Empir Software Eng (2017) 22:6–56
DOI 10.1007/s10664-015-9413-5

FOREPOST: finding performance problems automatically with feedback-directed learning software testing

Qi Luo[1] · Aswathy Nair[2] · Mark Grechanik[3] · Denys Poshyvanyk[1]

Published online: 11 December 2015

Abstract A goal of performance testing is to find situations when applications unexpectedly exhibit worsened characteristics for certain combinations of input values. A fundamental question of performance testing is how to select a manageable subset of the input data faster in order to automatically find performance bottlenecks in applications. We propose FOREPOST, a novel solution, for automatically finding performance bottlenecks in applications using black-box software testing. Our solution is an adaptive, feedback-directed learning testing system that learns rules from execution traces of applications. Theses rules are then used to automatically select test input data for performance testing. We hypothesize that FOREPOST can find more performance bottlenecks as compared to random testing. We have implemented our solution and applied it to a medium-size industrial application at a major insurance company and to two open-source applications. Performance bottlenecks were found automatically and confirmed by experienced testers and developers. We also thoroughly studied the factors (or independent variables) that impact the results of FOREPOST.

Communicated by: Ahmed E. Hassan

✉ Qi Luo
qluo@cs.wm.edu

Aswathy Nair
nair.a.87@gmail.com

Mark Grechanik
drmark@uic.edu

Denys Poshyvanyk
denys@cs.wm.edu

[1] Department of Computer Science, College of William and Mary, Williamsburg, VA 23188, USA

[2] Bank of American Merrill Lynch, Pennington, NJ 08534, USA

[3] Department of Computer Science, University of Illinois at Chicago, Chicago, IL 60601, USA

Keywords Performance testing · Feedback-directed learning system

1 Introduction

The goal of performance testing is to find performance bottlenecks, when an *application under test (AUT)* unexpectedly exhibits worsened characteristics for a specific workload (Molyneaux 2009; Weyuker and Vokolos 2000). One way to find performance bottlenecks effectively is to identify test cases for finding situations where an AUT suffers from unexpectedly high response time or low throughput (Jiang et al. 2009; Avritzer and Weyuker 1994). Test engineers construct performance test cases, and these cases include actions (e.g., interacting with GUI objects or invoking methods of exposed interfaces) as well as input test data for the parameters of these methods or GUI objects (IEEE 1991). It is difficult to construct effective test cases that can find performance bottlenecks in a short period of time, since it requires test engineers to test many combinations of actions and input data for nontrivial applications.

Developers and testers need performance management tools for identifying performance bottlenecks automatically in order to achieve better performance of software while keeping the cost of software maintenance low. In a survey of 148 enterprises, 92 % said that improving application performance was a top priority (Yuhanna 2009; Schwaber et al. 2006). In a recent work, Zaman et al. performed a qualitative study that demonstrated that performance bottlenecks are not easy to reproduce and that developers spend more time working on them (Zaman et al. 2012). Moreover, Nistor et al. found that fixing performance bottlenecks is difficult and better tools for locating and fixing performance bottlenecks are needed by developers (Nistor et al. 2013, 2015). As a result, different companies work on tools to alleviate performance bottlenecks. The application performance management market is over 2.3 billion USD and growing at 12 % annually, making it one of the fastest growing segments of the application services market (Ashley 2006; Garbani 2008). Existing performance management tools collect and structure information about executions of applications, so that stakeholders can analyze this information to obtain insight into performance. Unfortunately, none of these tools identifies performance bottlenecks automatically. The difficulty of comprehending the source code of large-scale applications and their high complexity lead to performance bottlenecks that result in productivity loss approaching 20 % for different domains due to application downtime (Group 2005).

Considering that source code may not even be available for some components, engineers concentrate on black-box performance testing of the whole application, rather than focusing on standalone components (Aguilera et al. 2003; Isaacs and Barham 2002). Depending on input values, an application can exhibit different behaviors with respect to resource consumption. Some of these behaviors involve intensive computations that are characteristic of performance bottlenecks (Zhang et al. 2011). Naturally, testers want to summarize the behavior of an AUT concisely in terms of its inputs. In this way, they can select input data that will lead to significantly increased resource consumption, thereby revealing performance bottlenecks. Unfortunately, finding proper rules that collectively describe properties of such input data is a highly creative process that involves deep understanding of input domains (Ammann and Offutt 2008, page 152).

Descriptive rules for selecting test input data play a significant role in software testing (Beck 2003), because these rules approximate the functionality of an AUT. For example, a rule for an insurance application is that some customers will pose a high insurance risk if these customers have one or more prior insurance fraud convictions and deadbolt locks are not installed on their premises. Computing an insurance premium may consume more resources for a customer with a high-risk insurance record that matches this rule versus a customer with an impeccable record. The reason is that processing this high-risk customer record involves executing multiple computationally expensive transactions against a database. Of course, we use this example of an oversimplified rule to illustrate the idea. However, even though real-world systems exhibit much more complex behavior, useful descriptive rules often enable testers to build effective performance bottleneck revealing test cases.

We offer a novel solution for *Feedback-ORiEnted PerfOrmance Software Testing (FOREPOST)* by finding performance bottlenecks automatically through learning and using rules that describe classes of input data that lead to intensive computations, which is summarized in our previous paper (Grechanik et al. 2012). FOREPOST is an adaptive, feedback-directed learning testing system that learns rules from an AUT's execution traces. These rules are used to automatically select test input data for performance testing. As compared to random testing, FOREPOST can find more performance bottlenecks in applications. FOREPOST uses runtime monitoring for a short duration of testing together with machine learning techniques and automated test scripts. It reduces large amounts of performance-related information collected during AUT runs to a small number of descriptive rules. These rules provide insights into properties of test input data that lead to increased computational loads in applications.

As compared to our previous conference paper (Grechanik et al. 2012), this journal paper introduces an integrated testing approach, which considers both the random input data and the specific inputs based on generated descriptive rules. We thoroughly evaluate this new combined approach. Moreover, besides the commercial renters insurance application and iBatis JPetStore, we also involve a new open source application, Dell DVD Store, in our empirical evaluation. To understand the impact of different parameters, we consider more independent variables, such as the number of profiles and the number of iterations, in the experiments. We also present the study evaluating accuracy of FOREPOST by detecting a priori injected and annotated bottlenecks.

This paper makes the following contributions:

– FOREPOST, a novel approach that collects and utilizes execution traces of the AUT to learn rules that describe the computational intensity of the workload in terms of the properties of the input data. These rules are used by an adaptive automated test script automatically, in a feedback loop, to steer the execution of the AUT by selecting input data based on the newly learned rules. We are not aware of any existing testing approaches that use similar ideas to automatically find performance bottlenecks in real-world applications.
– We provide a novel algorithm that identifies methods that lead to performance *bottlenecks*, which are phenomena where the performance of the AUT is limited by one or few components (Aguilera et al. 2003; Ammons et al. 2004).
– We have implemented FOREPOST and applied it to a real-word application currently deployed in a major insurance company. Performance bottlenecks were found automatically in the insurance application and were confirmed by experienced testers and developers. After implementing a fix, the performance of this application was improved by approximately seven percent as measured by running most frequent usage scenarios before and after the fix.
– We also applied FOREPOST to two open-source application benchmarks, JPetStore and Dell DVD Store. FOREPOST automatically found rules that steered executions of

JPetStore and Dell DVD Store towards input data that increased the average execution time by 78.2 and 333.3 % as compared to random testing.
- We also conducted a controlled experiment to analyze the impact of a number of independent variables on the power of FOREPOST to identify performance bottlenecks. In this experiment we compared two versions of the engine: one that uses only the rules learned from execution traces (FOREPOST) and another one that also uses a subset of random inputs in addition to using generated rules, namely FOREPOST$_{RAND}$. Our results demonstrate that FOREPOST$_{RAND}$ helps improve the accuracy of identifying bottlenecks at the expense of finding less computationally expensive bottlenecks.

2 Background and the Problem

In this section we describe the state of the art and practice in performance testing, show a motivating example, and formulate the problem statement.

2.1 State of the Art and Practice

The random testing approach, as its name suggests, involves the random selection of test input data for input parameter values, which was shown remarkably effective and efficient for testing and bug finding (Bird and Munoz 1983). It is widely used in industry, and has been proved to be more effective than systematic testing approaches (Hamlet 1994, 2006; Park et al. 2012). Concurrently, another implementation of performance testing involves selecting a small subset of “good” test cases with which different testing objectives can be achieved (Kaner 2003). Specifically, more performance bottlenecks can be found in a shorter period of time. Good test cases are more likely to expose bugs and to produce results that yield additional insight into the behavior of the application under test (i.e., they are more informative and more useful for troubleshooting). Constructing good test cases requires significant insight into an AUT and its features and useful rules for selecting test input data.

Performance testing of enterprise applications is manual, laborious, costly, and not particularly effective. Several approaches were proposed to improve the efficiency of performance testing (Koziolek 2005; Avritzer and Weyuker 1996; Jin et al. 2012). For example, operational profile models the occurrence probabilities of functions and the distributions of parameter values, which has been introduced to test most frequently used operations (Koziolek 2005). Rule-based techniques are effective for discovering performance bottlenecks by identifying the problematic patterns from the source code, such as misunderstandings in API calls or problematic call sequences (Jin et al. 2012). However, these techniques always work for some specific types of performance bottlenecks, not widely used in industry. In practice, a prevalent method for performance testing is *intuitive testing*, which is a method for testers to exercise the AUT based on their intuition and experience, surmising probable errors (Cornelissen et al. 1995). Intuitive testing was first introduced in 1970s as an approach to use the experience of test engineers to focus on error-prone and relevant system functions without writing time-consuming test specifications. Thus it lowers pre-investment and procedural overhead costs (Cornelissen et al. 1995). When running many different test cases and observing application’s behavior, testers intuitively sense that there are certain properties of test cases that are likely to reveal performance bottlenecks. However, one of the major risk of intuitive testing is losing key people (i.e., key testers). The knowledge and experience of test engineers are gone when they leave the company. Training new testers is time-consuming and expensive. Thus, it is necessary to distill the properties of test cases that reveal performance bottlenecks automatically to avoid losing money and time. *Distilling these properties automatically into rules that describe how these properties affect performance of the application is a subgoal of our approach.*

In psychology, intuition means a faculty that enables people to acquire knowledge by linking relevant but spatially and temporally distributed facts and by recognizing and discarding irrelevant facts (Westcott 1968). What makes intuitive acquisition of knowledge difficult is how relevancy of facts is perceived. In software testing, facts describe properties of systems under test, and many properties may be partially relevant to an observed phenomenon. Intuition helps testers to (i) form abstractions by correctly assigning relevancy rankings to different facts, (ii) form hypotheses based on these abstractions, and (iii) test these hypotheses without going through a formal process. With FOREPOST, we partially automate the intuitive process of obtaining performance rules.

2.2 A Motivating Example

Consider a renter insurance program, *Renters*, designed and built by a major insurance company. A goal of this program is to compute quotes for insurance premiums for rental condominiums. Renters is written in Java and it contains close to 8500 methods that are invoked more than three million times over the course of a single end-to-end pass through the application. Its database contains approximately 78 million customer profiles, which are used as test input data for Renters. Inputs that cause heavy computations are sparse, and random test selection often does not perform a good job of systematically locating these inputs. A fundamental question of performance testing is how to select a manageable subset of the input data for performance test cases with which performance bottlenecks can be found faster and automatically.

Consider an example of how intuitive testing works for Renters. An experienced tester notices at some point that it takes more CPU and hardware resources (fact 1) to compute quotes for residents of the states California and Texas (fact 2). Independently, the database administrator casually mentions to the tester that a bigger number of transactions are executed by the database when this tester runs test cases in the afternoon (fact 3). Trying to find an answer to explain this phenomenon, the tester makes a mental note that test cases with northeastern states are usually completed by noon and new test cases with southwestern states are executed afterwards (fact 4). A few days later the tester sees a bonfire (fact 5) and remembers that someone's property was destroyed in wildfires in Oklahoma (fact 6). All of a sudden the tester experiences an epiphany – it takes more resources for Renters to execute tests for the states California and Texas because these states have the high probability of having wildfires. When test cases are run for wildfire states, more data is retrieved from the database and more computations are performed. The tester then identifies other wildfire states (e.g., Oklahoma) and creates test cases for these states, thereby concentrating on more challenging tests for Renters rather than blindly forcing all tests, which is unfortunately a common practice now (Murphy 2008). Moreover, even if the tester detects that the test cases for the states take more execution resources, it is also important to pinpoint the reason why test cases that consume more resources. When the tester looks into the execution information of these test cases, he finds that these test cases always execute some specific methods that take an unexpectedly long time to execute. Thereby, the tester identifies the performance bottlenecks and tries to optimize these methods to save time for testing. Furthermore, it is also helpful to detect performance bottlenecks if testers find that test cases for some states perform against their intuition. For example, some northern states like Minesota never have wildfires, so its insurance quotes relevant with wildfires should be nearing zero. However, some intensive checking for wildfire area for these states may still be performed. Hence, it is possible that these test cases invoke some unnecessary methods consuming more resources than necessary. The testers can look into the corresponding execution traces to pinpoint the potential performance bottlenecks.

This long and cumbersome procedure reflects what stakeholders have to go through frequently to find performance bottlenecks. Doing it can be avoided if, in our example there was a rule that specified that additional computations are performed when the input data includes a state where wildfires are frequent. The methods invoked by this input data that take more resource can be pinpointed automatically. Unfortunately, abstractions of rules that provide insight into the behavior of the AUT and the identification of performance bottlenecks automatically are difficult to obtain. For example, a rule may specify that the method `checkFraud` is always invoked when test cases are good and the values of the attribute `SecurityDeposit` of the table `Finances` are frequently retrieved from the backend database. This information helps performance testers to create a holistic view of testing, and to select test input data appropriately, thereby reducing the number of tests. Thus, these rules can be used to select better test cases and identify the performance bottlenecks automatically.

Rules for selecting test input data that quickly lead to finding performance bottlenecks are notoriously difficult to capture. Since these rules are buried in the source code, they are hard to locate manually. Test engineers must intimately know the functionality of the subject application under test, understand how programmers designed and implemented the application, and hypothesize on how the application behavior matches the requirements. Without having useful rules that summarize these requirements, it is difficult to define objectives that lead to selecting good test cases (Kaner 2003). Moreover, the performance bottlenecks are also difficult to locate manually, since the tester needs to understand exactly how the AUT executes with the selected input data and analyze each methods to pinpoint the ones which take more resources.

Currently, the state-of-the-art for finding useful rules is to use the experience and the intuition of the performance test engineers who spent time observing the behavior of AUTs when running manually constructed test cases. There is little automated support for discovering problems with performance testing. A recent work by Jiang et al. is the first that can automatically detect performance bottlenecks in the load testing results by analyzing performance logs (Jiang et al. 2009). However, the test inputs that cause performance bottlenecks are not located. Experience and intuition are the main tools that performance test engineers use to surmise probable errors (Glenford 1979; Cornelissen et al. 1995). Our goal is to automate the discovery of rules and abstractions that can help stakeholders pinpoint performance bottlenecks and to reduce the dependency on experience and intuition of test engineers.

2.3 Abstractions for Testing

Abstraction is a fundamental technique in computer science to approximate entities or objects in order to infer useful information about programs that use these entities or objects (Dijkstra 1976). Abstract interpretation, static type checking, and predicate abstraction are examples of mathematical frameworks. These frameworks allow scientists to design abstractions over mathematical structures in order to build models and prove properties of programs. Software model checking is one of the largest beneficiary fields of computer science where abstractions enable engineers to deal with the problem of state space explosion.

