performance regressions.

Another threat to the validity of the results, is the evaluation done to compare the proposed approach's ability to detect performance regressions, in comparison to other approaches for performance regression. For that comparison, only two open source systems were utilized. That is mainly attributed to two main reasons. (1) One of the other approaches uses test inputs on web-based or desktop applications, hence would be hard to compare to the proposed approach. The proposed approach utilized unit tests on desktop applications mainly. Hence, for that other approach, we could only pick one of their analyzed systems. Yet, for that analyzed system, we could successfully identify the same performance regressions that they identified. We could also identify two additional performance regressions that were totally missed by their approach (see Table 7). (2) The proposed approach requires the presence of code changes that introduce performance regressions. As previously mentioned, locating such code changes is a non-trivial task, that should be followed by locating/ generating units tests to detect performance regressions. However, in our future work to intend to include more systems, and more approaches in such comparison

The proposed approach suffers from one basic limitation which is its inability to identify performance regressions introduced by newly added, rather than changed, source code. This limitation is mainly attributed to the absence of execution time data for such newly added source code. Yet, such limitation remains as an open issue that we plan to address in our future work.

7. Related work

Many recent approaches (e.g. Heger et al. [1]) provide root cause analysis by making use of unit tests and the version history of the system under development but these approaches are time-consuming because they demand running a complete set of test cases, this may not scale well for systems with huge test suites. For solving this problem, other approaches (e.g. Yu et al. [23]) use a regression test selection technique, to reduce the number of test cases that must be run on a changed program. These techniques prioritize functional tests but are not tailored to select such tests that would reveal performance regressions. Reichelt et al. [24] is depending on regression test selection to define performance changes using artificial test cases.

Several source code analysis techniques can be used to analyze the set of program elements that may be affected by the change. They can be divided into three categories: static analysis techniques [27–29], dynamic analysis techniques [7–10] and version control history-based techniques [30,31]. Huang et al. [32] estimated the risk of various code changes and tagged those changes likely leading to performance regressions. This approach relies only on static analysis techniques and focuses on specific types of performance regression such as intra-procedural paths and loop termination conditions. Recent work by Luo et al. [3] uses a genetic algorithm and dynamic analysis to investigate a large input search space that might lead to performance regressions in web context then, mine their execution traces to prioritize code changes associated with these input-specific performance regressions. In this work, there are some limitation such as: (a) the test inputs are randomly generated, (b) the changes identified to cause performance regressions, are mainly a suspected list, rather than a definite cause of the performance regression, (c) newly added features are not included in the conducted analysis for performance regressions identification, and (d) the analysis demands executing the whole system, for all its analyzed versions, thus might not scale well to large-scale software systems.

Performance regression testing has been studied in different application domains like desktop, web and mobile. Gomez et al. [33] presented an approach that automatically detects UI performance degradations in Android apps while considering context differences. This approach relies on poor GUI responsiveness and number of frames per second for measuring performance regression. Many approaches are interested in detecting performance changes in different languages as Python language [34].

As per our knowledge, existing approaches focus on analyzing complete systems to detect performance regressions, rather than focusing their analysis on only the modified/newly added portions of those systems on specific system parts to detect performance regressions. Furthermore, none of the existing performance regression detection approaches has employed source code analysis techniques to properly examine the tested systems for performance regressions.

Automatic test case generation is effective and frequently raised by software developers in software testing. However, there is little research at this aspect. Thus, there are different approaches in generating test cases. Alrawashed et al. research proposes an automatic test cases generator approach relying on use case description model to reduce the test complexity and enhance the percentage of test coverage [35]. Depending on use case description model is not available in all software systems. Other approaches [15,25,26] employ genetic algorithms to generate test cases or optimize test suites to a smaller subset, but these approaches focus on generating new test cases to achieve high code coverage, rather than reveal performance regressions. Genetic algorithms are widely used in different areas of software engineering, and software testing in particular [36]. Grosso et al. proposed an approach that uses GAs to generate test cases that cause buffer overflows and integrate domain knowledge with slicing and static analysis to reduce the search space [37]. Fraser et al. proposed EvoSuite, that uses GAs to optimize whole test suites to smaller subsets that satisfy certain coverage criteria [26]. Since EvoSuite

