2016 IEEE/ACM 13th Working Conference on Mining Software Repositories

Mining Performance Regression Inducing Code Changes in Evolving Software

Qi Luo
College of William and Mary
qluo@cs.wm.edu

Denys Poshyvanyk
College of William and Mary
denys@cs.wm.edu

Mark Grechanik
University of Illinois at Chicago
drmark@uic.edu

ABSTRACT

During software evolution, the source code of a system frequently changes due to bug fixes or new feature requests. Some of these changes may accidentally degrade performance of a newly released software version. A notable problem of regression testing is how to find problematic changes (out of a large number of committed changes) that may be responsible for performance regressions under certain test inputs.

We propose a novel recommendation system, coined as PerfImpact, for automatically identifying code changes that may potentially be responsible for performance regressions using a combination of search-based input profiling and change impact analysis techniques. PerfImpact independently sends the same input values to two releases of the application under test, and uses a genetic algorithm to mine execution traces and explore a large space of input value combinations to find specific inputs that take longer time to execute in a new release. Since these input values are likely to expose performance regressions, PerfImpact automatically mines the corresponding execution traces to evaluate the impact of each code change on the performance and ranks the changes based on their estimated contribution to performance regressions. We implemented PerfImpact and evaluated it on different releases of two open-source web applications. The results demonstrate that PerfImpact effectively detects input value combinations to expose performance regressions and mines the code changes are likely to be responsible for these performance regressions.

CCS Concepts

•**Software and its engineering** → **Software performance; Software testing and debugging;**

Keywords

Performance regression testing, mining execution traces, change impact analysis, genetic algorithms

MSR'16, May 14-15, 2016, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2901739.2901765

1. INTRODUCTION

Performance is an important metric of software quality [60, 43], whereas performance testing is a vital activity that developers routinely perform during software development and maintenance to ensure quality [19]. During software evolution, a number of code changes are committed, and some of them may be responsible for performance regressions. A performance regression is a situation in which an *application under test* (AUT) exhibits unexpectedly worsened performance in a new release as compared to the previous version for the same input values and for a given *workload* (i.e., the number of users, their requests and frequencies of interactions). Stakeholders are interested in understanding code changes behind these regressions.

Performance regression testing is challenging due to at least the following reasons. Firstly, modern software systems evolve rapidly. Many of them follow agile-driven cycles and release new versions in short iterations [18]. With a large number of commits submitted, the cost of detecting performance regressions and linking code changes to performance behaviors increases drastically. Therefore, performance regression testing is usually performed continuously during software maintenance [15, 41]. Secondly, detecting performance regressions and locating the associated code changes for specific inputs in AUTs with large spaces of input combinations are non-trivial and time-consuming tasks [61].

Let's consider a simplified scenario for detecting performance regressions. Assume there are two versions of an AUT, a newly released version (v_{i+1}) and a previous version (v_i). Programmers commit a number of changes between these two versions. Given the same test inputs, v_i and v_{i+1} the application may exhibit different performance behaviors with respect to its execution time. The test inputs that lead to worsened performance (e.g., longer execution time) in v_{i+1} but not in v_i are the desired inputs that may expose new performance regressions. Their corresponding execution traces are helpful for troubleshooting [41]. In order to find such inputs, stakeholders need to iterate through a large number of input combinations while mining the execution traces for both of v_i and v_{i+1} with the same inputs to monitor changes in performance for each input set. It is challenging for stakeholders to mine a large body of execution traces for identifying the ones can expose potential performance regressions and linking the inputs to these traces. Once such inputs are found (manually or automatically), the corresponding execution traces need to be further examined to detect changes responsible for observed performance regres-

sions. Unfortunately, this process is domain and knowledge dependent, oftentimes manual and expensive.

We propose a novel recommendation system, PERFIMPACT, to automatically recommend inputs and code changes for programmers that may be closely related to performance regressions using a combination of search-based input profiling [69] and change impact analysis [51]. The search-based input profiling has been extended to execute two different releases of AUT (v_i and v_{i+1}) independently with the same input values, mine execution traces to link inputs with AUT's behaviors, and use a genetic algorithm as a search heuristic for exploring the input value combinations for finding the ones likely exposing performance regressions. After the inputs are selected, PERFIMPACT mines the execution traces generated with these inputs, and uses change impact analysis to rank each code change based on its contribution to the AUT's performance regression(s). The code changes having significant impact on AUT's performance degradation in v_{i+1} are marked as problematic for follow-up code reviews. The goal of PERFIMPACT is to improve effectiveness of performance regression testing via identifying input combinations than worsen performance behaviors (i.e., longer execution time) in v_{i+1}, and mining the corresponding execution traces to prioritize code changes likely responsible for these regressions. It is possible that some code changes with longer execution time implement new features or fix bugs, not necessarily leading to performance regressions. Our approach may not precisely locate root cases behind performance regressions, but provide a ranked list of code changes potentially leading to performance regressions that can be used as a starting point for programmers in regression testing. This paper makes the following contributions:

- We propose a novel recommendation system, PERFIMPACT, that relies on search-based input profiling to expose performance regressions manifested in newer software versions, mines the corresponding traces, and uses change impact analysis to prioritize the code changes likely responsible for these performance regressions;
- We empirically evaluated PERFIMPACT on different releases of two open-source web applications, Agilefant ($v_{3.2}$, $v_{3.3}$, and $v_{3.5}$) and JPetStore ($v_{3.0.0}$ and $v_{4.0.5}$) containing numerous real changes. The results demonstrate that PERFIMPACT is able to effectively explore the combinations of input values and identify performance regressions between different releases. The results also demonstrate that PERFIMPACT can effectively recommend the changes (both real and injected) likely responsible for the identified regressions;
- We have made PERFIMPACT and the experimental results publicly available in our online appendix [7].

2. PROBLEM STATEMENT

In this section, we survey the state of the art and practice in performance regression testing, discuss an illustrative example, and describe the problem statement.

2.1 State of the Art and Practice

Many recent approaches aim at detecting performance regressions by comparing the values of different performance metrics (e.g., performance counters) in two system versions [60, 61, 52]. Typically, they execute the same test cases in each version and use control charts to check if the performance of a target test in v_{i+1} is similar to the performance of a baseline test in v_i. Other approaches use statistical methods, such as ANOVA, to detect performance differences between v_{i+1} and v_i [41]. All these approaches require running a complete set of test cases for detecting regressions. However, since performance testing is usually time-consuming [60], it is imperative to identify a subset of effective inputs or test cases more likely to exhibit performance regressions. While techniques for selecting regression tests have been proposed and evaluated in the context of functional testing [26, 49, 56, 71, 72], generating and selecting performance regression tests still remains a significant challenge.

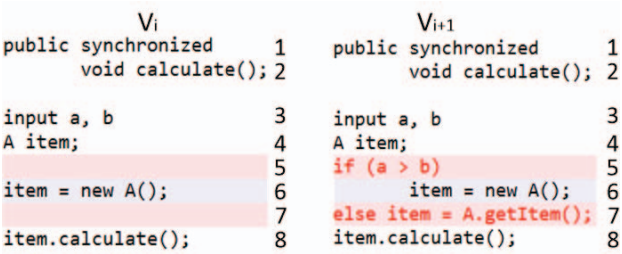


Figure 1: A performance regression example due to possible thread blocking.

Understanding which code changes are responsible for particular performance regressions poses to be even more challenging problem. Precisely pinpointing changes (out of thousands of commits) that may be responsible for performance regressions (for certain inputs) is a fairly involved task, requiring deep knowledge of the AUT's source code, behavioral semantics, and even change history. The closest approach to address this problem is the one by Huang *et al.* who proposed a model for estimating the risk of each commit and tagging commits likely leading to performance regressions [43]. This solution relies on static analysis and focuses on specific types of performance regressions, such as dramatic cost difference in intra-procedural paths and loop termination conditions affected by code changes (it does not identify changes responsible for input-specific bottlenecks).