User-defined abstractions are most effective in the solution domain (i.e., the domain in which engineers use their ingenuity to solve problems (Hull et al. 2005, pages 87,109)). In the problem domain, mathematical abstractions are used to express semantics of requirements. Conversely, in the solution domain engineers go into implementation details. To realize requirements in the solution domain, engineers look for user-defined abstractions that are often implemented using ad-hoc techniques (e.g., mock objects that abstract missing program components (Freeman et al. 2004)). Thus, user-defined abstractions are most effective when they reflect the reality of the solution domain (Achenbach and Ostermann 2009).

Abstractions play a significant role in software testing (Beck 2003). For example, input partitioning is a technique to divide input domain into regions that contain equally useful values from a testing perspective (Ammann and Offutt 2008, page 150). With input partitioning, only one value from each region can be used to construct test cases instead of all values. Of course, proper partitioning requires testers to use right abstractions. For example, an abstraction of input values for a calculator that operates on integers could consist of negative, positive, and zero regions. Finding these abstractions for large and ultra-large enterprize-strength applications is an intellectually laborious and difficult exercise.

Useful abstractions for testing approximate the functionality of an application under test. For example, a useful abstraction for the Renters is that renters will pose high insurance risk if they have one or more prior insurance fraud convictions and deadbolt locks are not installed on premises. Using this abstraction testers can model the system as two main components: one that computes insurance premium for renters with clean history and the other for renters with fraud convictions. With this model, testers can partition inputs in two regions that correspond to the functionalities of these main components. Even though real-world systems exhibit much more complex behavior, useful abstractions enable testers to build effective test cases.

2.4 Performance Test Scripting Approaches

Typically, performance testing is accomplished using *test scripts*, which are programs that test engineers write to automate testing. These test scripts perform actions (i.e., invoking methods of exposed interfaces or mimicking user actions on GUI objects of the AUT) to feed input data into AUT and trigger computation. Test engineers write code in test scripts that guides the selection of test inputs. Typically, this is done using randomly selected input values or by using algorithms of combinatorial design interactions (Grindal et al. 2005). It is impossible to performance test applications without test scripts, since it is not feasible to engage hundreds of testers who simulate multiple users who call multiple methods with high frequency manually (Dustin et al. 1999; Fewster and Graham 1999; Kaner 1997; Molyneaux 2009).

Test scripts are typically written with one of the following frameworks: a GUI testing framework (e.g., QuickTestPro from HP Corp) or a backend server-directed performance tool such as JMeter, an open source software that is widely used to load test functional behavior and measure performance of applications. These frameworks form the basis on which performance testing is mostly done in industry. Performance test scripts imitate large numbers of users to create a significant load on the AUT. JMeter provides programming constructs that enable testers to automatically generate a large number of virtual users to send `HTTP` requests to web servers, thereby creating significant workloads. Natural measures of performance include throughput (i.e., the number of executed requests per second) and the average response time it takes to execute a request. A goal of performance testing is to determine what combinations of requests lead to higher response times and lower throughput, which are helpful to reveal performance bottlenecks in AUTs.

2.5 The Problem Statement

Our goal is to automate finding performance bottlenecks by executing the AUT on a small set of randomly chosen test input data, and then inferring rules with a high precision for selecting test input data automatically to find more performance bottlenecks in the AUT. Specifically, these are `if-then` rules that describe properties of input data that result in good performance test cases. The good performance test cases lead to increased computational workload on applications as compared to the bad performance test cases, which have smaller computational workload. For example, a rule may say "if inputs `convictedFraud` is `true` and `deadboltInstalled` is `false` then the test case is good." In this work, we supply automatically learned rules using a feedback mechanism to test scripts. These scripts parse these rules and use them to guide test input data selection automatically to steer the execution of the AUT towards the code that exposes performance bottlenecks.

In this paper, we accept a performance testing definition of what constitutes a *good test case*.

One of the goals of performance testing is to find test cases that worsen response time or throughput of the AUT or its latency. This can be achieved by adding more users to the AUT, which leads to intensive computations and increased computational workloads, and by finding input data that makes the AUT take more resources and time to compute results. Conversely, *bad test cases* are those that utilize very few resources and take much less time to execute as compared to good test cases. The next step is to automatically produce rules that describe good and bad test cases, and to automatically use these rules to select input data for further testing.

This system should also have the ability to correct itself. In order to do that, it needs to apply the learned rules on the test input data that are selected based on these rules, and then verify that these selected input data lead to predicted performance results. This process increases the probability that the learned rules express genuine causation between input values and performance-related workloads.

Finally, no performance testing is complete without providing sufficient clues to performance engineers where in the AUT the problems lurk. A main objective of performance analysis is to find bottlenecks (i.e., a single method that drags down the performance of the entire application which is easy to detect using profilers). However, it is difficult to find bottlenecks when there are hundreds of methods whose elapsed execution times are approximately the same, which is often the case in large-scale applications (Aguilera et al. 2003; Ammons et al. 2004). A problem that we solve in this paper is that once the input space is clustered into good and bad performance test cases using learned rules, we identify methods that are specific to good performance test cases, which are most likely to contribute to bottlenecks.

3 The FOREPOST Approach

In this section we explain the key ideas behind our approach, give an overview of *Feedback-ORiEnted PerfOrmance Software Testing (FOREPOST)*, explain the detailed algorithm, and describe its architecture and workflow finally.

3.1 A Birds-Eye View of our Solution

Our work is based on a single key idea that is backed up by multiple experimental observations – performance bottlenecks often exhibit patterns, and these patterns are specific to applications. These patterns are complicated; recall our motivating example from Section 2.2 where understanding a pattern comprises the knowledge of properties of input parameters and execution paths. Intuitive testing is a laborious and intellectually intensive way for test engineers to obtain these patterns by observing the performance behavior of software applications for different inputs and configurations. On the other hand, a main goal of machine learning and data mining is to obtain patterns from large quantities of data automatically. The essence of our work is to design an approach where we can use machine learning and data mining algorithms to obtain patterns that can concisely describe the performance behavior of software applications.

We reduce the problem of performance testing to pattern recognition, which assigns some output values to sets of input values (Bishop 2006). To recognize different patterns in software applications, terms are extracted from the source code of applications and then serve as the input to a machine learning algorithm that eventually places these software applications into some categories (Tian et al. 2009; McMillan et al. 2011; Linares-Vásquez et al. 2014). For example, classifiers are used to obtain patterns from software applications to predict defects in these applications. In this case, extracted terms from software applications are inputs and resolved defects are the outputs of a classifier. After a classifier is trained, it may predict future defects in software applications. Even though automatic classification approaches do not achieve perfect precision, they still enable stakeholders to quickly learn models that concisely describe different patterns in these software applications. Classification is widely used in mining software repositories and software traceability to solve problems for which deterministic algorithms are difficult or infeasible to apply.

In performance testing, using a classifier will result in learning a prediction model that approximates the AUT as a function that maps inputs to outputs. Inputs include the properties of the input values (e.g., the gender of the user or their income categories) and the outputs include ranges of performance behaviors (e.g., response time or memory usages). To make the prediction model reliable, only legitimate test inputs, which comes from actual workload, are used to build the prediction model. For example, 78 million customer profiles extracted from the database of Renters are used as test input data. A straightforward application of a machine learning classifier to performance testing is to run the AUT many times, collect execution data, learn the pattern, and use it to predict the future behavior. Unfortunately, this approach does not work well in practice for a number of reasons.

First, nontrivial AUTs have enormous numbers of combinations of different input data – these numbers are measured in hundreds of billions. Executing the AUT exhaustively with all combinations of input data is infeasible, and even collecting a representative sample is a big problem. Consider that performance bottlenecks are often exhibited for much smaller numbers of combinations on input data. If these combinations were known in advance, then there would be no need for performance testing, and all performance bottlenecks would be fixed easily. Selecting a sample of input data that does not exhibit any performance bottlenecks might reduce the effectiveness of classification for performance testing. Thus, it is important to steer the selection of input data using some kind of guidance from the previous runs of the AUT, and this is what we accomplish in FOREPOST.

Second, predicting some output does not explain how different input values contribute to this output. Performance testing concentrates on localizing performance bottlenecks in the code of the AUT, so that engineers can fix them, rather than on predicting how the AUT would behave for a given input. Our goal is to extract explanations or rules from learned models in classifiers that we use to learn the performance behavior of the AUT. We use these rules in a feedback-directed loop to guide selection of the input data for the AUT and to pinpoint performance problem in the code of the AUT. In that, it is the major contribution of FOREPOST.

3.2 An Overview of FOREPOST

In this section, we describe two key ideas on which FOREPOST is built: `1)` extracting rules from execution traces that describe relations between properties of input data and workloads of performance tests that are executed with this data and `2)` identifying bottleneck methods using these rules.

3.2.1 Obtaining Rules

As part of the first key idea, the instrumented AUT is initially running using a small number of randomly selected test input data. Its execution profiles are collected and automatically clustered into different groups that collectively describe different performance results of the AUT. For example, there can be two groups that are corresponding to good and bad performance test cases, respectively.

The set of values for the AUT inputs for good and bad test cases represent the input to a Machine Learning (ML) classification algorithm. In FOREPOST, we choose the rule learning algorithm, called Repeated Incremental Pruning to Produce Error Reduction (RIPPER) (Cohen 1995), to obtain the rules that guide the selection of test input data in test scripts. RIPPER is a rule learning algorithm, which is modified from the Incremental Reduced Error Pruning (IREP) (Furnkranz and Widmer 1994). It integrates pre-pruning and post-pruning into a learning phrase and follows a separate-and-conquer strategy. Each rule will be pruned right after it is generated, which is similar to the IREP. But the difference is that it chooses an alternative rule-value metric in the pruning phrase, provides a new stopping condition and optimizes the initial rule set, which is obtained by IREP.

This input of ML algorithm is described as implications of the form $V_{I_1}, \ldots, V_{I_k} \rightarrow T$, where V_{I_m} is the value of the input I_m and $T \in \{G, B\}$, with G and B representing good and bad test cases correspondingly. In fact, T is the summarized score of an execution trace that describes summarily whether this execution has evidence of performance bottlenecks. The ML classification algorithm learns the model and outputs rules that have the form $I_1 \odot V_{I_1} \bullet I_2 \odot V_{I_2} \bullet \ldots \bullet I_k \odot V_{I_k} \rightarrow T$, where $\odot$ is one of the relational operators (e.g., $>$ and $=$) and $\bullet$ is one of the logical connectors (i.e., $\wedge$ and $\vee$). These rules are instrumental in guiding the selection of the test input data in test scripts. For example, in a web application, if I_1 refers to a URL request "viewing the page of cat", and I_2 refers to a URL request "viewing the page of dog", the rule $(I_1 = V_{I_1}) \wedge (I_2 > V_{I_2}) \rightarrow G$ means that "viewing the page of cat"

V_{I_1} times and "viewing the page of dog" more than V_{I_2} times is good to trigger performance bottlenecks. The next test cases should be generated based on this rule. More detailed rules are provided in Table 3.

We first repeatedly run the experiment with the randomly selected initial seeds from the input space, which are different each time. Then, new values are selected from the input space either randomly, if rules are not available, or based on learned rules.

A feedback loop is formed by supplying these learned rules, which are obtained using the ML classification algorithm, back into the test script to automatically guide the selection of test input data. Using the newly learned rules, the test input data is partitioned and the cycle repeats. The test script selects inputs from different partitions, and the AUT is executed again. New rules are *re*-learned from the collected execution traces. If no new rules are learned after some time of testing, the partition of test inputs is stable with a high degree of probability. At this point the instrumentation can be removed and the testing can continue, and the test input data is selected using the learned rules.

3.2.2 Identifying Bottlenecks

Our goal is to help test engineers to automatically identify bottlenecks as method calls whose execution seriously affects the performance of the whole AUT. For example, consider a method that is periodically executed by a thread which checks if the content of some file is modified. While this method may be one of the bottlenecks, it is invoked in both good and bad test cases. Thus, its contribution to the resource consumption as the necessary part of the application logic does not lead to any insight that may resolve a performance problem. Our second key idea is to consider the most significant methods that occur in good test cases and that are not invoked, or have little to no significance, in bad test cases, where the significance of a method is a function of the resource consumption that its execution triggers. We measure resource consumption as a normalized weighted sum of (i) the number of times that this method is invoked, (ii) the total elapsed time of its invocations minus the elapsed time of all methods that are invoked from this method, and finally, (iii) the number of methods whose invocations are spawned from this method. In FOREPOST, Independent Component Analysis (ICA) is used to identify the performance bottlenecks. The detailed algorithm will be explained in Sections 3.3 and 3.4.

3.3 Blind Source Separation

Large applications contain multiple features, and each of these requirements is implemented using different methods. For example, in JPetStore, the high-level requirements are "place an order", "search an item", or "create an account" et al. Each AUT's run involves thousands of its methods that are invoked millions of times. The resulting execution trace is a mixture of different method invocations, each of which addresses a part of some features. These traces are very large. In order to identify most significant methods, we need an approach that allows us to (i) compress information in these traces and (ii) automatically break these traces into components that match high-level features in order to identify the methods with the most significant contributions to these components. Unfortunately, using transactional boundaries to separate information in traces is not always possible (e.g., when dealing with file operations or GUI frameworks). We reduced the complexity of the collected execution traces by categorizing them into components that roughly correspond to different features.

We draw an analogy between separating method invocations in execution traces into components that represent high-level features and a well-known problem of separating signals that represent different sources from a signal that is a mixture of these separate signals. This problem is known as *blind source separation (BSS)* (Parsons 2005, pages 13–18).

The idea of BSS is illustrated using a model where two people speak at the same time in a room with two microphones `M1` and `M2` as it is shown in Fig. 1. Their speech signals are designated as `source 1` and `source 2`. Each microphone captures the mixture of the signals `source 1` and `source 2`, which are the corresponding signal mixtures from `M1` and `M2` respectively. The original signals `source 1` and `source 2` are separated from the mixtures using a technique called *independent component analysis (ICA)* (Hyvärinen and Oja 2000; Grant et al. 2008), which we describe in Section 3.4. ICA is based on the assumption that different signals from different physical processes are statistically independent. For example, different features are often considered independent since they are implemented in applications as separate concerns (Parnas 1972; Tarr et al. 1999). When physical processes are realized (e.g., different people speak at the same time, or stocks are traded, or an application is run and its implementations of different features are executed in methods), these different signals are mixed, and these signal mixtures are recorded by some sensors. Using ICA, independent signals can be extracted from these mixtures with a high degree of precision.

BSS is implemented using ICA. Even though the idea of BSS is illustrated using the speech model, ICA is widely used in econometrics to find hidden factors in financial data, image denoising and feature extraction, face recognition, compression, watermarking, topic extraction, and automated concept location in source code (Grant et al. 2008).