works only locally on the individual statements, they extended EvoSuite with a memetic algorithm enabling a global search algorithm to increase branch coverage [15]. Test suite augmentation techniques are used to generate test cases that cover code changes or code elements affected by changes [38,39]. Genetic algorithms used to generate test cases or optimize test suites, but these approaches focus on generating new test cases to achieve high code coverage, rather than reveal performance regressions and they are time-consuming for generating all test cases. existing approaches focus on analyzing complete systems to detect performance regressions, rather than focusing their analysis on only the modified/newly added portions of those systems on specific system parts to detect performance regressions.

8. Conclusion and future work

In this paper we propose a novel approach PerfDetect that aims to identify performance regressions and their root causes across two system versions. The approach depends on: (1) detecting changed code across the two system versions through version control systems, (2) identifying the relevant test cases to the changed code within the current version to compare its performance against a reference version, (3) analysis of performance problems to identify code portions and their root causes that may cause performance regression. We evaluated PerfDetect on two open-source applications as real applications written in Java with their test cases. The results demonstrate that PerfDetect can effectively recommend changed code portions caused performance regressions in comparison to other approaches.

The next step as a future work is to detect performance regressions for newly added method. Furthermore, we aim to apply our approach on more open source systems, while utilizing several test case generations tools.

Conflict of interest statement

There is no conflict of interest.