2.2 An Example Performance Regression

Let's consider the example shown in Fig. 1. This example illustrates that understanding AUT's behaviors and their relationships to input values (and combinations of inputs) is critical for detecting performance regressions. The example shows code snippets in two versions of a system, v_i and v_{i+1}. In both versions, lines 1-2 declare method *calculate*() as a synchronized method. Line 3 presents input variables a and b, and line 4 the object *item* of the type A is instantiated. In v_i, lines 5-7 assign a new instance to *item*; while, in v_{i+1}, lines 5-7 assign a new instance to *item* or invoke method *getItem*() to assign an existing instance to *item*, depending on the result of the branch condition in line 5. In both v_i and v_{i+1}, *item* calls method *calculate*() in line 8. Note that *calculate*() is a synchronized method, so if it is called with the same instance in multiple threads simultaneously, the threads will be blocked. However, in v_{i+1}, *item* is assigned an existing instance if the branch condition in line 5 is not satisfied. Thus, when multiple threads are executing concurrently and sharing the same instance of an *item*, method *calculate*() may be blocked, which can lead to a performance regression for certain inputs of a and b in v_{i+1}, but not in v_i. Moreover, even if the input values leading to this performance regression are identified, it may be difficult to locate code changes responsible for this performance

regression. If we simply rely on total execution time to evaluate performance, we would be able to observe performance degradation, for certain inputs, for method *calculate*(). Yet, in this case, the actual changes responsible for the performance regression are those in line 5 and line 7 in v_{i+1}.

2.3 The Problem Statement

In order to prioritize code changes likely responsible for performance regressions, first we need to find input combinations that execute the code changes which may trigger performance regressions. As an AUT evolves, a large number of changes are made between v_{i+1} and v_i, such as code changes, database restructuring, as well as changes in configuration files, potentially leading to performance regressions. In our paper, we only focus on the performance regressions caused by code changes. Static analysis techniques alone may not be suitable to solve this problem, since they are expensive and oftentimes language-dependent, whereas dynamic analysis techniques are likely to provide higher precision when understanding AUT's performance behaviors in terms of input values for detecting performance regressions. When running v_{i+1} and v_i with the same inputs, only certain combinations of inputs can trigger specific code changes that may cause AUT to take longer time to execute in v_{i+1} as compared to v_i. However, for non-trivial AUTs with large input spaces, the number of permutations of input values is too large to run in a reasonable amount of time. Also, it is nontrivial to mine a large body of execution traces for finding the ones likely to expose performance regressions. The first problem to solve is how to explore the large input space and mine the corresponding execution traces to effectively find a subset of inputs exposing performance regressions.

After finding the inputs triggering performance regressions, we aim at mining their execution traces to prioritize code changes associated with these input-specific performance regressions. The key problem here is how to link all code changes to AUT's performance behaviors and understand their impacts on observed performance regressions. Note that our approach is not precise root causes analysis of performance regressions. Instead, we propose to improve the effectiveness of performance regression testing for programmers by recommending a list of code changes likely responsible for performance regressions.

3. APPROACH

In this section, we describe our key ideas, algorithms, and the detailed workflow behind PerfImpact.

3.1 An Overview of Our Approach

PerfImpact rests on two key ideas: (1) rely on the search-based input profiling for mining execution traces to expose the AUT's performance degradations between two releases, v_{i+1} and v_i, and detecting input value combinations that maximize these degradations, and (2) mine execution traces and utilize change impact analysis to identify the code changes having significant impact on performance degradation for a given set of inputs.

Finding Inputs That Lead to Performance Regressions. The first key idea of PerfImpact is to rely on search-based input profiling [69] to mine execution traces for understanding AUT's performance behaviors, and use genetic algorithms (GAs) to explore different combinations of input values for finding the ones that take unexpectedly longer time to execute in v_{i+1} but not in v_i. Our hypothesis is that the input value combinations with larger execution time difference among two studied versions are more likely to trigger performance regressions. While search-based input profiling has been recently used for detecting performance bottlenecks in a given software version [69], PerfImpact instruments and runs two versions of the AUT with the same inputs independently. PerfImpact also defines a new fitness function aimed at mining execution traces to obtain the ones using more time to complete in v_{i+1} than in v_i and selecting input combinations associated with these executions. This fitness function is designed as a proxy for identifying inputs leading to performance regressions in v_{i+1}.

URL 1: http://localhost:8080/Agilefant/editUser.action
URL 2: http://localhost:8080/Agilefant/listTeams.action
URL 3: http://localhost:8080/Agilefant/editProduct.action?productId=5
URL 4: http://localhost:8080/Agilefant/editProduct.action?productId=8
URL 5: http://localhost:8080/Agilefant/ajax/retrieveProduct.action?productId=5
.....

Chromosome: 2, 18, 36, 27, 11, 13, 6, 43, 64, 12, 85, 49, 12, 53, 44, 78, 31, 47, 6

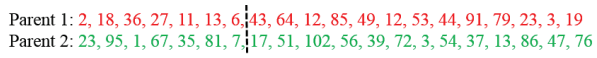
Figure 2: Examples of URLs and a chromosome in our GA implementation. Each number in the chromosome refers to a unique URL ID.

Identifying Code Change That Induce Performance Regression by Mining Execution Traces. The second key idea is to find the changes associated with the methods related to performance degradations. Specifically, PerfImpact obtains execution times of the invoked methods in v_{i+1} and v_i during profiling and compares their performance differences respectively. The methods with increased execution time in v_{i+1}, for the same inputs as in v_i, are tagged as potentially "problematic". Given a code change, PerfImpact relies on dynamic change impact analysis (CIA) [51] to mine execution traces and estimate a set of methods (i.e., an impact set) that is potentially impacted by this code change. Then, all the changes between v_{i+1} and v_i are ranked based on the performance of the methods in their respective impact sets. The changes that have more "problematic" methods in their impact sets are ranked higher. Conversely, the changes that have fewer or no "problematic" methods in their impact sets are ranked lower. The heuristic is that the higher ranked changes usually have more significant impact on performance regressions.

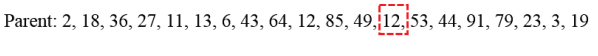
3.2 Search-based Input Profiling for Performance Regressions

Search-based input profiling mines a large body of execution traces and utilizes GAs to automatically search the input space for possible combinations of inputs responsible for the performance regressions. GAs are evolutionary algorithms that mimic the natural selection process to search for the solutions to optimization problems [42, 58], and have been widely used to generate test cases in the software testing domain [39, 45, 40]. In GAs, a solution or an individual is represented as a chromosome, which contains a sequence of genes. Typically, the initial individuals are generated randomly, and then GAs exploit a pre-defined fitness function to evaluate each individual. The fitter ones (i.e., parents) that have larger fitness values are selected to generate the individuals for the next generation (i.e., offsprings) via genetic operators, such as crossover and mutation.

The key idea behind our GA implementation is to identify the input combinations likely to expose performance regressions. In our implementation, an individual (i.e., a chro-



(a) The crossover operator in GAs.



(b) The mutation operator in GAs.

Figure 3: The examples of GA operators, crossover and mutation.

mosome) refers to a test case (or a set of inputs). Each chromosome contains a sequence of genes, referring to the inputs with different parameters. In case of a web-based application that takes URLs as inputs, the example of a chromosome encoding is shown in Fig. 2. Each URL is assigned an unique ID and a chromosome encoding represents a sequence of URL IDs. An URL input containing different parameters (e.g., URL 3 and 4 shown in Fig. 2) will be assigned different IDs. The implementation of crossover and mutation operators is illustrated in Fig. 3. The crossover operator selects a pair of parent chromosomes (i.e., ID sequences) and randomly chooses a cut point to swap these two sequences. The mutation operator takes a chromosome and changes the value of a selected gene (i.e., an ID) with another random value. The probabilities of these two operations are predefined as the crossover and mutation rates.

We define a fitness function to evaluate inputs and promote the ones that are more likely to trigger performance regressions. PerfImpact first mines execution traces to extract time information for each combination of inputs, then measures the inputs using the *time difference*, which is defined as the difference between the times it takes v_{i+1} and v_i to execute with the same inputs. The larger the *time difference*, the higher the probability that the corresponding inputs might lead to performance regressions. We define the fitness function as shown in Eq. 1, where I_j is a set of inputs selected from the whole AUT input set (i.e., I_{all}), td_j is the *time difference* for input I_j, t_j is the time it takes AUT to execute I_j, the superscripts 'i' and '$i+1$' refer to v_i and the v_{i+1} software releases respectively.

$$td_j = t_j^i - t_j^{i+1} \quad (1)$$

Our GA implementation is outlined in Alg. 1, which takes the whole AUT input set (I_{all}) and two releases (v_i, v_{i+1}) as inputs, and outputs the sets of inputs (I) for which performance regressions are observed. In detail, the initial population is selected randomly from I_{all} (1). Then crossover and mutation operators are executed with the pre-defined rates (r_c, r_m) on the initial population to generate new individuals (3-4). After that, each individual is sent as an input to v_i and v_{i+1}, and two traces are collected during the profiling (5-7). Then the fitness value is calculated based on the pre-defined fitness function (Eq. 1) for each individual (8-9). The fitter ones are selected to create the next generation (10). The above process repeats until the termination criterion is reached (2), and then sets of inputs (I) are returned (11-12). Typically, there are two types of termination criteria. One is a pre-defined maximum number of generations and the other one is the average fitness value. When the maximum number of generations is reached or the children's average fitness value does not increase significantly as compared to their parents' average fitness value (the increased percentage is less than a pre-defined threshold), the evolution process is terminated. The values of two types of termination criteria are settled experimentally (Section 4.3).