In this paper we adapt the BSS model to automatically decompose execution traces into components that approximately match high-level features, and then identifying the methods with the most significant contributions to these components. Nontrivial applications implement quite a few high-level features in different methods that are executed in different threads, often concurrently. We view each feature as a source of a signal that consists of method calls. When an application is executed, multiple features are realized, and method invocations are mixed together in a mixed signal that is represented by the execution profile. Microphones are represented by instrumenters that capture program execution traces; multiple executions of the application with different input data are equivalent to different speakers talking at the same time, and as a result, multiple signal mixtures (i.e., execution traces for different input data with mixed realized features) are produced. With ICA, not only it is possible to separate these signal mixtures into components, but also to define

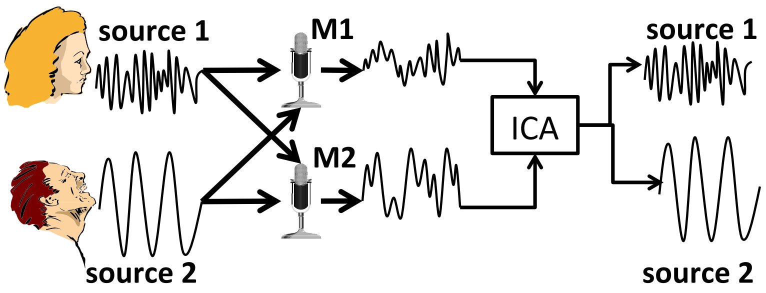


Fig. 1 A speech model of blind source separation

most significant constituents of these signals (i.e., method calls). We choose ICA because it works with non-Gaussian distributions of data, which is the case with FOREPOST.

3.4 Independent Component Analysis

A schematics of the ICA matrix decomposition is shown in Fig. 2. The equation $\mathbf{x} = \mathbf{A} \cdot \mathbf{s}$ described the process, where $\mathbf{x}$ is the matrix that contains the observed signal mixtures and $\mathbf{A}$ is the transformation or mixing matrix that is applied to the signal matrix $\mathbf{s}$. In our case, the matrix $\mathbf{x}$ is shown in Fig. 2 on the left hand side of the equal sign, and its rows correspond to application execution traces from different input data, and its columns correspond to method invocations that are observed for each trace.

Each element of the matrix $\mathbf{x}$ is calculated as $x_i^j = \sum_{k=1}^{n} \lambda_k \cdot M_{i,k}^j$, where λ are normalization coefficients computed for the entire matrix $\mathbf{x}$ to ensure $0 \leq x_i^j \leq 1$, M are different metrics that are considered for method i in the trace j. For different types of applications, different metrics can be considered. For example, in a generic application, matrix $\mathbf{x}$ is calculated with three different metrics, the number of times that the method j is invoked in the trace i ($M_{i,1}^j$), the total elapsed time of these invocations minus the elapsed time of all methods that are invoked from this method in this trace ($M_{i,2}^j$), and the number of methods that are invoked from this method ($M_{i,3}^j$). In a database application, matrix $\mathbf{x}$ can be calculated with two additional metrics, the number of attributes that this method accesses in the databases ($M_{i,4}^j$), and the amount of data that this method transfers between the AUT and the databases ($M_{i,5}^j$). For example, assume there is a method a, which is invoked 20 times during the execution, totally takes 32.8 ms to execute, calls methods b (8.1 ms) and c (2.3 ms), and accesses 12 attributes in the database for transferring totally 13.7 kb data. According to the equation, its weight is equal to $\lambda_1 \cdot 20 + \lambda_2 \cdot (32.8 - 8.1 - 2.3) + \lambda_3 \cdot 2 + \lambda_4 \cdot 12 + \lambda_5 \cdot 13.7$. Naturally, $x_i^j = 0$ means that the method i is not invoked in the trace j, while $x_i^j = 1$ means that the given method makes the most significant contribution to the computation in the given trace.

Using ICA, the matrix $\mathbf{x}$ is decomposed into a transformation and a signal matrices that are shown on the right hand side of the equal sign in Fig. 2. The input to ICA is the matrix $\mathbf{x}$ and the number of source signals, which in our case is the number of features (features in the Fig. 2) implemented in the application. The elements of the matrix $\mathbf{A}$, A_p^q, specify the weights that each profile p contributes to executing code that implements the feature q, and the elements of the matrix $\mathbf{s}$, s_q^k, specify the weights that each method k contributes to executing code that implements the feature q. Methods that have the highest weights for the given features are considered the most significant and interesting for troubleshooting performance bottlenecks. This is a hypothesis that we evaluate in Sections 4 and 5.

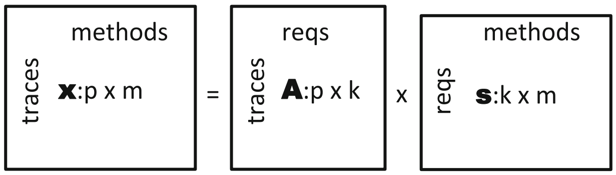


Fig. 2 Schematics of the ICA matrix decomposition

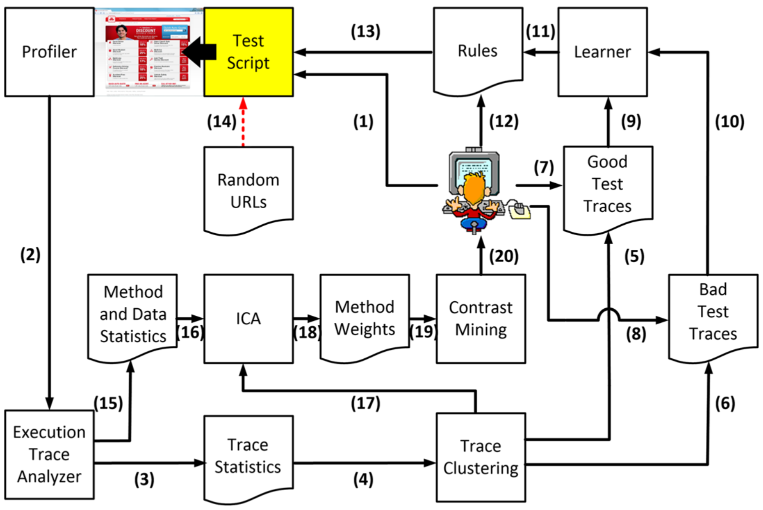


Fig. 3 The architecture and workflow of FOREPOST and FOREPOST$_{RAND}$. FOREPOST does not contain the step 14

3.5 FOREPOST and FOREPOST$_{RAND}$ Architecture and Workflow

The architecture of FOREPOST and FOREPOST$_{RAND}$ is shown in Fig. 3. Solid arrows show command and data flows between components, and numbers in circles indicate the sequence of operations in the workflow. The beginning of the workflow is shown with the fat arrow that indicates that the *Test Script* executes the application by simulating users and invoking methods of the AUT interfaces. The *Test Script* is written (1) by the test engineer as part of automating application testing as we described in Section 2.4.

Once the test script starts executing the application, its execution traces are collected (2) by the *Profiler*, and these traces are forwarded to the *Execution Trace Analyzer*, which produces (3) the *Trace Statistics*. We implemented the *Profiler* using the TPTP framework.[1] These statistics contain information on each trace, such as the number of invoked methods, the elapsed time it takes to complete the end-to-end application run, the number of threads, and the number of unique methods that were invoked in this trace. The trace statistics are supplied (4) to the module *Trace Clustering*, which uses the average execution time to perform unsupervised clustering of these traces into two groups that correspond to (5) *Good* and (6) *Bad test traces*. The user can review the results of clustering and (7,8) reassign the clustered traces if needed. These clustered traces are supplied (9,10) to the *Learner*, which uses a ML algorithm, RIPPER (Cohen 1995), to learn the classification model and (11) output rules that were described in Section 3.2. The user can review (12) these rules and mark some of them as erroneous if the user has sufficient evidence to do so. Next, the rules are supplied (13) to the *Test Script*. In FOREPOST, once the *Test Script* receives a new set of rules, it partitions the input space into blocks according to these rules and starts forming test inputs by selecting one input from each block. In FOREPOST$_{RAND}$, the *Test Script* is a combination of random input data and several blocks of input space that correspond to different rules. The major difference in the architecture of FOREPOST$_{RAND}$

[1] http://eclipse.org/tptp, last checked August 12, 2015

is that it considers random URLs as input data that is shown in step `(14)`, whereas the original version of FOREPOST (Grechanik et al. 2012) does not contain this step `(14)`. We expect that adding the random input data could enlarge the test coverage to find more potential performance bottlenecks. After generating new input data, the *Profiler* collects execution traces of these new test runs. The cycle repeats with new rules that are learned after several passes, and the input space is repartitioned adaptively to accommodate these rules. We implemented the ML part of FOREPOST using JRip,[2] which is implemented by Weka (Witten and Frank 2005).

The test input data is extracted from existing repositories or databases. This is a common practice in industry, and we confirmed it with different performance testing professionals after interviewing professionals at IBM, Accenture, two large health insurance companies, a biopharmaceutical company, two large supermarket chains, and three major banks. Recall that the application Renters has a database that contains approximately 78 million customer profiles, which are used as the test input data for different applications including Renters itself. We repeatedly ran the experiment with the randomly selected initial seeds from the input space, which are different each time. The new values are selected from the input space either randomly, if rules are not available, or are based on the newly learned rules.

Finally, recall from Section 2.5 that once the input space is partitioned into clusters that lead to good and bad test cases, we want to find methods that are specific to good performance test cases and that are most likely to contribute to bottlenecks. This task is accomplished in parallel to computing rules, and it starts when the *Execution Trace Analyzer* produces `(15)` the method and data statistics of each trace, and then uses this information to construct `(16)` two matrices $\mathbf{x_B}$ and $\mathbf{x_G}$ for bad and good test cases correspondingly, based on the information provided by `(17)`. Constructing these matrices is done as described in Section 3.4. Once these matrices are constructed, ICA decomposes them `(18)` into the matrices $\mathbf{s_B}$ and $\mathbf{s_G}$ corresponding to bad and good tests. Recall that our key idea is to consider the most significant methods that occur in good test cases and that are not invoked, or have little to no significance in bad test cases. Cross-referencing the matrices $\mathbf{s_B}$ and $\mathbf{s_G}$, which specify the method weights for different features, the *Contrast Mining* `(19)` compares the method weights in both of good and bad test cases, and determines the top methods that the performance testers should look at `(20)` to identify and debug possible performance bottlenecks. This step completes the workflow of FOREPOST. The detailed algorithm for identifying bottlenecks, including ICA and *Contrast Mining*, is shown in the Section 3.6.

3.6 The Algorithm for Identifying Bottlenecks

In this section we describe our algorithm for identifying bottlenecks using FOREPOST. This algorithm clusters the traces based on the trace statistics, and then generates the matrices for both good and bad test cases. By using the ICA algorithm, it calculates the weight for each method in both good and bad test cases. If one method is significant in good test cases but not significant in bad test cases, then we conjecture that it is likely to be a bottleneck. This algorithm provides a ranked lists of bottlenecks as its final output. The algorithm `FOREPOST` is shown in Algorithm 1. `FOREPOST` takes as its input the set of captured execution traces, T, and the signal threshold, U, which is used to select methods whose

[2]http://weka.sourceforge.net/doc.stable/weka/classifiers/rules/JRip.html, last checked Apr 10, 2015

signals indicate their significant contribution in execution traces. The set of methods that are potential bottlenecks, B, is computed and returned in line `17` of the algorithm.

In step `2` the algorithm initializes to the empty set the set of bottlenecks and the set of clusters that contain execution traces that are matched to good and bad test cases. In step `3` the procedure `ClusterTraces` is called that automatically clusters execution traces from the set T into good (C_{good}) and bad (C_{bad}) test case clusters. Next, in steps `4` and `5` the procedure `CreateSignalMixtureMatrix` is called on clusters of traces that correspond to bad and good test cases respectively to construct two matrices $\mathbf{x_b}$ and $\mathbf{x_g}$ corresponding to bad and good test cases, as described in Section 3.5. In step `6` and `7`, the procedure `ICA` decomposes these matrices into the matrices $\mathbf{s_b}$ and $\mathbf{s_g}$ corresponding to bad and good test cases, as described in Section 3.5.

Next, the algorithm implements the *Contrast Mining* component in steps `8–15`, mining all the methods for all the feature components in the decomposed matrices. More specifically, for each method whose signal exists in the transformation and signal matrices that correspond to good cases, we compare if this method does not occur in the counterpart matrices for bad test case decompositions. Alternatively, if the same method from the same component occurs, then the distance between these two signals in the good and bad test should be quite large. The distance is calculated as shown in (1), where $M_g^i = M_b^k \wedge R_g^j = R_b^l$, M = method, g = good, b = bad, R = component, which means the same method from the same component occurs.

$$D_{e_g} = \sum_{i=0}^{N_{M_g}} \sum_{j=0}^{N_{R_g}} \sqrt{\left(SL_g^{ij} - SL_b^{kl}\right)^2} \tag{1}$$

In this equation, SL = signal, D_{e_g} = distance for each method, N_{M_g} = the number of good methods, N_{R_g} = the number of components. We consider this distance as the weight for each method, and rank all the methods based on their weights, as the step `16` shows. This ranked list B_{RANK} is returned in line `17` as the algorithm terminates.

Algorithm 1 The algorithm for identifying bottlenecks.

```
1:  ForePost( Execution Traces $T$, Signal Threshold $U$ )
2:  $B \leftarrow \emptyset, C_{good} \leftarrow \emptyset, C_{bad} \leftarrow \emptyset$ {Initialize values for the set of bottlenecks, the set of clusters that co-
    ntain execution traces that are matched to good and bad test cases.}
3:  ClusterTraces($T$) $\mapsto (C_{good} \mapsto \{t_g\}, C_{bad} \mapsto \{t_b\}), t_g, t_b \in T, t_b \cap t_b = \emptyset$
4:  CreateSignalMixtureMatrix($C_{good}$) $\mapsto$ matrix $x_g$
5:  CreateSignalMixtureMatrix($C_{bad}$) $\mapsto$ matrix $x_b$
6:  ICA($x_g$) $\mapsto ((A_g, s_g) \mapsto (L_g \mapsto (\{< M_g, R_g, SL_g >\})))$
7:  ICA($x_b$) $\mapsto ((A_b, s_b) \mapsto (L_b \mapsto (\{< M_b, R_b, SL_b >\})))$
8:  for all $e_g \mapsto \{< M_g^i, R_g^j, SL_g^{ij} >\} \in L_g$ do
9:      for all $e_b \mapsto \{< M_b^k, R_b^l, SL_b^{kl} >\} \in L_b$ do
10:         if $M_g^i = M_b^k \wedge R_g^j = R_b^l$ then
11:             Calculate $D_{e_g}$
12:             $B \leftarrow B \cup < e_g, D_{e_g} >$
13:         end if
14:     end for
15: end for
16: Rank $B$
17: return $B_{RANK}$
```

4 Evaluation

In this section, we state our research questions (RQs) and we describe how we evaluated FOREPOST on three applications: the commercial application, Renters, that we described as our motivating example in Section 2.2 and two open-source applications, JPetStore and Dell DVD Store, which are frequently used as industry benchmarks.

4.1 Research Questions

In this paper, we make one major claim – FOREPOST is *more effective* than random testing, which is a popular industrial approach. We define "more effective" in two ways: (i) finding inputs that lead to significantly higher computational workloads and (ii) finding performance bottlenecks. We seek to answer the following research questions:

RQ$_1$: How effective is FOREPOST in finding test input data that steer applications towards more computationally intensive executions and identifying bottlenecks with a high degree of automation?

RQ$_2$: How do different parameters (or independent variables) of FOREPOST affect its performance for detecting injected bottlenecks in controlled experiments?

RQ$_3$: How effective is FOREPOST$_{RAND}$ in finding test input data that steer applications towards more computationally intensive executions and identifying bottlenecks with a high degree of automation?