References

[1] Heger, Christoph, Jens Happe, Roozbeh Farahbod. Automated root cause isolation of performance regressions during software development. In: Proceedings of the 4th ACM/SPEC international conference on performance engineering, pp. 27–38, 2013.
[2] Pham, Raphael, Yauheni Stoliar, Kurt Schneider. Automatically recommending test code examples to inexperienced developers. In: Proceedings of the 2015 10th joint meeting on foundations of software engineering, pp. 890–893. 2015.
[3] Luo, Qi, Denys Poshyvanyk, Mark Grechanik. Mining performance regression inducing code changes in evolving software. In: 2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR), pp. 25-36. IEEE, 2016.
[4] GitHub, 2020. https://github.com/.
[5] GitHub API, 2020. http://github-api.kohsuke.org/.
[6] Call Hierarchy plugin, 2020. http://eclipse-tools.sourceforge.net/call-hierarchy/index.html.
[7] Law, James, Gregg Rothermel. Whole program path-based dynamic impact analysis. In: 25th international conference on software engineering, 2003. Proceedings, pp. 308–318. IEEE, 2003.
[8] Orso, Alessandro, Taweesup Apiwattanapong, Mary Jean Harrold. Leveraging field data for impact analysis and regression testing. ACM SIGSOFT Software Engineering Notes 28, no. 5 (2003): 128–137.
[9] Orso, Alessandro, Taweesup Apiwattanapong, James Law, Gregg Rothermel, Mary Jean Harrold. An empirical comparison of dynamic impact analysis algorithms. In: Proceedings. 26th international conference on software engineering, pp. 491–500. IEEE, 2004.
[10] Apiwattanapong, Taweesup, Alessandro Orso, Mary Jean Harrold. Efficient and precise dynamic impact analysis using execute-after sequences. In: Proceedings of the 27th international conference on software engineering, pp. 432–441, 2005.
[11] ContiPerf, 2020. http://databene.org/contiperf.
[12] JUnitPerf, 2020. https://github.com/clarkware/junitperf.
[13] JProfiler, 2020. https://www.ej-technologies.com/products/jprofiler/overview.html.
[14] Evosuite, 2020. http://www.evosuite.org/.
[15] Fraser, Gordon, Andrea Arcuri, Phil McMinn. Test suite generation with memetic algorithms. In: Proceedings of the 15th annual conference on Genetic and evolutionary computation, pp. 1437–1444, 2013.
[16] EasyMock, 2020. http://easymock.org/.
[17] Agilefant, 2020. http://agilefant.com/.
[18] Commons math: The apache commons mathematics library, 2012. http://commons.apache.org/math.
[19] Commons IO: The apache commons IO library, 2020. https://commons.apache.org/proper/commons-io/.
[20] Xalan, 2020. https://xalan.apache.org/.
[21] Mackinnon, Tim, Steve Freeman, Philip Craig. Endo-testing: unit testing with mock objects. Extreme programming examined (2000): 287–301.
[22] Thomas D, Hunt A. Mock objects. IEEE Software 2002;19(3):22–4.
[23] Yu, Tingting, Witawas Srisa-an, Gregg Rothermel. SimRT: An automated framework to support regression testing for data races. In: Proceedings of the 36th international conference on software engineering, pp. 48–59, 2014.
[24] Reichelt, David Georg, Stefan Kühne. How to detect performance changes in software history: performance analysis of software system versions. In: Companion of the 2018 ACM/SPEC international conference on performance engineering, pp. 183–188, 2018.
[25] Gross, Florian, Gordon Fraser, Andreas Zeller. Search-based system testing: high coverage, no false alarms. In: Proceedings of the 2012 international symposium on software testing and analysis, pp. 67–77, 2012.
[26] Fraser, Gordon, Andrea Arcuri. Evolutionary generation of whole test suites. In: 2011 11th international conference on quality software, pp. 31–40. IEEE, 2011.
[27] Arnold, Robert S, Shawn A Bohner, Impact analysis-towards a framework for comparison. In: 1993 conference on software maintenance, pp. 292–301. IEEE, 1993.
[28] Turver Richard J, Munro Malcolm. An early impact analysis technique for software maintenance. J Software Maintenance: Res Practice 1994;6(1):35–52.
[29] Arnold, Robert S. Software change impact analysis. IEEE Computer Society Press, 1996.
[30] Zimmermann Thomas, Zeller Andreas, Weissgerber Peter, Diehl Stephan. Mining version histories to guide software changes. IEEE Trans Software Eng 2005;31(6):429–45.
[31] Sherriff, Mark, Laurie Williams. Empirical software change impact analysis using singular value decomposition. In: 2008 1st international conference on software testing, verification, and validation, pp. 268–277. IEEE, 2008.
[32] Huang, Peng, Xiao Ma, Dongcai Shen, Yuanyuan Zhou. Performance regression testing target prioritization via performance risk analysis. In: Proceedings of the 36th international conference on software engineering, pp. 60–71. 2014.
[33] Gómez, María, Romain Rouvoy, Bram Adams, Lionel Seinturier. Mining test repositories for automatic detection of ui performance regressions in android apps. In: Proceedings of the 13th international conference on mining software repositories, pp. 13–24. 2016.
[34] Chen, Jie, Dongjin Yu, Haiyang Hu, Zhongjin Li, Hua Hu. Analyzing performance-aware code changes in software development process. In: 2019 IEEE/ACM 27th international conference on program comprehension (ICPC), pp. 300–310. IEEE, 2019.
[35] Alrawashed Thamer A, Almomani Ammar, Althunibat Ahmad, Tamimi Abdelfatah. An automated approach to generate test cases from use case description model. Comput Modeling Eng Sci 2019;119(3):409–25.
[36] Le Goues Claire, Nguyen ThanhVu, Forrest Stephanie, Weimer Westley. Genprog: a generic method for automatic software repair. IEEE Trans Software Eng 2011;38(1):54–72.
[37] Del Grosso, Concettina, Antoniol G, Massimiliano Di Penta. An evolutionary testing approach to detect buffer overflow. In: Student Paper Proceedings of the international symposium of software reliability engineering (ISSRE), St. Malo, France. 2004.
[38] Xu, Zhihong, Yunho Kim, Moonzoo Kim, Gregg Rothermel. A hybrid directed test suite augmentation technique. In: 2011 IEEE 22nd international symposium on software reliability engineering, pp. 150–159. IEEE, 2011.
[39] Xu, Zhihong, Myra B Cohen, Gregg Rothermel. Factors affecting the use of genetic algorithms in test suite augmentation. In: Proceedings of the 12th annual conference on genetic and evolutionary computation, pp. 1365–1372, 2010.
[40] Shi-kuo, Chang, ed. Data structures and algorithms. Vol. 13. World scientific, 2003.