Algorithm 1: The Genetic Algorithm.

Input : Input (I_{all}), Two software releases (v_i, v_{i+1})
Output: Sets of inputs (I) that might trigger performance regressions.

1: Initial population $I \leftarrow I_{all}$
2: **while** Termination criterion is not satisfied **do**
3: $I \leftarrow crossover(I, r_c)$
4: $I \leftarrow mutation(I, r_m, I_{all})$
5: **for all** $I_j \in I$ **do**
6: $t_j^i \leftarrow$ Run I_j in v_i
7: $t_j^{i+1} \leftarrow$ Run I_j in v_{i+1}
8: $td_j \leftarrow t_j^{i+1} - t_j^i$, where $td_j \in TD$
9: **end for**
10: $I \leftarrow selectPopulation(I, TD)$
11: **end while**
12: **return** I

3.3 Identifying Performance Regression Inducing Changes via Mining

In general, performance regressions are exposed when some specific methods experience longer execution time in v_{i+1}. PerfImpact relies on path-based dynamic CIA [51] to identify the changes leading to performance regressions. For each change, the impact analysis is used to build an impact set containing all the methods that are potentially impacted by this change. PerfImpact mines execution traces to understand the performance of the impacted methods in two releases to rank the changes. *The key hypothesis here is that if the methods in the impact set exhibit longer execution times in v_{i+1} but not in v_i, for the same sets of inputs, then it is more likely that a change for this impact set is responsible for the observed performance regression.* Obviously, there may be cases where multiple inputs and changes are responsible for one or multiple performance regression(s) (i.e., some fault interaction may be present [24]). Note that CIA may not be helpful to accurately locate the code causing performance regressions. However, our goal is to pinpoint a starting point (i.e., changes related to observed performance regressions) for a detailed root cause analysis that needs to be performed by developers. In our paper, the code changes are extracted at the method level granularity. In particular, we consider changes in a method between v_{i+1} and v_i involving additions, modifications or deletions to the body, signature, or a return type, excluding comments.

The impact analysis technique that we rely upon in our implementation considers a change's impact that propagates along any (and only) dynamic paths that pass through the change [51]. Given a change c, only the methods, which are called after c and which are in the call stack after c returns, are added into the impact set. For example, three execution traces are shown in Fig. 4. Given a method a, a_e represents a method's entry and a_r represents a method's return. x represents the execution termination. In fig. 4, in the first execution, m is called first, then m calls b, b calls c, c calls f, f and c return, b returns, m returns, and finally the execution terminates. Assuming that the method c has been changed, its impact set in the first execution is {b, f, m}, since f is called after c, and b, m are in the call stack

Algorithm 2: Ranking changes for a given set of inputs.

Input : Changes $C(c_1, c_2, ...)$, Impact sets $IM(im_{c_1}, im_{c_2}, ...)$, Method Statistics.
Output: Ranked lists of changes RC.

1: **for all** $c_k \in C$ **do**
2: **for all** $m_q \in im_{c_k}$ **do**
3: $det_{m_q} = mt^{i+1}_{m_q} - mt^{i}_{m_q}$
4: $sdet_{c_k} += det_{m_q}$, where $sdet_{c_k} \in SDET$
5: **end for**
6: **end for**
7: $RC \leftarrow RANK(C, SDET)$
8: **return** RC

(1) $m_e, b_e, c_e, f_e, f_r, c_r, b_r, m_r, x$
(2) $m_e, a_e, d_e, a_e, a_r, d_r, c_e, f_e, f_r, c_r, a_r, m_r, x$
(3) $m_e, b_e, c_e, c_r, e_e, e_r, b_r, m_r, x$

Figure 4: Three sample execution traces of an AUT.

after c returns. Similarly, its impact set is $\{a, f, m\}$ in the second execution, and its impact set is $\{b, e, m\}$ in the third execution. Thus, the final impact set for the method c is the union of these three sets, which is $\{a, b, e, f, m\}$.

In PerfImpact, a trace is collected for one set of inputs. We considered the trace segment of one distinct input (i.e., a URL) as an execution, so each trace can be divided into different executions corresponding to different inputs. In CIA, when one trace contains multiple executions, the backward and forward searching do not cross the termination symbol of each execution (i.e., x in Fig. 4). For a web application, one set of inputs refers to a sequence of URLs, thus a trace is collected for each sequence of URLs. Each trace can be divided into different trace segments for different URLs. For example, if there are 50 URLs in one set of inputs, the corresponding trace is divided into 50 trace segments, where each segment refers to one execution used in CIA.

For a given set of inputs, the impact set of each change is estimated using CIA. PerfImpact mines execution traces to obtain the performance differences of each method in the impact set and ranks the code changes based on their impacted methods' performance. The performance difference of a method is measured using the difference in its execution times between v_{i+1} and v_i. PerfImpact ranks the changes based on the sum of the differences in execution times of all methods in its impact set, which is shown in Alg. 2. Alg. 2 takes the changes C, the corresponding impact sets IM and method execution times (execution time for each method would exclude its callee's execution time) as inputs, and outputs a ranked list of changes RC. For each change c_k in C (line 1), it calculates the difference in execution time for each method in its impact set im_{c_k} (line 2). For example, the method m_q's difference in execution times (i.e., det_{m_q}) is equal to the method execution time in v_{i+1}, $mt^{i+1}_{m_q}$, minus the method execution time in v_i, $mt^{i}_{m_q}$ (line 3). If m_q is not invoked in v_i, $mt^{i}_{m_q}$ is assigned zero. $sdet_{c_k}$ is the sum of the differences in execution times of all methods in the impact set im_{c_k} (lines 4-6). Finally, each code change (e.g., c_k) is ranked based on its value $sdet_{c_k}$ and Alg. 2 terminates (lines 7-8). PerfImpact runs CIA on v_{i+1} to estimate impact sets of changes, hence the methods deleted in v_{i+1} are not included in the impact sets. As a result, the differences in execution times of these methods are not taken into account while evaluating the impact of changes on AUT's performance.

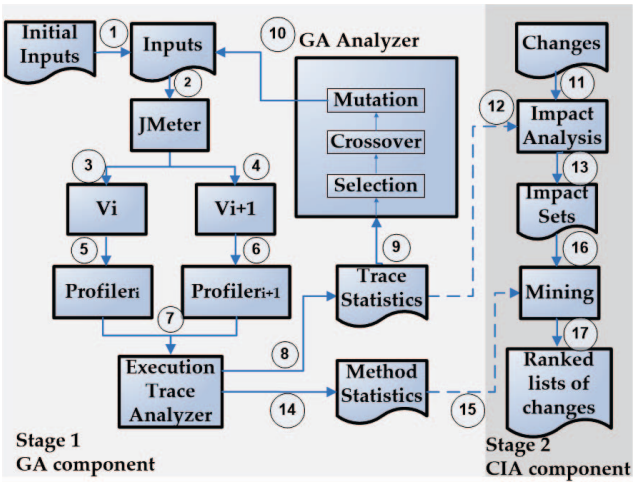


Figure 5: The workflow of PerfImpact.