The rationale for these RQs lies in the complexity of the process of detecting performance bottlenecks. Not all methods are bottlenecks whose execution times are large. For example, the method *main* can be described as a bottleneck, since it takes naturally the most time to execute. However, it is unlikely that a solution may exist to reduce its execution time significantly. Thus, the effectiveness of bottleneck detection involves not only the precision with which performance bottlenecks are identified, but also in how fast they can be found and how different parameters affect this process.

In order to address these RQs, we conducted three empirical studies. In this section, we first describe the subject applications used in the studies, then we cover the methodology and variables for each empirical study. The results are presented in Section 5.

Table 1 Characteristics of the insurance application Renters

Renters Component	Size [LOC]	NOC	NOM	NOA	MCC	NOP
Authorization	742	3	26	1	4.65	1
Utils	15,283	16	1,623	1,170	1.52	9
Libs	85,892	284	6,390	5,752	1.68	26
Eventing	267	3	11	1	4.27	1
AppWeb	8,318	116	448	351	1.92	11
Total	110,502	422	8,498	7,275	–	48

Size = lines of code (LOC), *NOC* = number of classes, *NOM* = number of methods, *NOA* = number of attributes, *MCC* = Average McCabe cyclomatic Complexity, *NOP* = number of packages

4.2 Subject AUTs and Experimental Hardware

We evaluate FOREPOST on three subject applications: Renters, JPetStore and Dell DVD Store. Renters is a commercial medium-size application that is built and deployed by a major insurance company. Renters serves over 50,000 daily customers in the U.S. and it has been deployed for over seven years. JPetStore and Dell DVD Store are open-source applications that are often used as industry benchmarks, since they are highly representative of enterprise-level database-centric applications.

The Renters is a J2EE application that calculates the insurance premiums for rental condominium. Its software metrics are shown in Table 1. The backend database is DB2 running on the IBM Mainframe, its schema contains over 700 tables including close to 15,000 attributes that contain data on over 78 million customers, which are used as the input to FOREPOST. The application accepts input values using 89 GUI objects. The total number of combinations of input data is approximately 10^{65}, making it infeasible to comprehensively test Renters. We used Renters in our motivating example in Section 2.2.

JPetStore is a Java implementation of the PetStore benchmark, where users can browse and purchase pets, and rate their purchases. This sample application is typical in using the capabilities of the underlying component infrastructures that enable robust, scalable, portable, and maintainable e-business commercial applications. It comes with full source code and documentation, therefore, we used it in the evaluation of FOREPOST and demonstrated that we can build scalable security mechanisms into enterprise solutions. We used iBatis JPetStore 4.0.5.[3] JPetStore has 2139 lines of code, 386 methods, 36 classes in 8 packages, with the average cyclomatic complexity of 1.224; it is deployed using the web server Tomcat 6.0.35 and it uses Derby as its backend database.

The Dell DVD Store[4,5] is an open source simulation of an online e-commerce site, and it is implemented in MySQL along with driver programs and web applications. For the evaluation, we injected artificial bottlenecks into Dell DVD Store for experiments. It contains 32 methods totally, and it uses MySQL as its backend database. For both of the JPetStore and Dell DVD Store, we have an initial set of URLs as the input for FOREPOST.

The experiments on Renters were carried out at the premises of the insurance company using Dell Precision T7500 with a Six Core Intel Xeon Processor X5675, 3.06GHz,12M L3, 6.4GT/s, 24GB, DDR3 RDIMM RAM, 1333MHz. The experiments with JPetStore were carried out using two Dell PowerEdge R720 servers each with two eight-core Intel Xeon CPUs E5-2609 2.40 GHz, 10M, 6.4GT/s, 32GB RAM, 1066 MHz. The experiments with Dell DVD Store were carried out using one Thinkpad W530 laptop with an Intel Core i7-2640M processor, 32GB DDR3 RAM.

4.3 Research Question 1

Our goal is to determine which approach is better for finding good performance test cases faster. Given the complexity of the subject applications, it is not clear with what input data the performance can be worsened significantly for these applications. In addition, given the large space of the input data, it is not feasible to run these applications on all the inputs to

[3] http://sourceforge.net/projects/ibatisjpetstore, last checked Apr 10, 2015

[4] http://en.community.dell.com/techcenter/extras/w/wiki/dvd-store.aspx, last checked Apr 10, 2015

[5] http://linux.dell.com/dvdstore/, last checked Apr 10, 2015

obtain the worst performing execution profiles. These limitations dictate the methodology of our experimental design, specifically for choosing the competitive approaches to FOREPOST. We selected the random testing as the main competitive approach to FOREPOST, since it is widely used in industry and it has been proved to consistently outperform different systematic testing approaches (Park et al. 2012; Hamlet 1994). To support our claims in this paper, our goal is to show, with strong statistical significance, under what conditions FOREPOST outperforms random testing.

In designing the methodology for this experiment we aligned with the guidelines for statistical tests to assess randomized algorithms in software engineering (Arcuri and Briand 2011). Our goal was to collect highly representative samples of data when applying different approaches, perform statistical tests on these samples, and draw conclusions from these tests. Since our experiments involved random selection of input data, it was necessary to conduct the experiments multiple times and pick the average to avoid skewed results. We ran each experiment at least 50 times with each approach on the Renters to consider the collected data as a good representative sample.

JPetStore is based on the client-server architecture, where its GUI front end is web-based and it communicates with the J2EE-based backend that accepts HTTP requests in the form of URLs containing an address to different components and parameters for those components. For example, a URL can contain the address to the component that performs checkout and its parameters could contain the session ID. We define a set of URL requests that originate from a single user as a *transaction*. The JPetStore backend can serve multiple URL requests from multiple users concurrently. Depending on the type of URL requests in these transactions and their frequencies, some transactions may cause the backend server of JPetStore to take longer time to execute.

To obtain URL requests that exercise different components of JPetStore, we used the spider tool in JMeter to traverse the web interface of JPetStore, and recorded the URLs that were sent to the backend during this process. In random testing, multiple URLs were randomly selected to form a transaction. In FOREPOST, the URL selection process was guided by the learned rules. We limited the number of URLs in each transaction to 100. This number was chosen experimentally based on our observations of JPetStore users who explored approximately 100 URLs before switching to other activities. Increasing the number of certain URL requests in transactions at expense of not including other URL requests may lead to increased workloads, and the goal of our experimental evaluation is to show that FOREPOST eventually selects test input data (i.e., customer profiles for Renters or combinations of URLs for JPetStore and Dell DVD Store) that lead to increased workloads when compared to the competitive approaches.

When testing JPetStore and Dell DVD Store, URLs in a transaction are issued to the backend consecutively to simulate a single user. Multiple transactions are randomly selected and issued in parallel to simulate concurrent users using the system. During the testing we used different numbers of concurrent transactions, and measured the average time required by AUT backend to execute a transaction. A goal of this performance testing was to find combinations of different URLs in transactions for different concurrent users that lead to significant increase in average time per transaction, which is often correlated with the presence of performance bottlenecks.

We measured inputs as transactional units, where one transactional unit for JPetStore or Dell DVD Store is a combination of different URLs that map to different functional units of the application. At first, these URLs are randomly selected into a transactional unit, but as rules that describe limits on specific URLs are learned, some URLs will be given more preference for inclusion in transactions. For Renters, a transactional unit is an end-to-end run of the application with an input that is a customer profile that comes from the application database.

Dependent variables are the throughput or the average number of transactions or runs that the subject AUTs can sustain under the load, the average time that it takes to execute a transaction or run the AUT end to end. Thus, if an approach achieves a lower throughput or higher average time per transaction with some approach, it means that this particular approach finds test input data which are more likely to expose performance. The effects of other variables (the structure of AUT and the types and semantics of input parameters) are minimized by the design of this experiment.

4.4 Research Question 2

The goal of Empirical Study 2 is to provide empirical evidence to answer the following two questions. **The first one is:** ***can FOREPOST identify injected bottlenecks?*** To test the sensitivity of FOREPOST in detecting performance bottlenecks, we introduced different artificial bottlenecks, such as obvious bottlenecks and borderline bottlenecks. The obvious bottlenecks are computationally expensive operations that have a clear impact on software performance, but the borderline bottlenecks are the operations that may or may not be spotted as potential bottlenecks. With different injected bottlenecks, can FOREPOST identify the borderline bottlenecks correctly? If not, what kind of bottlenecks can or can not be found?

We added two different groups of delays into JPetStore as bottlenecks. The first group contains bottlenecks with exactly the same delay, whereas the second group contains bottlenecks with different length of delays. The bottlenecks#1 contain methods with the same delay of 0.1s in each bottleneck, and the bottlenecks#2 contain methods with different delays (e.g., 0.05 s, 0.1 s and 0.15 s). The artificial bottlenecks were injected into nine methods from the 386 probed methods. On the other hand, we injected one group of bottlenecks with the same delay into Dell DVD Store. Since Dell DVD Store only contains 32 native methods, we decided to inject the artificial bottlenecks into both the Dell DVD Store source code and the standard libraries. Dell DVD Store uses MySQL as its backend database, therefore, we probed methods from the library called *mysql-connector-java.jar* and injected bottlenecks into it. Five bottlenecks were injected into Dell DVD Store, two of them were injected in the source code and three of them were injected in the standard library. Since some of the bottlenecks were injected in the library, these bottlenecks would be invoked more frequently than the methods from the source code. Tracing methods from the standard libraries that come with the underlying platforms imposed much greater overhead. Thus, we decided to inject only five bottlenecks for Dell DVD Store (as opposed to nine for JPetStore, see Table 11). To avoid any threats to validity in this empirical study, we randomly selected methods from the library to inject these bottlenecks.

To make sure that the injected bottlenecks are going to be representative, before injecting these bottlenecks, we ran FOREPOST on JPetStore to find the original bottlenecks (i.e., methods from the original code that were ranked on the top when no artificial bottlenecks were injected), as well as the original positions of the injected bottlenecks. We chose nine artificial bottlenecks and the results are shown in Table 12. The injected bottlenecks are ranked on low positions, that implies that our injected bottlenecks are not really original bottlenecks and we choose them randomly. After injecting bottlenecks, some of the artificial bottlenecks are ranked in the top ten results, but the original bottlenecks are ranked on lower positions , implying that the lengths of delays which we injected are significant enough to be detected.

The second question is: ***how do different parameters (or independent variables) in FOREPOST affect its performance for detecting injected bottlenecks?*** To answer this question, we introduced a controlled experiment for sensitivity analysis on FOREPOST. In the sensitivity analysis, we considered the following two key parameters, namely, the *number of profiles* collected for learning rules and the *number of iterations*, as they affect the effectiveness of the rules. Furthermore, the *number of artificial bottlenecks* and the *number of users* may also impact the performance of FOREPOST in both of finding inputs steering application towards computationally intensive executions and identifying performance bottlenecks. All in all, four independent variables are investigated in the sensitivity analysis. Since our experiments involved random selection of input data, it was important to conduct these experiments multiple times to avoid skewed results. In this study, we ran each configuration five times and reported the average results.

The values of four independent variables are shown in Table 2. The first independent variable is the *number of profiles* (i.e., n_p) that needs to be collected in order to enable learning rules. FOREPOST collects the execution traces that saved as profiles, and then uses the average execution time to cluster these traces into two classes corresponding to *Good* and *Bad* traces. Next, the Learner analyzes them via a machine learning algorithm which extracts the rules. Intuitively, the number of collected profiles can affect the resulting rules. For example, the rules extracted from only ten profiles should contain different information from execution traces or profiles, as compared to the configuration containing 15 profiles. In our sensitivity analysis, the numbers of profiles are set to 10, 15, and 20. Our goal is to empirically investigate whether the number of profiles has substantial impact on the accuracy of FOREPOST.

The second independent variable is called the *number of iterations* (i.e., n_i), which is defined as the process between the generations of two sets of rules. For example, setting the number of iterations to two means that FOREPOST uses the ICA algorithm to identify the bottlenecks after the second round of learning rules. The number of iterations are set to 1, 2, 3, and 4. Intuitively, the rules tend to converge as the number of iterations increases. Our goal is to analyze the performance of FOREPOST after different numbers of iterations.

The third independent variable is the *number of bottlenecks* (i.e., n_b), which is the number of artificial performance bottlenecks injected into subject applications. Artificial delays were injected randomly into methods for simulating the realistic performance bottlenecks. The numbers of bottlenecks are set to 6, 9, and 12, and all bottlenecks have the same delay. Our goal is to empirically investigate the performance of FOREPOST on detecting different numbers of performance bottlenecks. All the artificial bottlenecks are shown in Tables 8, 9, and 10, which are located in Appendix.

The fourth independent variable is the *number of users* (i.e., n_u) that send URL requests simultaneously to the subject applications. The numbers of users are set to 5, 10, and 15. Using multiple users may lead to different AUT performance behaviors, where multi-threading, synchronization and database transactions may expose new types of performance

Table 2 Independent variables in sensitivity analysis

Profiles n_p = number of profiles for learning rules, iterations n_i = times of learning rules, bottlenecks n_b = number of artificial bottlenecks, users n_u = number of users

Factors	Value
profiles n_p	10, 15, 20
iterations n_i	1, 2, 3, 4
bottlenecks n_b	6, 9, 12
users n_u	5, 10, 15

bottlenecks. Our goal is to empirical analyze the performance of FOREPOST with different numbers of users.

4.5 Research Question 3

FOREPOST$_{RAND}$ is a combinational testing approach, which considers both the random input data and the specific inputs based on generated rules. We instantiated and evaluated FOREPOST$_{RAND}$ on JPetStore and Dell DVD Store. The main question addressed is whether considering random inputs in addition to generated rules is useful in terms of identifying known bottlenecks. In this empirical study, we fixed the independent variables in both of FOREPOST and FOREPOST$_{RAND}$ to compare these two approaches side by side.

5 Results

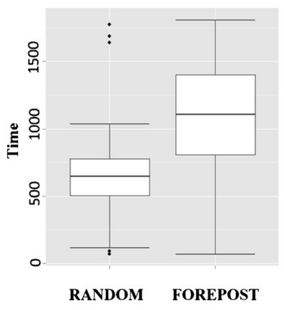
In this section, we describe and analyze the results obtained from our experiments with Renters, JPetStore and Dell DVD Store. We provide only parts of the results in this paper. The complete results of all our experiments are shown in our online appendix.[6]

5.1 Research Question 1

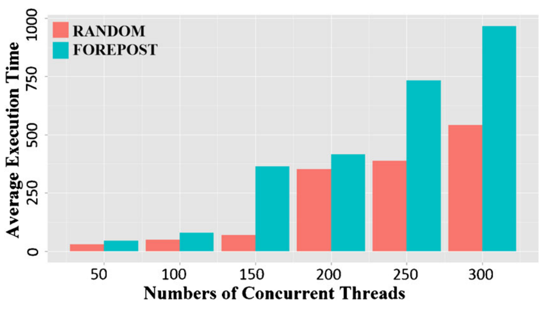
Finding Test Inputs for Increased Workloads. The results for Renters are shown in the box-and-whisker plots in Fig. 4a that summarize the execution times for end-to-end single application runs with different test input data. We first calculated the effect size to compare the FOREPOST with RANDOM using Cohen's d (Cohen 2013). The result is 1.2. According to Cohen's definition, the value of the effect size is large (≥ 0.8) implying that there is a difference between the execution times for RANDOM and FOREPOST. To further test the NULL hypotheses that there is no significant difference between the execution time for the random and FOREPOST approaches, we performed statistical tests for two paired sample means. Before applying paired significance test, we first applied the Shapiro-Wilk Normality Test (Shapiro and Wilk 1965) to check the normality distribution assumption. The results show that the sample data does not follow normal distribution even at the 0.01 significance level. Therefore, we chose to use the Wilcoxon Signed-Rank Test (Wilcoxon 1945) to compare the two sample sets, because it is suitable for the case that the sample data may not be normally distributed (Lowry 2014). The results of the statistical test allow us to reject the NULL hypotheses and accept the alternative hypotheses with strong statistical significance ($p < 0.0001$), which states that **FOREPOST is more effective at finding test input data that steers applications towards more computationally intensive executions than random testing**, thus addressing RQ$_1$.

This conclusion is confirmed by the results for JPetStore and Dell DVD Store that are shown in Fig. 4b and c, which are the average end-to-end execution times for five runs. In JPetStore, while in performing random testing, it takes on average 542.1 seconds to execute 300 transactions. With FOREPOST, executing 300 transactions takes on average 965.8 seconds, which shows 78.2 % increase. In Dell DVD Store, while performing random testing, it takes on average 100.0 seconds to execute 300 transactions. With FOREPOST, executing 300 transactions takes on average 433.3 seconds, which shows 333.3 % increase.

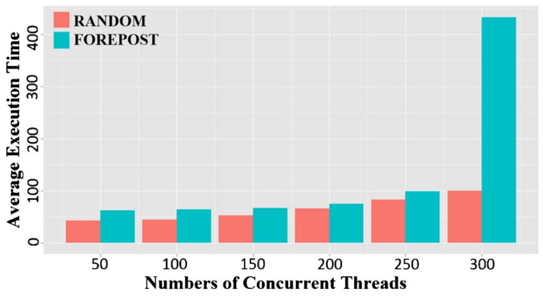
[6]http://www.cs.wm.edu/semeru/data/EMSE-forepost/



(a) Renters application



(b) JPetStore application



(c) Dell DVD Store application

Fig. 4 The summary of the results for Empirical Study 1. The *box-and-whisker* plots for Renters are shown in Figure (**a**), where the time for end-to-end runs is measured 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 *thicker vertical line* extends from the minimum to the maximum value. The *bar graphs* for JPetStore and Dell DVD Store are shown in Figure (**b**) and (**c**), where the bars represent average times per transaction in seconds for the Random and FOREPOST approaches for different numbers of concurrent transactions ranging from 50 to 300

This implies that FOREPOST outperforms random testing by more than one order of magnitude. Random testing is evaluated on the instrumented JPetStore and Dell DVD Store, so that the cost of instrumentation is evenly factored into the experimental results. FOREPOST has a large overhead, close to 80 % of the baseline execution time, however, once rules are learned and stabilized, they can be used to partition the input space without using instrumentation.

Identifying Bottlenecks and Learned Rules When applying the algorithm for identifying bottlenecks (see Section 3.6) on Renters, we obtained a list of top 30 methods that the algorithm identified as potential performance bottlenecks out of approximately 8,500 methods. To evaluate how effective this algorithm is, we asked the insurance company to allocate the most experienced developer and tester for Renters to review this list and provide feedback on it. According to the management of the insurance company, it was the first time when a developer and a tester were in the same room together to review results of testing.

The reviewing process started with the top bottleneck method, `checkWildFireArea`. The developer immediately said that FOREPOST did not work since this method could not be a bottleneck for a simple reason – this method computes insurance quotes only for U.S. states that have wildfires, and FOREPOST selected test input data for northern states like Minnesota that never have wildfires. We explained that FOREPOST automatically selected the method `checkWildFireArea` as important because its weight was significant in execution traces for good test cases, and it was absent in traces for bad test cases. It meant that this method was invoked many times for the state of Minnesota and other northern states, even though its contribution in computing insurance quotes was zero for these states. Invoking this method consumes more resources and time in addition to significantly increasing the number of interactions with the backend databases. After hearing our arguments, the developer and the tester told us that they would review the architecture documents and the source code and get back to us.

A day later they got back with a message that this and few other methods that FOREPOST identified as bottlenecks were true bottlenecks. It turned out that the implementation of the `Visitor` pattern in Renters had a bug, which resulted in incorrect invocations of the method `checkWildFireArea`. Even though it did not contribute anything to computing the insurance quote, it consumed significant resources. After implementing a fix based on the feedback from FOREPOST, the performance of Renters increased by approximately seven percent, thus addressing RQ_1 that **FOREPOST is effective at identifying bottlenecks**. More experiments of identifying bottlenecks in FOREPOST presented in the Sections 5.2 and 5.3 also support this conclusion.

Examples of learned rules are shown in Table 3. When professionals from the insurance company looked at these and other rules in more depth, they identified certain patterns that indicated that these rules were logical and matches some features. For example, the rules `R-1` and `R-3` point out to strange and inconsistent insurance quote inputs, where

Table 3 Selected rules that are learned for **R**enters and **J**PetStore, where the first letters of the names of the AUTs are used in the names of rules to designate to which AUTs these rules belong

Rule	Antecedent	Cons
R–1	(`customer.numberOfResidents` $\leq$ 2)$\wedge$ (`coverages.limitPerOccurrence` $\geq$ 400000)$\wedge$ (`preEligibility.numberOfWildAnimals` $\leq$ 1)	Good
R–2	(`adjustments.homeAutoDiscount` = 2)$\wedge$ (`adjustments.fireOrSmokeAlarm` = `LOCAL PLUS CENTRAL`)$\wedge$ (`dwelling.construction` = `MASONRY VENEER`)$\wedge$ (`coverages.limitEachPerson` $\leq$ 5000)	Bad
R–3	(`coverages.deductiblePerOccurrence` $\leq$ 500)$\wedge$ (`adjustments.burglarBarsQuickRelease` = `Y`)$\wedge$ (`nurseDetails.prescribeMedicine` = `Y`)$\wedge$ (`coverages.limitPerOccurrence` $\geq$ 500000)	Good
J–1	(`viewItem_EST-4` $\leq$ 5) $\wedge$ (`viewCategory_CATS` $\leq$ 23)$\wedge$ (`viewItem_EST-5` $\leq$ 6) $\wedge$ (`Checkout` $\geq$ 269)$\wedge$ (`Updatecart` $\geq$ 183) $\wedge$ (`AddItem_EST-6` $\geq$ 252)$\wedge$ (`viewCategory_EST-6` $\geq$ 71)	Good
J–2	(`viewItem_EST-4` $\leq$ 5) $\wedge$ (`viewCategory_CATS` $\leq$ 0)	Bad

The last column (Cons) designates the consequent of the rule that corresponds to good and bad test cases that these rules describe

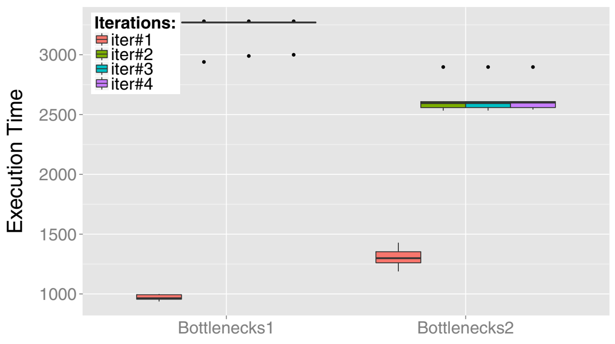


Fig. 5 Average execution times (in second) for different groups of injected bottlenecks (i.e.,bottlenecks#1 and bottlenecks#2), where $n_p = 10$ and $n_u = 5$. *Different colors* represent different iterations. The *central box* represents the values from the lower to upper quartile (i.e., 25 to 75 percentile). The *middle line* represents the median

low deductible goes together with very high coverage limit, and it is combined with the owner of the condo taking prescribed medications, and with the condo having fewer than two residents. All these inputs point to situations that are considered higher risk insurance policies. These classes of input values trigger more computations that lead to significantly higher workloads.

For JPetSore, rules `J-1` and `J-2` describe inputs as the number of occurrences of URLs in transactions, where URLs are shown using descriptive names (e.g., "Checkout" for the URL that enables customers to check out their shopping carts). It is important that rules for both applications are input-specific. While we do not expect that rules learned for one system would apply to a completely different system, training a new set of rules using the same algorithm should deliver similar benefits.

5.2 Research Question 2

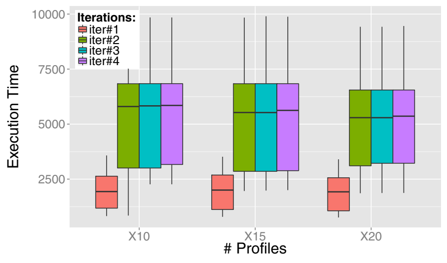
Finding Test Inputs for Increased Workloads Recall that we injected two different groups of artificial bottlenecks to investigate how FOREPSOT identifies different injected

Table 4 Detailed execution times (in seconds) for different configurations of the independent variables for JPetStore

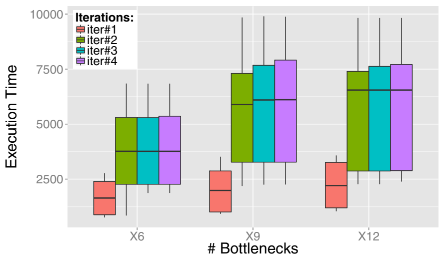
	6 Bottlenecks				9 Bottlenecks				12 Bottlenecks			
n_u-n_p	iter #1	iter #2	iter #3	iter #4	iter #1	iter #2	iter #3	iter #4	iter #1	iter #2	iter #3	iter #4
5–10	859	1983	2269	2270	970	3206	3216	3218	1180	2759	2790	2815
5–15	860	1990	2013	2024	963	2369	2420	2446	1138	3001	3010	3022
5–20	809	2000	2024	2031	975	3252	3253	3252	1114	2462	2473	2553
10–10	1651	3758	3787	3857	1971	6093	6108	6118	2120	6551	6551	6551
10–15	1694	4075	4075	4084	2002	6157	6175	6201	2231	6042	6054	6078
10–20	1623	3391	3488	3659	1922	6094	6118	6126	2278	5898	5962	6006
15–10	2510	6837	6838	6837	3089	7973	9324	9334	3484	8167	8180	8221
15–15	2491	6070	6077	6088	3062	9828	9860	9852	3360	7487	7684	7750
15–20	2530	5314	5380	5448	2920	7468	7676	7810	3267	8199	8220	8326

bottlenecks. The bottlenecks#1 refer to the methods that have same artificial delay, and the bottlenecks#2 refer to the methods that have different artificial delays. The results with the same configurations of FOREPOST but different groups of bottlenecks are shown in Fig. 5. As the results show, the execution times for bottlenecks#1 are generally larger than the execution times for bottlenecks#2. The reason is that different delays in these two groups of bottlenecks lead to different AUT behaviors. Some injected bottlenecks in bottlenecks#1 have longer delays, leading to more computationally intensive executions as compared to the bottlenecks#2. These empirical results confirm the conjecture that different types of bottlenecks affect the performance of the FOREPOST engine in finding test inputs for increased workloads.

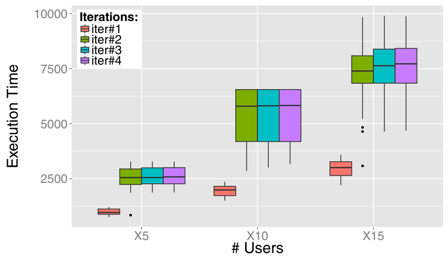
The results for the average execution times of the sensitivity analysis are shown in detail in Table 4. In the table, Column 1 presents the different settings for the number of users (i.e., n_u) and the number of profiles (i.e., n_p). Columns 2–5 present the average time (across 5 runs) for each iteration when injecting six bottlenecks. Similarly, Columns 6–9 and Columns 10–13 present the average execution time for each iteration when injecting nine and twelve bottlenecks, respectively. To further investigate the impacts of various independent variables on the execution time, we control the value of each independent variable, and present the corresponding results in Fig. 6. In this figure, each sub-figure represents the execution time information when we control the value of each independent variable (e.g., number of profiles, bottlenecks, and users). In each sub-figure, the x-axis presents the values for the controlled independent variable, the y-axis presents the execution time, and boxplots in different color represent different iteration. Note that the boxplots present the median (line



(a) Average execution times when controlling the number of profiles.



(b) Average execution times when controlling the number of bottlenecks.



(c) Average execution times when controlling the number of users

Fig. 6 Average execution times (in second) when controlling different independent variables. *Different colors* represent different iterations. The *central box* represents the values from the lower to upper quartile (i.e., 25 to 75 percentile). The *middle line* represents the median

in the box), upper/lower quartile, and 90th/10th percentile values. From the figure, we can infer the following observations:

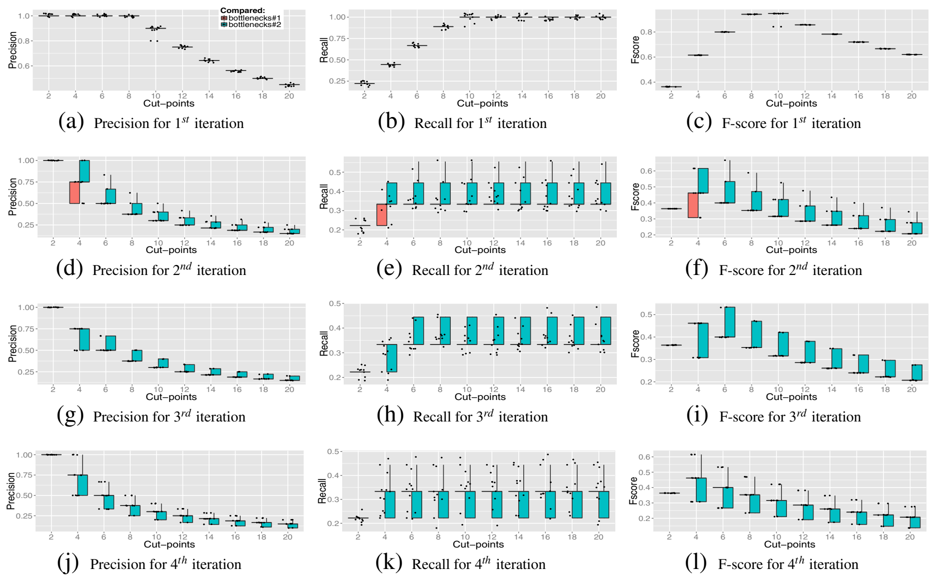
First, across all the sub-figures, we can find that after the 1^{st} iteration, the execution time increases dramatically, implying that FOREPOST can find inputs that steer applications towards computationally intensive executions. But after the 2^{nd} iteration, the execution times increase slightly, implying that the learnt rules converge to some stable states. The reason is that the inputs are selected randomly in the 1^{st} iteration. From the 2^{nd} iteration, a number of interesting rules are inferred to cover the hot paths in the system under test. Thus the execution time increases dramatically. However, from the 2^{nd} iteration, the majority of the hot paths have been covered by FOREPOST, making the execution time relatively stable after the 2^{nd} iteration. This also implies that different numbers of iterations do not significantly help identify different behaviors in order to find test input data that steers applications towards more computationally intensive executions.

When controlling the independent variable of the *number of profiles* (see Fig. 6a), we find that the execution time does not change much across different number of profiles. We also observe an interesting finding that when the number of profiles per iteration increases, the execution time for various other settings tends to be more stable. For example, the boxplot for using 20 profiles is more stable than that for ten profiles. The reason is that after collecting more profiles, the machine learning results can be more accurate in guiding meaningful rule generation. Therefore, the empirical results when controlling the number of profiles demonstrate that FOREPOST becomes more stable when using more profiles.