3.4 Workflow of PerfImpact

The workflow of PerfImpact is shown in Fig. 5. Solid arrows indicate command and data flows between components, and the numbers in circles indicate the sequence of operations in the workflow. The dashed arrows denote transition in control flow once GA termination criteria is satisfied. Initially, sequences of inputs (i.e., individuals) are selected randomly for the first generation (1). While our paper starts this step (i.e., GA component) with random inputs, in practice, developers can also supply inputs that reveal performance bottlenecks in v_i (or any other inputs they would like to start with). JMeter [5] simulates users sending the inputs into two releases of the AUT automatically (2-4). *Profiler*$_i$ and *Profiler*$_{i+1}$ collect execution traces of each set of inputs on v_i and v_{i+1} respectively (5, 6). *Profilers* are implemented using Probekit [8], a lightweight profiling tool that injects the code fragments into specific points (e.g., method entry and exit) of the binary code for collecting the runtime data. *Execution Trace Analyzer* processes the execution traces (7) and extracts *Trace Statistics* (8) for *GA Analyzer* to evaluate each set of inputs (9). *GA analyzer* calculates the fitness value for each set of inputs according to Eq. 1 and selects the fitter ones to generate new inputs. The new inputs are sent back the AUT, starting the next iteration (10). GAs are implemented using JGAP [4].

After the GA component terminates, which means that PerfImpact finds the inputs likely to expose performance regressions, the second stage of PerfImpact (i.e., CIA component) is initiated with these inputs. By combining the *Change* information (e.g., full method names, signatures, return types) (11) and *Trace Statistics* (12), an *Impact Set* is derived for each change for the given inputs, using the *Impact Analysis* algorithm (13). *Method Statistics* are extracted to calculate the execution time in two releases for each method (14). In *Mining* phase, PerfImpact integrates *Method Statistics* (15) with *Impact Sets* (16), and uses the Alg. 2 to rank the changes for the given inputs (17). The changes ranked higher on the list are the ones likely leading to performance regressions. Note that the CIA component is initiated right after the GAs' search is terminated, since we expect mining execution traces for selected inputs to be useful to analyze the impact of each change on performance regressions. Alternatively, the CIA component can be also run simultaneously while running the GA component. This

usage of PerfImpact depends on two specific scenarios. In the first scenario, when stakeholders want to obtain the final ranked lists of changes, they can run the CIA component after GA component is terminated, as shown in Fig. 5. However, if stakeholders prefer to monitor the impact of inputs on performance changes, they can run the CIA component for the inputs that are selected at each generation (second scenario). To evaluate PerfImpact thoroughly, we choose the second scenario for our empirical study (section 4.3).

4. EVALUATION

In this section, we state our research questions (**RQs**) and explain how we conducted an empirical study aimed at evaluating our approach on two open-source applications.

4.1 Research Questions

RQ_1: How effective is PerfImpact in finding inputs that likely expose performance regressions in v_{i+1}?

RQ_2: Can PerfImpact effectively recommend changes between v_i and v_{i+1} likely responsible for performance regressions in v_{i+1} for a given set of inputs?

To answer $\mathbf{RQ_1}$, we introduced the following null (H_0) and alternative (H_1) hypotheses aimed at comparing inputs selected by PerfImpact with random inputs. Inputs with larger *time differences* (defined in Eq 1) are more likely to lead to performance regressions. The hypotheses are evaluated at a 0.05 level of significance:

H_0: There is no statistically significant difference in the *time differences* for the inputs generated by PerfImpact and random inputs.

H_1: There is a statistically significant difference in the *time differences* for the inputs generated by PerfImpact and random inputs.

To answer $\mathbf{RQ_2}$, after GA component is finished and changes are ranked, we run AUTs with the selected inputs to further understand the changes' impact on performance of two releases. We expect the changes ranked higher would lead to much longer execution time in v_{i+1} as compared to v_i.

4.2 Subject AUTs

We evaluated PerfImpact on two open-source web applications, JPetStore ($v_{3.0.0}$, $v_{4.0.5}$) and Agilefant ($v_{3.2}$, $v_{3.3}$, $v_{3.5}$). The statistics for all subjects are shown in Table 1. JPetStore [6] is a three-tier Java implementation of PetStore, which is widely used as performance benchmark [46, 47, 68, 27]. The GUI front end accepts users' URL requests, and the backend executes the requests and communicates with its database. Both JPetStore versions are deployed in Tomcat 6.0.35 and rely on Apache Derby 10.6.2.1 [2] as the backend database. Agilefant [1] is an open source application for managing agile software development, written in Java. All versions of Agilefant are deployed in Tomcat 7.0.47 with MySQL as the backend database.

4.3 Methodology

The *first goal* of the empirical study is to determine that whether the inputs selected by PerfImpact are likely to trigger performance regressions. To achieve this goal, we ran PerfImpact to obtain the inputs and compared them with randomly selected inputs. Random inputs are widely used in the testing field as they appear to be remarkably effective and reliable in test case generation [64, 38]. *Time difference* (see Eq. 1) was chosen to evaluate both the selected and random inputs. The inputs with larger *time differences* were more likely to trigger performance regressions.

Table 1: The stats of the subject programs.

Subjects	Version	#Methods	#Classes	Inputs(URLs)	
				Get	Post
JPetStore	$v_{3.0.0}$	307	52	115	5
JPetStore	$v_{4.0.5}$	407	43		
Agilefant	$v_{3.2}$	3,212	382	51	70
Agilefant	$v_{3.3}$	3,314	413		
Agilefnat	$v_{3.5}$	3,339	408		

The *second goal* of the empirical study is to demonstrate that PerfImpact can effectively mine execution traces for ranking the changes that lead to performance regressions on the top. This goal is twofold. First, we show the ranks of each change across generations in our GA implementation. With GA search converging, we expect the inputs to steer AUT executions to expose performance regressions. Thus, we conjecture that the ranks of some changes would stably converge to some high positions, identified as the ones highly likely to trigger regressions. Second, after ranking the changes, we show the changes' impacts on the performance of two releases with selected inputs (i.e., inputs selected in the last generation) to see whether the top ones really led to the expected performance regressions when increasing the workload. The impact of each change on AUT's performance was evaluated using its total execution time, which was equal to the sum of the execution time of all methods in its respective impact set. We expected the changes ranked higher on the list to have longer total execution times in v_{i+1}, yet shorter total execution times in v_i, which implies that changes with higher ranks impacted many methods that took longer time to execute in v_{i+1}. Especially when increasing the workload, the total execution times in v_{i+1} is expected to increase nonlinearly, implying that the performance may be degrading noticeably. We vary a number of users to simulate several realistic workloads.

We chose three pairs of AUT releases, JPetStore $v_{3.0.0}$ and $v_{4.0.5}$, Agilefant $v_{3.2}$ and $v_{3.3}$, and Agilefant $v_{3.2}$ and $v_{3.5}$, to evaluate PerfImpact . Two types of changes, *real* and *injected*, were involved. To extract the real changes, we computed diffs for each pair of releases [3]. Some changes were ignored since their inputs cannot be tested in our experiments (e.g., an input that triggers specific functionality that removes the same data from database and, hence, causes a database error). As a result, we extracted 68 changes between JPetStore $v_{3.0.0}$ and $v_{4.0.5}$, 24 changes between Agilefant $v_{3.2}$ and $v_{3.3}$, and 95 changes between Agilefant $v_{3.2}$ and $v_{3.5}$. Furthermore, we also wanted to determine how well PerfImpact is able to identify the known problematic changes. Thus, we also injected artificial changes in the second set of experiments. Injecting artificial changes to mimic the real performance regressions has been widely used in evaluating the effectiveness of performance regression testing techniques [41, 61, 67]. We randomly injected nine artificial changes (three for each group) into the source code of v_{i+1} (JPetStore $v_{4.0.5}$, Agilefant $v_{3.3}$ or Agilefant $v_{3.5}$). All these changes will lead to the synchronization problems similar in nature to one explained in the illustrative example (section 2.2), which would lead to longer latency during execution. The complete information on the injected changes is provided in our online appendix [7].