When controlling the independent variable of the *number of bottlenecks* (see Fig. 6b), we find that the execution time increases gradually and the performance has wider range when injecting more bottlenecks. One possible reason is that more injected bottlenecks may incur more bottlenecks to be actually executed, leading to longer execution time. Moreover, the interaction of different bottlenecks can make the AUT's performance rather unstable, which enlarges the possible range for the application execution time.

When controlling the independent variable of the *number of users* (see Fig. 6c), we find that the execution time increases linearly when simulating more users. For example, when using five users, the average execution time for the inputs selected by FOREPOST (e.g., the 4^{th} iteration) is around 2500 seconds, while it is approximately 5500 and 7500 seconds when the number of users increases to ten and 15 respectively. Obviously, when the number of users increases, it would take more time to execute the URL requests sending by the increased users. We can also find that the execution time becomes more unstable when simulating ten users as compared to five users. The possible reason is that when increasing the number of users, the problems of multi-threading, synchronization, etc., may arise, causing the studied application to have performance bottlenecks of wider range.

Identifying Bottlenecks The results of experiments with different groups of artificial bottlenecks (i.e., bottlenecks#1 and bottlenecks#2) are shown in Fig. 7. From the figures, we observe that the results of bottlenecks#2 vary greatly as compared to the results of bottlenecks#1. One possible reason is that the injected bottlenecks with different delays (i.e., bottlenecks#2) make the AUT performance vary, thus FOREPOST may converge to the different executions for uncovering different performance bottlenecks, leading to unstable results. On the contrary, FOREPOST always converge to some stable states when injecting the same performance bottlenecks (i.e., bottlenecks#1). Given our empirical results, we conclude that the length of the delays in bottlenecks, given the delay ranges we experimented with, impacts the performance of FOREPOST.



(a) Precision for 1st iteration (b) Recall for 1st iteration (c) F-score for 1st iteration

(d) Precision for 2nd iteration (e) Recall for 2nd iteration (f) F-score for 2nd iteration

(g) Precision for 3rd iteration (h) Recall for 3rd iteration (i) F-score for 3rd iteration

(j) Precision for 4th iteration (k) Recall for 4th iteration (l) F-score for 4th iteration

Fig. 7 Comparison between FOREPOST using uniform or different bottlenecks for JPetStore. The *red boxplots* refer to bottlenecks#1. The *green boxplots* refer to bottlenecks#2

For the sensitivity analysis, let's look into the results where $n_p = 10$, and $n_u = 5$. The precision, recall and F-score are shown in Tables 5, 6, and 7. Note that due to the space limitation, we put the detailed results for all the other settings in the Appendix section. The precision is measured as the percentage of the artificial bottlenecks returned in the top methods. The recall is measured as the ratio of the artificial bottlenecks within the top methods to all the injected bottlenecks. The F-score is calculated based on the precision and recall (i.e., $F - score = \frac{2*precision*recall}{precision+recall}$). To calculate those metrics, we used a cut point to separate methods in the ranking list into two parts, and we only considered the ranks of

Table 5 Precision for FOREPOST when $n_u = 5$ and $n_p = 10$

Cut	6 Bottlenecks				9 Bottlenecks				12 Bottlenecks			
points	iter #1	iter #2	iter #3	iter #4	iter #1	iter #2	iter #3	iter #4	iter #1	iter #2	iter #3	iter #4
2	100.00	100.00	90.00	90.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	90.00
4	95.00	65.00	55.00	60.00	100.00	65.00	70.00	75.00	100.00	75.00	80.00	75.00
6	90.00	46.67	40.00	40.00	100.00	50.00	50.00	50.00	100.00	50.00	60.00	56.67
8	72.50	35.00	30.00	30.00	100.00	37.50	37.50	40.00	100.00	40.00	45.00	45.00
10	60.00	28.00	24.00	24.00	88.00	32.00	30.00	32.00	100.00	34.00	36.00	36.00
12	50.00	26.67	20.00	20.00	75.00	26.67	25.00	26.67	100.00	28.33	30.00	30.00
14	42.86	22.86	17.14	17.14	64.29	22.86	21.43	22.86	85.71	24.29	27.14	25.71
16	37.50	20.00	15.00	15.00	56.25	20.00	18.75	20.00	75.00	21.25	23.75	22.50
18	33.33	17.78	13.33	13.33	50.00	17.78	16.67	17.78	66.67	20.00	21.11	20.00
20	30.00	16.00	12.00	12.00	45.00	16.00	15.00	16.00	60.00	18.00	19.00	18.00

Table 6 Recall for FOREPOST when $n_u = 5$ and $n_p = 10$

Cut	6 Bottlenecks				9 Bottlenecks				12 Bottlenecks			
points	iter #1	iter #2	iter #3	iter #4	iter #1	iter #2	iter #3	iter #4	iter #1	iter #2	iter #3	iter #4
2	33.33	33.33	30.00	30.00	22.22	22.22	22.22	22.22	16.67	16.67	16.67	15.00
4	63.33	43.33	36.67	40.00	44.44	28.89	31.11	33.33	33.33	25.00	26.67	25.00
6	90.00	46.67	40.00	40.00	66.67	33.33	33.33	33.33	50.00	25.00	30.00	28.33
8	96.67	46.67	40.00	40.00	88.89	33.33	33.33	35.56	66.67	26.67	30.00	30.00
10	100.00	46.67	40.00	40.00	97.78	35.56	33.33	35.56	83.33	28.33	30.00	30.00
12	100.00	53.33	40.00	40.00	100.00	35.56	33.33	35.56	100.00	28.33	30.00	30.00
14	100.00	53.33	40.00	40.00	100.00	35.56	33.33	35.56	100.00	28.33	31.67	30.00
16	100.00	53.33	40.00	40.00	100.00	35.56	33.33	35.56	100.00	28.33	31.67	30.00
18	100.00	53.33	40.00	40.00	100.00	35.56	33.33	35.56	100.00	30.00	31.67	30.00
20	100.00	53.33	40.00	40.00	100.00	35.56	33.33	35.56	100.00	30.00	31.67	30.00

methods whose positions are higher than the cut point. For example, if we set the cut point as ten, it means we only consider the ranks of methods which are in top ten and the methods outside the top ten are ignored. In each of the three tables, Column 1 lists the different cut points used, Columns 2–5/6–9/10–13 list the corresponding metric results for FOREPOST when injecting 6/9/12 bottlenecks. According to the three tables, we have the following observations:

First, the best cut point depends on the number of potential bottlenecks in the application under test as well as the number of iterations used in FOREPOST. For example, as shown in Table 7 ($n_u = 5$ and $n_p = 10$), when controlling the number of bottlenecks to be six, the cut point with the highest F-score value is six after 1^{st} iteration, and four after 2^{nd} iterations. Similarly, when controlling the number of iterations to be four, the cut point with the highest F-score value is four when injecting six bottlenecks, and six when injecting twelve bottlenecks.

Table 7 F-score for FOREPOST when $n_u = 5$ and $n_p = 10$

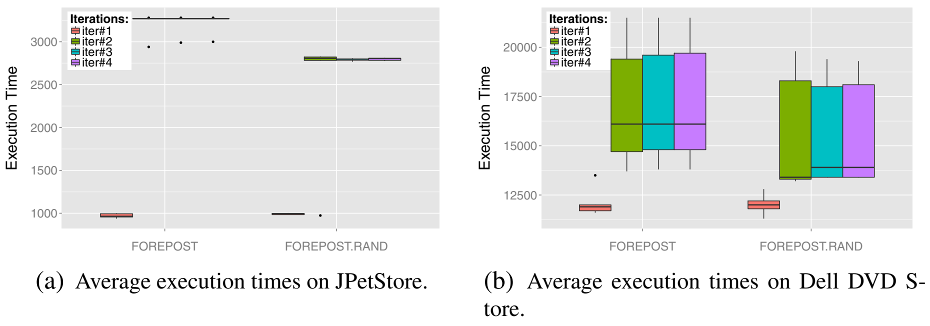
Cut	6 Bottlenecks				9 Bottlenecks				12 Bottlenecks			
points	iter #1	iter #2	iter #3	iter #4	iter #1	iter #2	iter #3	iter #4	iter #1	iter #2	iter #3	iter #4
2	50.00	50.00	45.00	45.00	36.36	36.36	36.36	36.36	28.57	28.57	28.57	25.71
4	76.00	52.00	44.00	48.00	61.54	40.00	43.08	46.15	50.00	37.50	40.00	37.50
6	90.00	46.67	40.00	40.00	80.00	40.00	40.00	40.00	66.67	33.33	40.00	37.78
8	82.86	40.00	34.29	34.29	94.12	35.29	35.29	37.65	80.00	32.00	36.00	36.00
10	75.00	35.00	30.00	30.00	92.63	33.68	31.58	33.68	90.91	30.91	32.73	32.73
12	66.67	35.56	26.67	26.67	85.71	30.48	28.57	30.48	100.00	28.33	30.00	30.00
14	60.00	32.00	24.00	24.00	78.26	27.83	26.09	27.83	92.31	26.15	29.23	27.69
16	54.55	29.09	21.82	21.82	72.00	25.60	24.00	25.60	85.71	24.29	27.14	25.71
18	50.00	26.67	20.00	20.00	66.67	23.70	22.22	23.70	80.00	24.00	25.33	24.00
20	46.15	24.62	18.46	18.46	62.07	22.07	20.69	22.07	75.00	22.50	23.75	22.50

Second, as the number of iterations increases, FOREPOST tends to miss some injected performance bottlenecks. For example, when using six as the cut point value where $n_b = 9$, the random inputs (i.e., the 1^{st} iteration) have a F-score of 80.00, but the selected inputs (e.g., the 2^{nd} iteration) only has a F-score of 40.00. A possible explanation is that FOREPOST runs a much more manageable and focused subset of input data after generating rules, which means the size of this subset is much smaller and easier to test, but it can lead to intense computations faster than other subsets. However, since the input data is limited by the rules, the domain would become smaller if rules are only associated with a small subset of methods. Although some methods have a high probability to be associated with the bottlenecks, FOREPOST still does not list them since they are not invoked in the execution traces. For example, before learning rules, there are 386 methods invoked in JPetStore. But only 208 methods are invoked after learning rules, which means the learned rules focused on a subset of input data, therefor less methods are invoked during the execution. Meanwhile, as observed from Fig. 6, the average execution times of selected inputs are quite higher than the average execution times of random inputs (i.e., inputs in the 1^{st} iteration), which implies that FOREPOST finds the subset of input data that lead to intense computations in a short time. So FOREPOST identifies computationally more expensive execution paths as compared to random performance testing at the expense of lower precision.

Our current implementation of FOREPOST only considers cumulative execution time of each method in the ICA algorithm.The other metrics, such as the number of invocations and the amount of data transited between methods and database, may affect the method weights when using ICA algorithm to identify bottlenecks. We leave this investigation as future work. For different applications, the FOREPOST behaves differently. It took around 24 hours to finish five runs on JPetStore with nine artificial bottlenecks, where the number of profiles (n_p) is ten, the number of iterations (n_i) is four and the number of users (n_u) is five. The time of the experiment related to the number of the bottlenecks and also their delay. Without any artificial bottlenecks, it took approximately five hours to finish five runs on JPetStore. Moreover, as the n_p, n_i, and n_u increase, the experiments become more time-consuming. It took around two months to finish all the experiments on sensitivity analysis for JPetStore. Currently the experiments in empirical study 2 are done on JPetStore only, since the experiments on Dell DVD Store are extremely time-consuming. The reason is that we injected artificial bottlenecks into the standard library for Dell DVD Store. These bottlenecks were invoked more frequently, thus, the elapsed time was substantially longer.

5.3 Research Question 3

Comparing FOREPOST and FOREPOST$_{RAND}$ in Finding Test Input for Increased Workloads The results for comparing FOREPOST and FOREPOST$_{RAND}$ are shown in Fig. 8. On JPetStore, the inputs selected by FOREPOST take around 3200 seconds after the 2^{nd} iteration, while inputs selected by FOREPOST$_{RAND}$ take around 2800 seconds. Although there is no significant difference after the 2^{nd} iteration, the average execution times of FOREPOST are always larger than those of FOREPOST$_{RAND}$. The results demonstrate that FOREPOST is more effective in finding test input data that steer applications towards more computationally intensive executions compared with FOREPOST$_{RAND}$. The conclusion is confirmed by the results for Dell DVD Store shown in Fig. 8b. Morever, the increase between random inputs and selected inputs in execution time on Dell DVD Store is smaller as compared to JPetStore since Dell DVD Store has relatively smaller number of combinations of inputs. Thus, even randomly selected inputs can cover significant part of

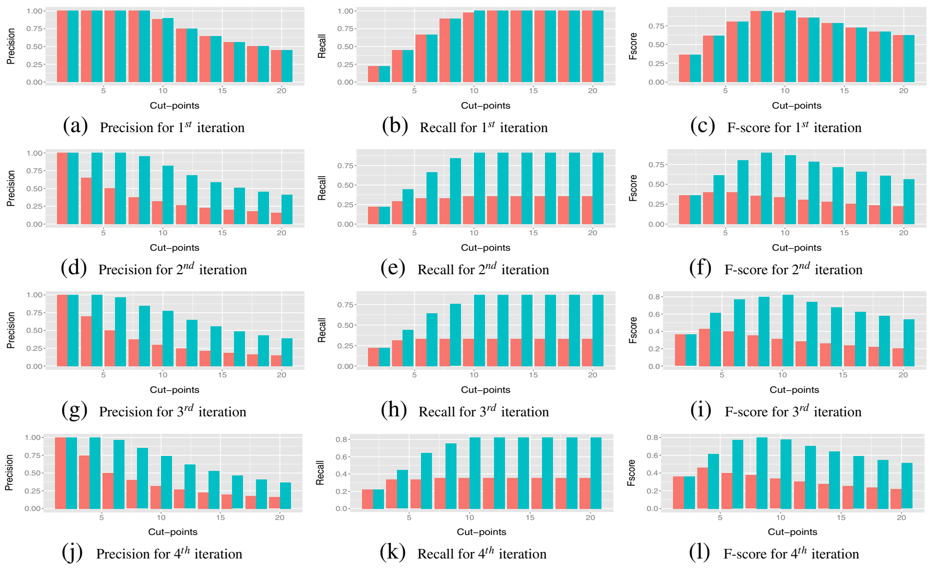


(a) Average execution times on JPetStore. (b) Average execution times on Dell DVD S-tore.

Fig. 8 Average execution times (in second) for FOREPOST and FORPOST$_{RAND}$, where $n_p = 10$ and $n_u = 5$. *Different colors* represent different iterations. The *central box* represents the values from the lower to upper quartile (i.e., 25 to 75 percentile). The *middle line* represents the median

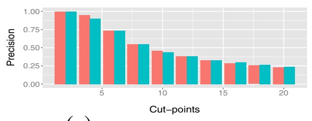
the computationally intensive executions. All in all, FOREPOST is more effective in finding inputs to cover computationally intensive executions, thus addressing RQ$_3$.