The inputs in our study were URLs, since we focused on web applications. One sequence of URLs sent by one user is defined as a *transaction*. Once URLs are selected randomly or by PerfImpact, JMeter simulates multiple users sending transactions into two releases of the AUT, and their

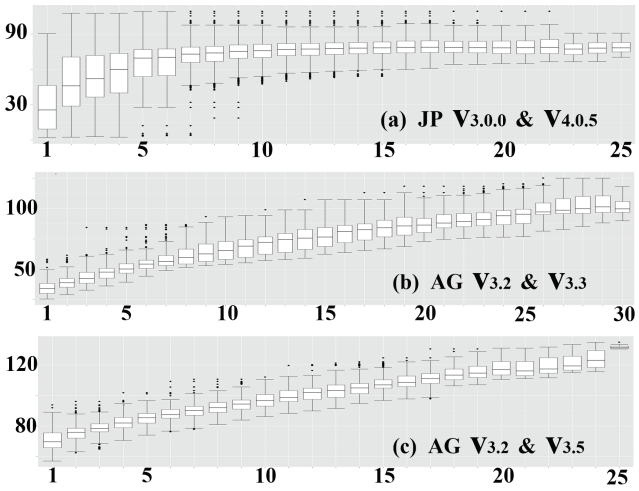


Figure 6: The box-and-whisker plots represent *time differences* between two released versions across generations on JPetStore (JP) and Agilefant (AG).

backends executing URL requests independently (see Fig. 5). Each transaction contained 50 URLs, and the number of users for the initial workload was set to five. Since PerfImpact selected random URLs to generate the initial population, it was necessary to conduct every experiment multiple times to avoid skewed results. Following the guidelines for using statistical tests to assess randomized algorithms [11, 10], we ran our experiments with the same configurations thirty times on JPetStore and ten times on Agilefant. That is, we ran JPetStore with random inputs thirty times and Agilefant with random inputs ten times. For each time, the number of combinations of inputs is equal to the number of individuals per generation. After identifying performance regression inducing changes, we also experiment with increased workloads (5, 10, 15, 20 and 25 users) to analyze these changes' impacts on performance regressions. The experiment with the same workload was run five times.

Our genetic algorithm was instantiated with a crossover rate of 0.3 and a mutation rate of 0.1. There were 30 individuals in each population, and the *time difference* was used as the fitness value. We set two criteria experimentally to terminate the GA cycle. First, if the increment of average *time difference* was less than or equal to 3% in ten successive generations, the GAs were terminated automatically. Second, we limited the number of generations to 30 - since each experiment is computationally expensive (e.g., Agilefant needs more than five days to finish one run on our hardware infrastructure).

The experiments on JPetStore were carried out using a Think Pad W530 laptop with Intel Core i7-3840QM processor 2.80 GHz, 32 GB DDR3 RAM. The experiments on Agilefant were carried out using two servers with 8 Intel Xeon Core E5-2609 CPU 2.40 GHz, 10 M Cache, 32 GB RAM.

5. EMPIRICAL RESULTS

This section analyzes the results of our empirical study. More experimental results are available online [7].

5.1 Finding Performance Regression Inputs

Fig. 6 shows the results of *time differences* between two releases across GA generations on JPetStore and Agilefant. The x-axis represents the generations, and the y-axis represents *time differences* between two releases (in seconds). The central box represents the values from the lower to upper quartile (i.e., 25 to 75 percentile). The middle line represents the median. The vertical line extends from the minimum to the maximum value. Note that, if a set of inputs leads to larger *time difference*, this set is likely to trigger performance regressions. As shown in Fig. 6, the *time difference* increases as the GAs progress, implying that PerfImpact steered execution of the AUTs to the paths which triggered performance regressions. Specifically, Table 2 compares the *time differences* of selected inputs in the last generation with the random inputs in the first generation. The average *time differences* for the selected inputs are significantly larger than the time differences for the random inputs ($162.35\% - 288.72\%$ increase), which clearly demonstrates that the inputs selected by PerfImpact were more likely to trigger performance regressions. The values of the standard deviation (SD) of the selected inputs are much smaller as compared to the random inputs for JPetStore. We suggest that the selected inputs converge to a stable subset of inputs. However, the values of SD of the selected inputs are larger as compared to the random inputs in Agilefant. Recall that Agilefant has relatively more sophisticated architecture than JPetStore. Thus, PerfImpact has more chances to steer the executions to different paths, leading to larger values of SD. Additionally, a paired t-test with one-tailed distribution was performed to compare the *time differences* of random inputs and selected inputs. The $p-value$ of these three groups are significantly smaller than 0.05. Based on these results we reject the null hypothesis. *These results demonstrate that* PerfImpact *can find the combinations of inputs that were significantly more effective as compared to random inputs in exposing these performance regressions.*

Table 2: The *time difference* between two versions for random inputs (Rd) and PerfImpact selected inputs (PI) in JPetStore (JP) and Agilefant (AF).

App	Inputs	MIN	MAX	AVG	SD	P-value
$JP_{3.3.0\&4.0.5}$	Rand	2.13	90.39	32.17	23.77	<1.23E-296
	PI	66.47	109.22	79.82	6.28	
$AF_{3.2\&3.3}$	Rand	25.50	58.22	34.75	6.30	1.37E-236
	PI	76.84	125.03	100.33	11.19	
$AF_{3.2\&3.5}$	Rand	57.07	93.66	70.54	6.70	2.64E-198
	PI	96.12	134.84	114.52	10.84	

5.2 Identifying Code Changes

To evaluate PerfImpact's effectiveness in identifying problematic code changes, we provide the rankings of six randomly chosen code changes from Agilefant as examples, including five real and one injected change. The detailed information on the changes is shown in Table 3. Due to lack of space, the experimental results for other changes can be found in the online appendix [7]. Fig. 7 shows the ranks of these changes across generations. The central box represents the values from the lower to upper quartile (i.e., 25 to 75 percentile). The middle line represents the median. The vertical line extends from the minimum to the maximum value. The blue lines are the fitting lines generated using generalized linear model. For Agilefant, there are 27 changes (i.e., 24 real and three injected changes) between $v_{3.2}$ and $v_{3.3}$, and 98 changes (i.e., 95 real and three injected changes) between $v_{3.2}$ and $v_{3.5}$, thus the range of ranks in $v_{3.3}$ was from 1 to 27 and the range of ranks in $v_{3.5}$ was from 1 to 98. Note that, the methods with smaller values (close to one) for ranks are ranked higher. Fig. 7 shows that the ranks for changes vary in the first generation, since the inputs are generated randomly. As the GAs progress, the executions are steered

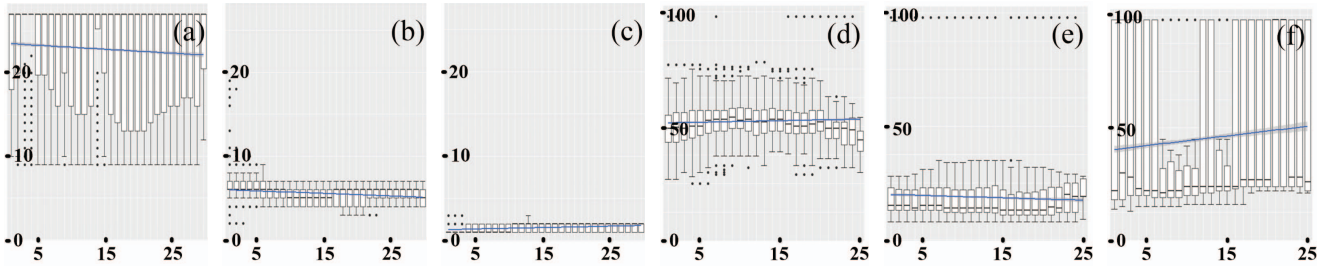


Figure 7: The box-and-whisker plots represent the ranks of the changes in Table 3. The x-axis represents the generations, and the y-axis represents the ranks. Smaller values that appear on y-axis imply higher ranks.

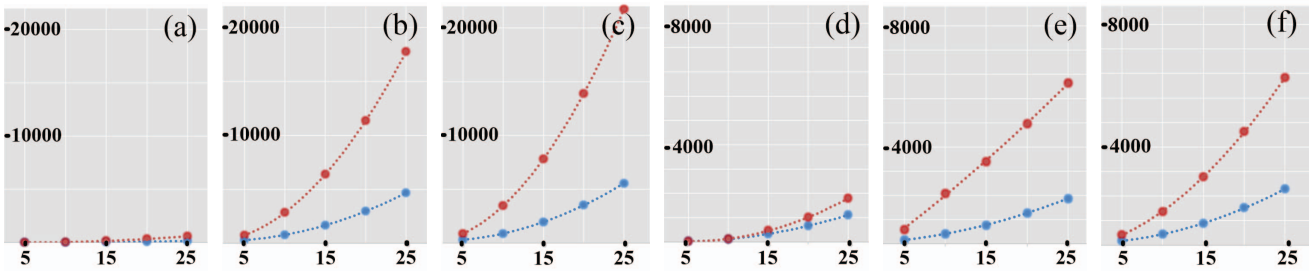


Figure 8: The figures show the average of total execution times of the changes in Table 3. This total execution time of one change is the total execution time of all methods in its respective impact set. The blue dots show the average of total execution time in old version of Agilefant ($v_{3.2}$), and the red dots show the average of total execution time in new version of Agilefant ($v_{3.3}$ or $v_{3.5}$). The curves are the fitting curves generated using Polynomial Function model. The inputs were selected in the last generation. The x-axis represents the average of total execution time, and the y-axis represents the number of users. Time is measured in seconds.

to the paths where the performance regressions are exposed, thus the ranks of some changes (e.g., change (b), (c), (d) and (e)) become more stable and converge to the final ranks.

Based on the stable ranks in the last generation, we can easily identify two types of changes. One change type that has relatively higher ranks (i.e., smaller values on y-axis in Fig. 7), such as changes (b), (c), and (e), is identified as representing problematic changes. Specially, change (c) is an injected change. We also checked the ranks of other injected changes. All of them were ranked on the top, demonstrating that PerfImpact can effectively identify the injected changes. The other change type that has noticeably lower ranks (i.e., larger values on y-axis in Fig. 7), such as change (d), is identified as the one less likely to trigger performance regressions. Unlike the changes that have stable ranks in the last generation, change (a) and (f) vary significantly. We further analyzed their ranks to understand the reason behind these variations. Change (f) had relatively higher median ranks (middle lines in boxplots), implying that it may trigger performance regressions for some specific inputs. We will discuss its source code later to show more details. However, the median ranks of change (a) were close to the bottom (i.e, rank 27 in $v_{3.3}$), implying that it was not invoked for most of the selected inputs and it had less contribution to performance regressions. PerfImpact tended to discard the inputs less likely to trigger performance regressions as the GAs progressed, thus the corresponding methods were not invoked. In conclusion, based on the ranks in the last generation, we can identify different types of changes.

To demonstrate that the changes with higher ranks were likely to trigger performance regressions, we ran the selected inputs on AUTs with different workloads (i.e, different numbers of users) and obtained the average total execution times for each change in two releases. In general, one change with longer total execution times in v_{i+1} is more likely to trigger performance degradation. As the results show in Fig. 8, the changes with higher ranks (e.g., changes (b), (c), (e) and (f)) have much larger averages of the total execution times in v_{i+1} (i.e., red lines in Fig. 8) as compared to the ones in v_i (i.e., blue lines in Fig. 8). We used polynomial functions to fit the results, demonstrating that the average of the total execution times increased nonlinearly when the workload increased. The polynomial functions for all examples in Table 3 are shown in our online appendix [7]. Conversely, the changes with lower ranks (e.g., changes (a) and (d)) have relatively shorter average total execution times in both v_i and v_{i+1}. Recall that change (a) was not invoked by most of selected inputs. Its averages of total execution times in $v_{3.2}$ and $v_{3.3}$ were close to zero. As expected, the changes with higher ranks led to longer execution times in v_{i+1}, and the times increased nonlinearly given an increase in the workload.

Table 3: Examples of code changes in Agilefant.

	Method Name	Versions
a	fi.hut.soberit.agilefant.business.impl.SearchBusinessImpl.taskListSearchResult	$v_{3.2}$ vs $v_{3.3}$
b	fi.hut.soberit.agilefant.business.impl.SettingBusinessImpl.retrieveByName	$v_{3.2}$ vs $v_{3.3}$
c	injected code change	$v_{3.2}$ vs $v_{3.3}$
d	fi.hut.soberit.agilefant.business.impl.StoryHierarchyBusinessImpl.calculateStoryTreeMetrics	$v_{3.2}$ vs $v_{3.5}$
e	fi.hut.soberit.agilefant.business.impl.ProjectBusinessImpl.retrieveLeafStories	$v_{3.2}$ vs $v_{3.5}$
f	fi.hut.soberit.agilefant.web.TimesheetAction.generateTree	$v_{3.2}$ vs $v_{3.5}$

To further demonstrate that PerfImpact identified the problematic changes effectively, we looked into the source code of each change. Fig. 9 shows two examples of such code changes. More examples are available in the online appendix [7]. Fig. 9 (a) shows the source code of change (f) in Table 3, which was ranked highly for some selected inputs. As expected, PerfImpact found the inputs that satisfied

Figure 9: Examples of code changes in Agilefant. (a) shows the source code of change (f) in Table 3, and (b) shows the source code of change (d) in Table 3.

the *if* clauses, which led to different performance in two releases. In $v_{3.2}$, the method was returned directly with a *Action.ERROR*. Instead, in $v_{3.5}$, it called *storeaAllTimeSheets* to obtain a collection of *Products*, and added products' IDs into *selectdBacklogIds*. Then, the execution went through the following steps in change (f). Apparently, change (f) required more time to execute in $v_{3.5}$, especially when the size of the *products* increased, leading to a performance regression. Note that the inputs that did not satisfy the *if* clause would not lead to performance degradation. This example demonstrates that PerfImpact can find specific inputs that trigger the performance regressions and effectively locate the problematic changes. Fig. 9 (b) shows the source code of change (d) in Table 3, which got relatively lower ranks in PerfImpact. The change was that, in the *for* loop, the current iteration would be skipped in $v_{3.5}$, when *story.getId* was equal to *child.getId*. Apparently, change (d) would not degrade the performance in $v_{3.5}$, thus it was correctly ranked lower by PerfImpact. *These results show that* PerfImpact *can be used to effectively identify the changes that are responsible for performance regressions.*

6. LIMITATIONS

First, our current implementation of PerfImpact only focuses on the identical input values that are valid for both releases, v_i and v_{i+1}. The differences in inputs between two releases, such as the new inputs in v_{i+1} that may no longer be valid in v_i, were not tested, since they cannot be sent into both of two releases for performance comparison. Moreover, when generating new inputs, some constraints (e.g., the order of URLs in a chromosome) must be considered to guarantee that the new inputs are valid. However, our current implementation deals with some straightforward constraints, such as a login with a predefined username and the password at the beginning. Testing different inputs between two releases and considering other constraints are currently out of the scope of this paper and we leave them for future work.

Second, PerfImpact does not analyze root causes behind detected performance regressions and does not take into account potential interactions among performance regressions [24, 50]. Multiple inputs and changes may be responsible for one or many performance regressions, thus, our approach may not necessarily be able to capture cases where the behaviors of performance regressions are changing due to interactions among those regressions (e.g., a situation where one performance regression obscures effects of another regression for certain inputs). Also, if an AUT is multithreaded, even if it runs twice with the same input, the execution time may be different due to multithreaded interleavings.

Third, in our empirical study, we only applied PerfImpact to several releases of two open-source web applications. It is hard to generalize the results given that our experiments are based on the two applications (even though we considered five releases of these two apps in total). However, JPetStore has been widely used as a benchmark in performance testing [46, 47, 68, 27] and Agilefant is an enterprise-level real-world application. Thus, we believe that these applications are representative real-world software systems. Also, another potential threat is that we only considered one type of inputs (i.e., URL requests), since we experimented with web-based applications. However, PerfImpact can be used with other types of applications and inputs (the chromosomes can be reformatted to accommodate other types of inputs). We leave this extension for future work.

Finally, we only injected one type of artificial changes to simulate performance regressions. Also we had to discard some real changes since they can not be covered by PerfImpact. However, we extracted 187 different real changes in the subject applications. Thus, we believe that all the changes (real and injected) used in evaluation constitute a solid experimental design to support our current conclusions. Furthermore, PerfImpact only focuses on method-level changes in the native source code. Currently, PerfImpact does not take into account different granularity and possible changes in the underlying third-party or standard libraries. While analyzing the impact of changes in underlying libraries on the performance of a client application is an important problem [37], we leave it for the future work.

7. RELATED WORK

Genetic algorithms. Genetic algorithms are widely used in different areas of software engineering [14, 35, 70], and software testing in particular [29, 44, 33, 34, 12, 56, 23]. In software testing, many approaches rely on GAs for test case generation. Fraser *et al.* proposed EvoSuite, that uses GAs to optimize whole test suites to smaller subsets which satisfy certain coverage criteria [28, 30]. 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 [30]. An approach proposed by Gross *et al.* introduced a test case generation technique that employs GAs to systematically generate test cases at

GUI level while learning the code behavior for achieving high coverage and avoiding false failures in unit testing [36]. Test suite augmentation techniques are used to generate test cases that cover code changes or code elements affected by changes [72, 71]. These approaches focus on finding new test cases to achieve higher code coverage, whereas PERFIMPACT focuses on using GAs to identify the inputs exposing performance regressions during profiling process.