Comparing FOREPOST and FOREPOST$_{RAND}$ in Identifying Bottlenecks While FOREPOST finds artificial bottlenecks with lower F-score, FOREPOST$_{RAND}$ can identify almost all artificially injected bottlenecks, which is shown in Figs. 9 and 10. The figures show the precision, recall and F-score after each iterations on JPetStore and Dell DVD Store, respectively. We observe that the comparison results show similar trends on both applications. The precision, recall and F-score are quite similar after the 1st iteration since inputs in both FOREPOST and FOREPOST$_{RAND}$ are selected randomly. After the 2nd iteration,

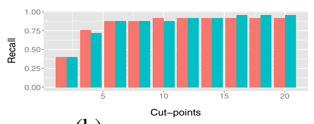


(a) Precision for 1st iteration (b) Recall for 1st iteration (c) F-score for 1st iteration
(d) Precision for 2nd iteration (e) Recall for 2nd iteration (f) F-score for 2nd iteration
(g) Precision for 3rd iteration (h) Recall for 3rd iteration (i) F-score for 3rd iteration
(j) Precision for 4th iteration (k) Recall for 4th iteration (l) F-score for 4th iteration

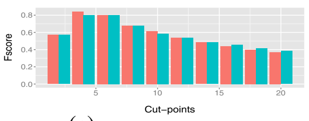
Fig. 9 Comparison between FOREPOST and FOREPOST$_{RAND}$ for JPetStore. Each bar represents the average precision/recall/f-score across five runs of the same setting. The *red bars* refer to FOREPOST. The *green bars* refer to FOREPOST$_{RAND}$



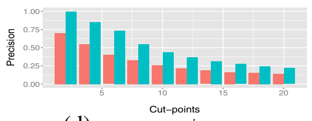
(a) Precision for 1st iteration



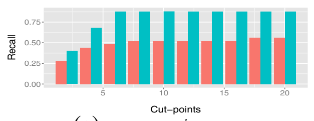
(b) Recall for 1st iteration



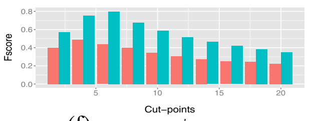
(c) F-score for 1st iteration



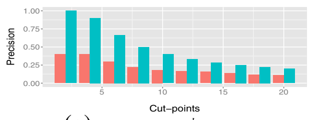
(d) Precision for 2nd iteration



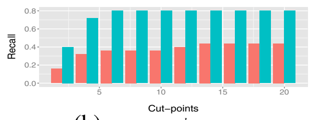
(e) Recall for 2nd iteration



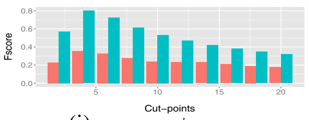
(f) F-score for 2nd iteration



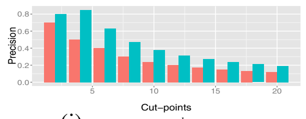
(g) Precision for 3rd iteration



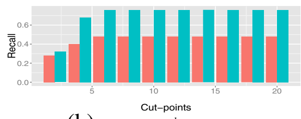
(h) Recall for 3rd iteration



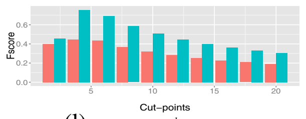
(i) F-score for 3rd iteration



(j) Precision for 4th iteration



(k) Recall for 4th iteration



(l) F-score for 4th iteration

Fig. 10 Comparison between FOREPOST and FOREPOST$_{RAND}$ for Dell DVD Store. Each bar represents the average precision/recall/f-score across five runs of the same setting. The *red bars* refer to FOREPOST. The *green bars* refer to FOREPOST$_{RAND}$

FOREPOST$_{RAND}$ have all clearly larger values as compared to FOREPOST. This implies that FOREPOST$_{RAND}$ outperforms FOREPOST in terms of accuracy. Among the reasons mentioned above, FOREPOST may miss to identify some bottlenecks since the input data is generated only based on rules which focus on traces that correspond to computationally intensive executions. Since FOREPOST$_{RAND}$ also involves random input data in addition to the specific input data based on the rules, it also covers other traces without losing accuracy. On the contrary, because FOREPOST focuses on more computationally expensive executions, it is hard to identify all the scenarios that lead to identifying specific performance bottlenecks. As our results demonstrated, software testers can choose either FOREPOST or FOREPOST$_{RAND}$ based on their goals: either identifying extreme bottlenecks by focusing on the more intensive executions or identifying as many bottlenecks as possible at a time but less intensive executions.

6 Threats to Validity

In this section we systematically review three different types of threats to validity to the studies reported in this paper: internal, external and construct validity.

6.1 Internal Validity

The first threat to internal validity relates to the fact that we injected artificial bottlenecks into the subject software systems. While we injected these bottlenecks randomly, there is a threat that some of the bottlenecks may not necessarily appear in the "natural" locations in program paths or where they are likely to appear in some real world scenarios. However, this particular design allowed us to evaluate FOREPOST in a controlled setting. Thus, we believe that we sufficiently minimized this threat, and our results are reliable.

In our implementation of profiling system, we used Probekit to inject probes into binary code for collecting execution traces, which affects the AUT performance behaviors. However, we only logged for some simple events like current system time for method entry and exit, and the overhead of Probekit was rather negligible. Thus, we believe that the overhead did not affect the results and current conclusions in our paper.

FOREPOST analyzes execution trace for each test case, and uses machine learning algorithm to extract rules for selecting test cases that lead to performance bottlenecks. It is possible that the rules converge to some local input space thus the selected test cases only steer executions to some specific paths. However, with more execution traces collected, it is possible to obtain more meaningful rules to select test cases uncovering more performance bottlenecks. Furthermore, it would be interesting to use different techniques like genetic algorithms to explore the input space. We leave this extension and rigorous comparison for the future work.

6.2 External Validity

The main external threat to our experimental design is that we experimented only with three subject AUTs. The results may vary for AUTs that have different logic or different architectures. Furthermore, we only have the authority to access the data and source code of Renters to conduct the experiments in empirical study 1, since it is a closed-source application that belongs to an insurance company. Thus, we did not perform the experiments on Renters in empirical studies 2 and 3. This threat makes it difficult to generalize the obtained results. There are many other different kinds of systems and different types of performance bottlenecks that can be tested in our experiments. However, since all the applications used are highly representative of enterprise-level applications and frequently used as benchmarks (Jiang et al. 2009; Foo et al. 2010) in performance testing research in software engineering, we suggest that our results are generalizable, at least in part, to a larger population of applications from these domains.

To evaluate the effectiveness of FOREPOST, we only compared FOREPOST with random testing and FOREPOST$_{RAND}$. This constitutes a threat, in that if we compare FOREPOST to other performance testing approaches, our results may compare differently. Thus, it may be difficult to derive general conclusions based solely on the comparisons made. However, the goal of FOREPOST is specific to find input data that leads to intensive computations which identify bottlenecks; and controlled experiments related to different performance testing approaches are difficult to compare. Comparing FOREPOST with FOREPOST$_{RAND}$ made the controlled experiments feasible and also reliable. We suggest that it minimized this threat effectively.

6.3 Construct Validity

In this paper, we used the execution time to measure the AUT performance and cluster execution traces, since the execution time is a representative performance metric and is widely used in performance testing area. The threat is that we did not consider other performance metrics. For example, memory leaks may lead to performance bottlenecks that arise over time, but memory usage is not taken into account in our current version. The methods that automatically scale or reconfigure themselves may also affect the AUT performance, introducing performance bottlenecks. However, our approach is not limited to use only the execution time as performance metric, and it can be extended using other types of performance metrics, like involving different metrics in matrix x (Section 3.4). We leave this extension as future work.

7 Related Work

There are many approaches that aid in generating test cases for testing. Avritzer et al. proposed an approach that automatically generates test cases and extended it by applying it into a "performability model", which is used to track the resource failures (Avritzer et al. 2011). Partition testing is a set of strategies that divides the program's input domain into subdomains (subsets) from which test cases can be derived to cover each subset at least once (Ammann and Offutt 2008, pages 150–152). Closely related is the work by Dickinson et al. (2001), which uses clustering analysis execution profiles to find failures among the executions induced by a set of potential test cases. Although their work and ours used clustering techniques, our work differs in that we cluster the execution profiles based on the length of the execution time and number of methods that have been invoked, and we target performance bottlenecks instead of functional errors.

Load testing is used to determine the AUT performance behaviors under specific workloads. Bayan et al. proposed an approach that uses a PID controller to automatically drive the test cases for achieving a pre-specified level of stress/load for a specific resource, such as response time (Bayan and Cangussu 2008). Another related work by Barna et al. proposed an autonomic framework to explore workload space and identify the points that cause the worst case behavior (Barna et al. 2011). It contains a feedback loop that generates workloads, monitors the software system, analyzes the effects of each workload and plans the new workloads. However, this work focuses on the effects of workloads (i.e., number of requests). Thus, it does not consider the effects of different types of requests (e.g., browse, buy, pay) in web applications. The Menasce's work discusses three important activities, load testing, benchmarking, and application performance management, on web-based applications, and provides a performance models that illustrates the relationship between workload and throughput/response time for improving load testing (Menascé 2002). Briand et al. proposed an approach that uses genetic algorithms to find combinations of inputs that ensure that completion times of a specific task's executions are as close as possible to their deadlines (Briand et al. 2005). However, all these approaches do not point out the potential performance bottlenecks in the AUT. In contrast, FOREPOST explores input space and uses machine learning algorithms to identify the combinations of inputs (i.e., requests in web application) for finding performance bottlenecks.

Operational profile is commonly used in performance load testing, where a system can be tested more efficiently because the operations most frequently used are tested the most (Musa 1993). It is a quantitative characterization of how the software will be used, which indicates the occurrence probabilities of function calls and the distributions of parameter values. Avritzer et al. proposed an approach that uses operational profiles to improve performance testing, where an application-independent performance workload is designed for comparing the existing production with the proposed architecture (Avritzer and Weyuker 1996). In this approach, operational data are collected in the current production environment, and a synthetic workload is fabricated which has a profile close to the average profile compiled by the application in production for the selected operations. However, this work is not aimed at pinpointing specific methods leading to the different performance behaviors of the application.

Learning rules helps stakeholders to reconfigure distributed systems online to optimize for dynamically changing workloads (Wildstrom et al. 2007). This work is similar to FOREPOST in using the learning methodology to learn rules, from only low-level system statistics, which of a set of possible hardware configurations will lead to better performance under the current unknown workload. In contrast, FOREPOST uses feedback-directed adaptive performance test scripts to locate most computationally intensive execution profiles and bottlenecks.

There is a recent work that studied 109 real-world performance bugs and found the guidance to detect performance bugs (Jin et al. 2012). The study demonstrated the root causes of performance bugs, thus the efficiency rules should exist and could be collected from patches. Then, the extracted rules from real-world performance-bug patches are used to check performance bottlenecks. Compared with FOREPOST, these rules are extracted manually, and they are used to analyze software binary code (Jin et al. 2012). In contrast, FOREPOST extracts rules by using black-box software testing. Furthermore, the study by Zaman et al. (2011) compared performance bugs and the security bugs, and found that performance bugs fixes impacted more files and took more time, while security bugs were fixed and triaged faster, but reopened and tossed frequently, required more developers and were more complex overall (Zaman et al. 2011).

Another technique related to FOREPOST automatically classifies execution data, collected in the field, which comes from either passing or failing program runs (Haran et al. 2005). This technique attempts to learn a classification model to predict if an application run failed using execution data. Jovic et al. presented an approach, called *Lag Hunting*, that collects runtime information such as the stack samples, and analyzes this information to detect the latency bugs automatically (Jovic et al. 2011). Malik et al. developed an automated approach that ranked the subsystems that likely involved performance deviations by using the performance signatures (Malik et al. 2010). Subsequently they proposed and compared one supervised and three unsupervised approaches for detecting performance deviations automatically for the loading testing in large scale systems, with a smaller and manageable subset of performance counters (Malik et al. 2013). Moreover, Syer et al. recently combined performance counters and execution logs to detect memory-related issues automatically (Syer et al. 2013). On the other hand, FOREPOST learns rules that it uses to select test input data that steer applications towards computationally intensive runs to expose performance bottlenecks.

In the recent work, Zhang, Elbaum, and Dwyer generate performance test cases using dynamic symbolic execution (Zhang et al. 2011). Similar to FOREPOST, they used heuristics that guided the generation of test cases by determining paths of executions that can introduce higher workloads. Shen et al. proposed an approach that uses genetic algorithms to explore input space for finding the combinations of inputs likely to trigger performance bottlenecks. It treats determining inputs to reveal performance bottlenecks as a search and optimization problem, while FOREPOST tries to learn a precise model for AUT's performance behavior for performance testing (Shen et al. 2015). Zaparanuks and Hauswirth presented algorithmic profiler that identifies the ingredients of algorithms and their inputs, presented the execution cost by using the repetition tree, and provided the cost function that illustrates the relationship between the cost and the input size, which can be used to identify algorithms with higher algorithmic complexity (Zaparanuks and Hauswirth 2012). Unlike FOREPOST, white-box testing approach are used, thus requiring access to source code, while FOREPOST is a black-box approach. It is also unclear how the approach (Zhang et al. 2011) will scale to industrial applications with over 100KLOC. We view these approaches as complementary, where a hybrid technique may combine the benefits of both approaches in a gray-box performance testing. This is left for the future work.

8 Conclusion

We offer a novel solution for automatically finding performance bottlenecks in applications using black-box software testing. Our solution, FOREPOST, is an adaptive, feedback-directed learning testing system that learns rules from execution traces of applications. These rules are then used to automatically select test input data for performance testing. As compared with random testing, our solution can find more performance bottlenecks. We have implemented our solution and applied it to a nontrivial closed-source application at a major insurance company and to two open-source applications in a controlled experiment. The results demonstrate that performance bottlenecks were found automatically in all applications and were confirmed by experienced testers and developers. Moreover, we implemented a new version of the approach referred to as FOREPOST$_{RAND}$. We compared FOREPOST with FOREPOST$_{RAND}$ on two open-source applications and confirmed that while FOREPOST$_{RAND}$ can identify bottlenecks with higher precision, FOREPOST was able to find not only the subset of bottlenecks, but also the scenarios that lead to substantially more intense computations, which could potentially lead to more serious performance bottlenecks in certain situations. Our results demonstrate that testers can use FOREPOST$_{RAND}$ for initial performance testing to outline possible roots of performance bottlenecks and use FOREPOST for more focused search of scenarios that result in substantial delays in system execution.

Acknowledgments We are grateful to the anonymous ICSE'12 and EMSE journal reviewers for their relevant and useful comments and suggestions, which helped us to significantly improve an earlier version of this paper. We would like to thank Bogdan Dit and Kevin Moran for reading the paper and providing the feedback on early drafts. We would also like to thank Du Shen for his pertinent feedback on improving the current version of FOREPOST as well as pointing out areas for improvement. We also thank Chen Fu and Qin Xie for their contributions to the earlier version of this work. This work is supported by NSF CCF-0916139, NSF CCF-1017633, NSF CCF-1218129, NSF CCF-1525902, a major insurance company, and Accenture.