In our own recent work, we proposed GA-Prof, which uses GAs to search for input values leading to performance bottlenecks in a given software release (e.g. v_{i+1})[69]. In contrast, PERFIMPACT uses GAs to find the inputs that reveal performance regressions between two AUT releases (e.g. v_i and v_{i+1}) and is designed to work in the context of software evolution to support performance regression testing. A performance bottleneck (in v_{i+1}) detected by GA-Prof is not necessarily a performance regression. Since this bottleneck may already exist in v_i, no performance degradation is involved between two releases. PERFIMPACT is able to further help developers to ignore this type of performance problems, and focus on the methods with larger differences in performance between two releases. Additionally, the goals of these two works are quite different. GA-Prof identifies the bottlenecks that have significant contributions to longer execution time, but PERFIMPACT uses CIA to analyze the impact of code changes on the problematic methods for identifying the ones that are responsible for actual performance regressions.

Change Impact Analysis is a technique aimed at helping developers to understand the effects of a change on the rest of the source code [55, 53]. Many CIA approaches have been proposed [31, 13, 54, 25, 16]. Law and Rothermel proposed a dynamic path-based impact analysis, which assumes that a change has a potential impact on the code reachable from this change [51]. Following this approach, Apiwattanapong *et al.* presented a method that only considers essential dynamic information by using execute-after sequences [9]. Ren *et al.* presented a tool, Chianti, to identify the changes that induce the failure of one specific test [65]. Zhang *et al.* introduced FaultTracer, which adapts spectrum-based fault localization techniques with a CIA-based algorithm to rank the changes for identifying failure-inducing ones [78, 79, 80]. However, these approaches do not focus on performance regressions. To the best of our knowledge, PERFIMPACT is the first technique to combine CIA with search-based input profiling to analyze the impact of changes on an AUT's performance.

Regression Testing. The default approach for regression testing is to retest all test cases after releasing a new version, which is an expensive proposition. To solve this problem, a number of techniques for selecting regression tests have been proposed [26, 49, 21, 66, 32, 74]. 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 [22]. Yu *et al.* provided a new approach, namely SimRT, which identifies variables shared by multiple threads and employs a test selection technique to select the test cases that exercise these shared variables, detecting data races [75]. These techniques prioritize functional test cases and may not be directly applicable in the context of performance testing.

Performance faults have been found to be more difficult to fix as compared to non-performance faults [77, 76, 63], hence, several approaches have been proposed to support performance regression testing see Table 4. There are *three* major differences between these approaches (see Table 4). *First*, some approaches rely on profiling of the AUT and some do not. Profiling is a well-established and useful technique for analyzing the AUT's behaviors, and is widely used in performance testing field [52, 57]. PERFIMPACT uses differential profiling to run the same inputs in two software versions simultaneously, which enables accurate detections of performance regressions. *Second*, some approaches mine information from repositories to identify performance regressions [27]. However, many software systems may not necessarily maintain well-structured repositories. PERFIMPACT detects performance regressions without relying on the testing history, which makes it applicable to other contexts including testing legacy systems. *Third*, performance regression testing is not completed until the code changes responsible for performance regressions are identified. Yet, only a very few approaches address this concern. For instance, Huang *et al.* detect high-risk commits that may lead to performance regressions using static analysis [43]. However, this work relies on static analysis and focuses on specific types of performance regressions. A recent work analyzes root causes behind performance regressions, yet it requires the AUT to maintain an accurate set of unit tests [41]. On the contrary, PERFIMPACT does not require unit tests and relies on dynamic information to automatically and effectively identify actual bottlenecks (that can be observed and confirmed at run-time) as well as problematic changes.

Table 4: Performance regression testing approaches.

Approaches	Analysis		Profiling	Repository	Identify Changes
	Static	Dynamic			
Our approach	·	●	●	·	●
Shang *et al.* [67]		●	·	·	·
Huang *et al.* [43]	●	·	·	·	●
Nguyen *et al.* [62]	·	●	·	●	●
Heger *et al.* [41]	·	●	●	·	●
Lee *et al.* [52]	·	●	●	·	●
Nguyen *et al.* [61]	·	●	·	·	·
Foo *et al.* [27]	·	●	·	●	·
Mostafa *et al.* [59]	·	●	●	●	●
Mi *et al.* [57]	·	●	●	·	·
Chen *et al.* [20]	·	●	·	·	·
Kalibera *et al.* [48]	·	●	·	·	·
Bulej *et al.* [17]	·	●	·	·	·
Yilmaz *et al.* [73]	·	●	·	·	·

8. CONCLUSION

We propose a novel recommendation system, PERFIMPACT, aimed at automatically recommending code changes likely responsible for performance regressions. Our approach uses search-based input profiling to detect input combinations likely leading to performance regressions, and mines execution traces to estimate the impact of code changes on detected performance regressions. We implemented PERFIMPACT and tested it on different releases of two open-source web applications. The results demonstrate that PERFIMPACT can effectively select the inputs exposing performance regressions. Also, the ranked lists of changes computed with PERFIMPACT are useful for stakeholders to identify potential changes behind performance regressions for further inspection and root cause analysis.

Acknowledgments

This work is supported in part by the NSF CCF-1217928, CCF-1218129, IIP-1547597 grants and Microsoft SEIF. Any opinions, findings, and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

9. REFERENCES

[1] Agilefant, http://agilefant.com/.

[2] Apache derby, http://db.apache.org/derby/.

[3] Beyond compare, http://www.scootersoftware.com/.

[4] Jgap, http://jgap.sourceforge.net/.

[5] Jmeter, http://jmeter.apache.org/.

[6] Jpetstore, http://sourceforge.net/projects/ibatisjpetstore.

[7] Perfimpact, http://www.cs.wm.edu/semeru/data/MSR16-PerfImpact/.

[8] Probekit, http://www.eclipse.org/tptp/platform/documents/probekit/probekit.html.

[9] T. Apiwattanapong, A. Orso, and M. J. Harrold. Efficient and precise dynamic impact analysis using execute-after sequences. In *ICSE '05*, pages 432–441.

[10] A. Arcuri and L. Briand. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *STVR '14*, 24:219–250.

[11] A. Arcuri and L. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *ICSE '11*, pages 1–10.

[12] A. Baars, M. Harman, Y. Hassoun, K. Lakhotia, P. McMinn, P. Tonella, and T. Vos. Symbolic search-based testing. In *ASE '11*, pages 53–62.

[13] L. Badri, M. Badri, and D. St-Yves. Supporting predictive change impact analysis: A control call graph based technique. In *APSEC '05*, pages 167–175.

[14] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro. The plastic surgery hypothesis. In *FSE '14*, pages 306–317.

[15] B. Boehm. Software engineering economics. *TSE*, SE-10(1):4–21, 1984.

[16] J. Branchaud, S. Person, and N. Rungta. A change impact analysis to characterize evolving program behaviors. In *ICSM '12*, pages 109–118.

[17] L. Bulej, T. Kalibera, and P. Tma. Repeated results analysis for middleware regression benchmarking. *Perform. Eval.*, 60(1-4):345–358, 2005.

[18] A. Capiluppi, J. Fernandez-Ramil, J. Higman, H. C. Sharp, and N. Smith. An empirical study of the evolution of an agile-developed software system. In *ICSE '07*, pages 511–518.

[19] N. Chapin, J. E. Hale, K. M. Kham, J. F. Ramil, and W.-G. Tan. Types of software evolution and software maintenance. *Journal of Software Maintenance*, 13(1):3–30, 2001.

[20] T. Chen, L. I. Ananiev, and A. V. Tikhonov. Keeping kernel performance from regressions. In *Linux Symposium*, volume 1, pages 93–102, 2007.

[21] Y.-F. Chen, D. S. Rosenblum, and K.-P. Vo. Testtube: A system for selective regression testing. In *ICSE '94*, pages 211–220.

[22] C. Del Grosso, G. Antoniol, and M. Di Penta. An evolutionary testing approach to detect buffer overflow. In *ISSRE '04*.

[23] M. Di Penta, G. Canfora, G. Esposito, V. Mazza, and M. Bruno. Search-based testing of service level agreements. In *GECCO '07*, pages 1090–1097.

[24] N. DiGiuseppe and J. A. Jones. Fault interaction and its repercussions. In *ICSM '11*, pages 3–12.

[25] B. Dit, M. Wagner, S. Wen, W. Wang, M. Linares-Vásquez, D. Poshyvanyk, and H. Kagdi. Impactminer: A tool for change impact analysis. In *ICSE '14*, pages 540–543.

[26] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Prioritizing test cases for regression testing. In *ISSTA '00*, pages 102–112.

[27] K. Foo, Z. M. Jiang, B. Adams, A. Hassan, Y. Zou, and P. Flora. Mining performance regression testing repositories for automated performance analysis. In *QSIC '10*, pages 32–41.

[28] G. Fraser and A. Arcuri. Evolutionary generation of whole test suites. In *QSIC '11*, pages 31–40.

[29] G. Fraser and A. Arcuri. Achieving scalable mutation-based generation of whole test suites. *EMSE*, pages 1–30, 2014.

[30] G. Fraser, A. Arcuri, and P. McMinn. Test suite generation with memetic algorithms. In *GECCO '13*, pages 1437–1444.

[31] M. Gethers, B. Dit, H. Kagdi, and D. Poshyvanyk. Integrated impact analysis for managing software changes. In *ICSE '12*, pages 430–440.

[32] M. Gligoric, R. Majumdar, R. Sharma, L. Eloussi, and D. Marinov. Regression test selection for distributed software histories. In *CAV '14*, pages 293–309.

[33] P. Godefroid and S. Khurshid. Exploring very large state spaces using genetic algorithms. *Int. J. Softw. Tools Technol. Transf.*, 6(2), 2004.

[34] A. Goffi, A. Gorla, A. Mattavelli, M. Pezzè, and P. Tonella. Search-based synthesis of equivalent method sequences. In *FSE '14*, pages 366–376, 2014.

[35] C. L. Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *TSE*, 38:54–72, 2012.

[36] F. Gross, G. Fraser, and A. Zeller. Search-based system testing: High coverage, no false alarms. In *ISSTA '12*, pages 67–77.

[37] J. Gui, S. Mcilroy, M. Nagappan, and W. G. Halfond. Truth in advertising: The hidden cost of mobile ads for software developers. 2015.

[38] R. Hamlet. Random testing. In *Encyclopedia of Software Engineering*, pages 970–978, 1994.

[39] M. Harman, Y. Jia, and W. B. Langdon. Strong higher order mutation-based test data generation. In *FSE '11*, pages 212–222.

[40] M. Harman and P. McMinn. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *TSE '10*, 36(2):226–247.

[41] C. Heger, J. Happe, and R. Farahbod. Automated root cause isolation of performance regressions during software development. In *ICPE '13*, pages 27–38.

[42] J. H. Holland. *Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence.* 1975.

[43] P. Huang, X. Ma, D. Shen, and Y. Zhou. Performance regression testing target prioritization via performance risk analysis. In *ICSE 2014*, pages 60–71.

[44] S. Huang, M. B. Cohen, and A. M. Memon. Repairing gui test suites using a genetic algorithm. In *ICST '10*, pages 245–254.

[45] M. Z. Iqbal, A. Arcuri, and L. Briand. Empirical investigation of search algorithms for environment

model-based testing of real-time embedded software. In *ISSTA '12*, pages 199–209.

[46] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora. Automated performance analysis of load tests. In *ICSM '09*, pages 125–134.

[47] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora. Automatic identification of load testing problems. In *ICSM '08*, pages 307–316.

[48] T. Kalibera, L. Bulej, and P. Tuma. Automated detection of performance regressions: the mono experience. In *MASCOTS '05*, pages 183–190.

[49] J.-M. Kim and A. Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *ICSE '02*, pages 119–129.

[50] J.-M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test application frequency. In *ICSE '00*, pages 126–135.

[51] J. Law and G. Rothermel. Whole program path-based dynamic impact analysis. In *ICSE '03*, pages 308–318.

[52] D. Lee, S. K. Cha, and A. H. Lee. A performance anomaly detection and analysis framework for dbms development. *IEEE Trans. on Knowl. and Data Eng.*, 24(8), 2012.

[53] S. Lehnert. A taxonomy for software change impact analysis. In *IWPSE-EVOL '11*, pages 41–50.

[54] B. Li, X. Sun, and H. Leung. Combining concept lattice with call graph for impact analysis. *Adv. Eng. Softw.*, 53:1–13, 2012.

[55] B. Li, X. Sun, H. Leung, and S. Zhang. A survey of code-based change impact analysis techniques. *Softw. Test., Verif. Reliab.*, 23(8):613–646, 2013.

[56] Z. Li, M. Harman, and R. M. Hierons. Search algorithms for regression test case prioritization. *IEEE Trans. Softw. Eng.*, 33(4), 2007.

[57] N. Mi, L. Cherkasova, K. Ozonat, J. Symons, and E. Smirni. Analysis of application performance and its change via representative application signatures. In *NOMS '08*, pages 216–223.

[58] M. Mitchell. *An Introduction to Genetic Algorithms*. 1998.

[59] N. Mostafa and C. Krintz. Tracking performance across software revisions. In *PPPJ '09*, pages 162–171.

[60] T. Nguyen, B. Adams, Z. M. Jiang, A. Hassan, M. Nasser, and P. Flora. Automated verification of load tests using control charts. In *APSEC '11*, pages 282–289.

[61] 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 *ICPE '12*, pages 299–310.

[62] 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 *MSR '14*, pages 232–241.

[63] A. Nistor, T. Jiang, and L. Tan. Discovering, reporting, and fixing performance bugs. In *MSR '13*, pages 237–246.

[64] S. Park, B. M. M. Hossain, I. Hussain, C. Csallner, M. Grechanik, K. Taneja, C. Fu, and Q. Xie. Carfast: Achieving higher statement coverage faster. In *FSE '12*, pages 35:1–35:11.

[65] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: A tool for change impact analysis of java programs. In *OOPSLA '04*, pages 432–448.

[66] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Trans. Softw. Eng. Methodol.*, 6(2):173–210, 1997.

[67] W. Shang, A. E. Hassan, M. Nasser, and P. Flora. Automated detection of performance regression using regression models on clustered performance counters. In *ICPE '15*.

[68] A. Shankar, M. Arnold, and R. Bodik. Jolt: Lightweight dynamic analysis and removal of object churn. In *OOPSLA '08*, pages 127–142.

[69] D. Shen, Q. Luo, D. Poshyvanyk, and M. Grechanik. Automating performance bottleneck detection using search-based application profiling. In *ISSTA'15*.

[70] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *ICSE '09*, pages 364–374.

[71] Z. Xu, M. B. Cohen, and G. Rothermel. Factors affecting the use of genetic algorithms in test suite augmentation. In *GECCO '10*, pages 1365–1372.

[72] Z. Xu, Y. Kim, M. Kim, and G. Rothermel. A hybrid directed test suite augmentation technique. In *ISSRE '11*, pages 150–159.

[73] C. Yilmaz, A. S. Krishna, A. Memon, A. Porter, D. C. Schmidt, A. Gokhale, and B. Natarajan. Main effects screening: A distributed continuous quality assurance process for monitoring performance degradation in evolving software systems. In *ICSE '05*, pages 293–302.

[74] T. Yu, X. Qu, M. Acharya, and G. Rothermel. Oracle-based regression test selection. In *ICST '13*, pages 292–301.

[75] T. Yu, W. Srisa-an, and G. Rothermel. Simrt: An automated framework to support regression testing for data races. In *ICSE '14*, pages 48–59.

[76] S. Zaman. Empirical studies of performance bugs and performance analysis approaches for software systems. In *Master thesis*, 2012.

[77] S. Zaman, B. Adams, and A. E. Hassan. A qualitative study on performance bugs. In *MSR '12*, pages 199–208.

[78] L. Zhang, M. Kim, and S. Khurshid. Localizing failure-inducing program edits based on spectrum information. In *ICSM '11*, pages 23–32.

[79] L. Zhang, M. Kim, and S. Khurshid. Faulttracer: A change impact and regression fault analysis tool for evolving java programs. In *FSE*, page 40, 2012.

[80] L. Zhang, M. Kim, and S. Khurshid. Faulttracer: a spectrum-based approach to localizing failure-inducing program edits. *JSEP*, 25(12):1357–1383, 2013.