Appendix

Table 8 Six Injected bottlenecks in JPetStore, where the length of delay is measured in seconds

Method name	Method container	Delay
getProductId()	/domain/Product	0.10
getCategoryId()	/domain/Product	0.10
getItemListByProduct(String)	/persistence/sqlmapdao/ItemSqlMapDao	0.10
getProductListByCategory(String)	/persistence/sqlmapdao/ProductSqlMapDao	0.10
getUsername()	/presentation/AccountBean	0.10
getAccount()	/presentation/AccountBean	0.10

Table 9 Nine injected bottlenecks in JPetStore, where the length of delay is measured in seconds

Method name	Method container	Delay in bottlenecks#1	Delay in bottlenecks#2
setZip(String)	/domain/Account	0.10	0.05
getProductId()	/domain/Product	0.10	0.10
getCategoryId()	/domain/Product	0.10	0.15
getName()	/domain/Product	0.10	0.05
getItemList-ByProduct(String)	/persistence/sqlmapdao /ItemSqlMapDao	0.10	0.10
getProductList-ByCategory(String)	/persistence/sqlmapdao /ProductSqlMapDao	0.10	0.15
getUsername()	/presentation/AccountBean	0.10	0.05
setPassword(String)	/presentation/AccountBean	0.10	0.10
getAccount()	/presentation/AccountBean	0.10	0.15

Table 10 Twelve Injected bottlenecks in JPetStore, where the length of delay is measured in seconds

Method name	Method container	Delay
setZip(String)	/domain/Account	0.10
getLanguagePreference()	/domain/Account	0.10
getItemId()	/domain/Item	0.10
getProductId()	/domain/Product	0.10
getCategoryId()	/domain/Product	0.10
getName()	/domain/Product	0.10
getItemListByProduct(String)	/persistence/sqlmapdao/ItemSqlMapDao	0.10
isItemInStock()	/persistence/sqlmapdao/ItemSqlMapDao	0.10
getProductListByCategory(String)	/persistence/sqlmapdao/ProductSqlMapDao	0.10
getUsername()	/presentation/AccountBean	0.10
setPassword(String)	/presentation/AccountBean	0.10
getAccount()	/presentation/AccountBean	0.10

Table 11 Injected bottlenecks in Dell DVD Store and the standard jar file mysql-connector-java.jar, where the length of delay is measured in seconds

Method name	Method container	Delay
getBrowsetype()	/com/ds2/ConnectionManager	0.10
setNew_item_length(int)	/com/ds2/ConnectionManager	0.10
getBufLength()	/com/mysql/jdbc/Buffer	0.10
loadAuthenticationPlugins()	/com/mysql/jdbc/MysqlIO	0.10
checkForIntegerTruncation(int, byte[], int)	/com/mysql/jdbc/ResultSetImpl	0.10

Table 12 Ranks of bottlenecks for FOREPOST in JPetStore, where there are five original bottlenecks and nine artificial bottlenecks

		No injected bottleneck				Bottleneck#1			
Bottlenecks	Method name	iter #1	iter #2	iter #3	iter #4	iter #1	iter #2	iter #3	iter #4
Original	pushConnection()	4	1	2	1	29	8	37	16
bottlenecks	popConnection()	1	3	1	2	11	18	47	5
	getResults()	2	5	6	3	16	5	5	7
	getNullValue()	17	7	5	7	46	27	28	30
	executeQuery()	6	11	3	4	13	9	7	9
Injected	setZip(String)	380	N/A	N/A	N/A	8	N/A	N/A	N/A
bottlenecks	getProductId()	34	24	18	43	2	1	1	3
	getCategoryId()	253	N/A	N/A	N/A	4	N/A	N/A	N/A
	getName()	140	200	167	201	3	4	4	1
	getItemListByProduct(String)	188	N/A	N/A	N/A	7	N/A	N/A	N/A
	getProductListByCategory(String)	280	204	111	141	6	2	3	123
	getUsername()	142	N/A	N/A	N/A	9	N/A	N/A	N/A
	setPassword(String)	235	N/A	N/A	N/A	5	N/A	N/A	N/A
	getAccount()	11	N/A	N/A	N/A	1	3	2	2

The vertical columns of group "No injected bottleneck" shows the ranks when there is no artificial bottleneck injected. The vertical columns of group "Bottleneck#1" shows the ranks when there are nine artificial bottlenecks injected. The number of profiles is equal to 10

Table 13 Precision for FOREPOST when $n_u = 5$ and $n_p = 15$

Cut	6 Bottlenecks				9 Bottlenecks				12 Bottlenecks			
points	iter #1	iter #2	iter #3	iter #4	iter #1	iter #2	iter #3	iter #4	iter #1	iter #2	iter #3	iter #4
2	100.00	90.00	80.00	80.00	100.00	90.00	100.00	100.00	100.00	100.00	100.00	70.00
4	95.00	50.00	50.00	45.00	100.00	75.00	75.00	75.00	100.00	90.00	70.00	80.00
6	83.33	33.33	36.67	30.00	96.67	56.67	53.33	56.67	100.00	60.00	53.33	60.00
8	70.00	27.50	27.50	25.00	92.50	45.00	40.00	42.50	100.00	45.00	45.00	47.50
10	58.00	22.00	22.00	20.00	80.00	36.00	32.00	34.00	100.00	40.00	36.00	38.00
12	48.33	18.33	18.33	16.67	70.00	30.00	26.67	28.33	96.67	33.33	30.00	31.67
14	41.43	15.71	15.71	14.29	61.43	25.71	22.86	24.29	84.29	28.57	25.71	27.14
16	37.50	13.75	13.75	12.50	53.75	22.50	20.00	21.25	73.75	25.00	22.50	23.75
18	33.33	12.22	12.22	11.11	47.78	20.00	17.78	18.89	65.56	22.22	20.00	21.11
20	30.00	11.00	11.00	10.00	43.00	18.00	16.00	17.00	60.00	20.00	18.00	19.00

Table 14 Recall for FOREPOST when $n_u = 5$ and $n_p = 15$

Cut	6 Bottlenecks				9 Bottlenecks				12 Bottlenecks			
points	iter #1	iter #2	iter #3	iter #4	iter #1	iter #2	iter #3	iter #4	iter #1	iter #2	iter #3	iter #4
2	33.33	30.00	26.67	26.67	22.22	20.00	22.22	22.22	16.67	16.67	16.67	11.67
4	63.33	33.33	33.33	30.00	44.44	33.33	33.33	33.33	33.33	30.00	23.33	26.67
6	83.33	33.33	36.67	30.00	64.44	37.78	35.56	37.78	50.00	30.00	26.67	30.00
8	93.33	36.67	36.67	33.33	82.22	40.00	35.56	37.78	66.67	30.00	30.00	31.67
10	96.67	36.67	36.67	33.33	88.89	40.00	35.56	37.78	83.33	33.33	30.00	31.67
12	96.67	36.67	36.67	33.33	93.33	40.00	35.56	37.78	96.67	33.33	30.00	31.67
14	96.67	36.67	36.67	33.33	95.56	40.00	35.56	37.78	98.33	33.33	30.00	31.67
16	100.00	36.67	36.67	33.33	95.56	40.00	35.56	37.78	98.33	33.33	30.00	31.67
18	100.00	36.67	36.67	33.33	95.56	40.00	35.56	37.78	98.33	33.33	30.00	31.67
20	100.00	36.67	36.67	33.33	95.56	40.00	35.56	37.78	100.00	33.33	30.00	31.67

Table 15 F-score for FOREPOST when $n_u = 5$ and $n_p = 15$

Cut	6 Bottlenecks				9 Bottlenecks				12 Bottlenecks			
points	iter #1	iter #2	iter #3	iter #4	iter #1	iter #2	iter #3	iter #4	iter #1	iter #2	iter #3	iter #4
2	50.00	45.00	40.00	40.00	36.36	32.73	36.36	36.36	28.57	28.57	28.57	20.00
4	76.00	40.00	40.00	36.00	61.54	46.15	46.15	46.15	50.00	45.00	35.00	40.00
6	83.33	33.33	36.67	30.00	77.33	45.33	42.67	45.33	66.67	40.00	35.56	40.00
8	80.00	31.43	31.43	28.57	87.06	42.35	37.65	40.00	80.00	36.00	36.00	38.00
10	72.50	27.50	27.50	25.00	84.21	37.89	33.68	35.79	90.91	36.36	32.73	34.55
12	64.44	24.44	24.44	22.22	80.00	34.29	30.48	32.38	96.67	33.33	30.00	31.67
14	58.00	22.00	22.00	20.00	74.78	31.30	27.83	29.57	90.77	30.77	27.69	29.23
16	54.55	20.00	20.00	18.18	68.80	28.80	25.60	27.20	84.29	28.57	25.71	27.14
18	50.00	18.33	18.33	16.67	63.70	26.67	23.70	25.19	78.67	26.67	24.00	25.33
20	46.15	16.92	16.92	15.38	59.31	24.83	22.07	23.45	75.00	25.00	22.50	23.75

Table 16 Precision for FOREPOST when $n_u = 5$ and $n_p = 20$

Cut	6 Bottlenecks				9 Bottlenecks				12 Bottlenecks			
points	iter #1	iter #2	iter #3	iter #4	iter #1	iter #2	iter #3	iter #4	iter #1	iter #2	iter #3	iter #4
2	100.00	100.00	100.00	90.00	100.00	100.00	100.00	90.00	100.00	100.00	100.00	100.00
4	100.00	55.00	55.00	50.00	100.00	80.00	70.00	70.00	100.00	70.00	80.00	85.00
6	90.00	36.67	40.00	40.00	100.00	53.33	53.33	50.00	100.00	56.67	60.00	56.67

Table 16 (continued)

Cut	6 Bottlenecks				9 Bottlenecks				12 Bottlenecks			
points	iter #1	iter #2	iter #3	iter #4	iter #1	iter #2	iter #3	iter #4	iter #1	iter #2	iter #3	iter #4
8	72.50	30.00	30.00	30.00	90.00	40.00	40.00	40.00	100.00	45.00	47.50	45.00
10	58.00	24.00	24.00	24.00	78.00	32.00	32.00	32.00	100.00	38.00	38.00	36.00
12	48.33	20.00	20.00	20.00	65.00	26.67	26.67	26.67	91.67	33.33	31.67	30.00
14	41.43	17.14	17.14	17.14	58.57	22.86	22.86	22.86	80.00	28.57	27.14	25.71
16	37.50	15.00	15.00	15.00	52.50	20.00	20.00	20.00	70.00	25.00	23.75	22.50
18	33.33	13.33	13.33	13.33	46.67	17.78	17.78	17.78	63.33	22.22	21.11	20.00
20	30.00	12.00	12.00	12.00	43.00	16.00	16.00	16.00	58.00	20.00	19.00	18.00

Table 17 Recall for FOREPOST when $n_u = 5$ and $n_p = 20$

Cut	6 Bottlenecks				9 Bottlenecks				12 Bottlenecks			
points	iter #1	iter #2	iter #3	iter #4	iter #1	iter #2	iter #3	iter #4	iter #1	iter #2	iter #3	iter #4
2	33.33	33.33	33.33	30.00	22.22	22.22	22.22	20.00	16.67	16.67	16.67	16.67
4	66.67	36.67	36.67	33.33	44.44	35.56	31.11	31.11	33.33	23.33	26.67	28.33
6	90.00	36.67	40.00	40.00	66.67	35.56	35.56	33.33	50.00	28.33	30.00	28.33
8	96.67	40.00	40.00	40.00	80.00	35.56	35.56	35.56	66.67	30.00	31.67	30.00
10	96.67	40.00	40.00	40.00	86.67	35.56	35.56	35.56	83.33	31.67	31.67	30.00
12	96.67	40.00	40.00	40.00	86.67	35.56	35.56	35.56	91.67	33.33	31.67	30.00
14	96.67	40.00	40.00	40.00	91.11	35.56	35.56	35.56	93.33	33.33	31.67	30.00
16	100.00	40.00	40.00	40.00	93.33	35.56	35.56	35.56	93.33	33.33	31.67	30.00
18	100.00	40.00	40.00	40.00	93.33	35.56	35.56	35.56	95.00	33.33	31.67	30.00
20	100.00	40.00	40.00	40.00	95.56	35.56	35.56	35.56	96.67	33.33	31.67	30.00

Table 18 F-score for FOREPOST when $n_u = 5$ and $n_p = 20$

Cut	6 Bottlenecks				9 Bottlenecks				12 Bottlenecks			
points	iter #1	iter #2	iter #3	iter #4	iter #1	iter #2	iter #3	iter #4	iter #1	iter #2	iter #3	iter #4
2	50.00	50.00	50.00	45.00	36.36	36.36	36.36	32.73	28.57	28.57	28.57	28.57
4	80.00	44.00	44.00	40.00	61.54	49.23	43.08	43.08	50.00	35.00	40.00	42.50
6	90.00	36.67	40.00	40.00	80.00	42.67	42.67	40.00	66.67	37.78	40.00	37.78
8	82.86	34.29	34.29	34.29	84.71	37.65	37.65	37.65	80.00	36.00	38.00	36.00
10	72.50	30.00	30.00	30.00	82.11	33.68	33.68	33.68	90.91	34.55	34.55	32.73
12	64.44	26.67	26.67	26.67	74.29	30.48	30.48	30.48	91.67	33.33	31.67	30.00
14	58.00	24.00	24.00	24.00	71.30	27.83	27.83	27.83	86.15	30.77	29.23	27.69
16	54.55	21.82	21.82	21.82	67.20	25.60	25.60	25.60	80.00	28.57	27.14	25.71
18	50.00	20.00	20.00	20.00	62.22	23.70	23.70	23.70	76.00	26.67	25.33	24.00
20	46.15	18.46	18.46	18.46	59.31	22.07	22.07	22.07	72.50	25.00	23.75	22.50

Table 19 Precision for FOREPOST when $n_u = 10$ and $n_p = 10$

Cut	6 Bottlenecks				9 Bottlenecks				12 Bottlenecks			
points	iter #1	iter #2	iter #3	iter #4	iter #1	iter #2	iter #3	iter #4	iter #1	iter #2	iter #3	iter #4
2	100.00	80.00	90.00	80.00	100.00	100.00	100.00	90.00	100.00	100.00	100.00	100.00
4	100.00	40.00	50.00	40.00	100.00	75.00	75.00	70.00	100.00	75.00	65.00	75.00
6	90.00	30.00	33.33	26.67	100.00	53.33	50.00	50.00	100.00	56.67	46.67	56.67
8	70.00	25.00	25.00	20.00	97.50	40.00	40.00	37.50	100.00	42.50	37.50	45.00
10	58.00	22.00	22.00	20.00	86.00	32.00	32.00	30.00	100.00	34.00	30.00	36.00
12	48.33	18.33	18.33	16.67	73.33	26.67	26.67	25.00	100.00	28.33	25.00	30.00
14	41.43	15.71	15.71	14.29	64.29	22.86	22.86	21.43	85.71	24.29	21.43	25.71
16	36.25	13.75	13.75	12.50	56.25	20.00	20.00	18.75	75.00	21.25	18.75	22.50
18	32.22	12.22	12.22	11.11	50.00	17.78	17.78	16.67	66.67	18.89	16.67	20.00
20	29.00	11.00	11.00	10.00	45.00	16.00	16.00	15.00	60.00	18.00	15.00	18.00

Table 20 Recall for FOREPOST when $n_u = 10$ and $n_p = 10$

Cut	6 Bottlenecks				9 Bottlenecks				12 Bottlenecks			
points	iter #1	iter #2	iter #3	iter #4	iter #1	iter #2	iter #3	iter #4	iter #1	iter #2	iter #3	iter #4
2	33.33	26.67	30.00	26.67	22.22	22.22	22.22	20.00	16.67	16.67	16.67	16.67
4	66.67	26.67	33.33	26.67	44.44	33.33	33.33	31.11	33.33	25.00	21.67	25.00
6	90.00	30.00	33.33	26.67	66.67	35.56	33.33	33.33	50.00	28.33	23.33	28.33
8	93.33	33.33	33.33	26.67	86.67	35.56	35.56	33.33	66.67	28.33	25.00	30.00
10	96.67	36.67	36.67	33.33	95.56	35.56	35.56	33.33	83.33	28.33	25.00	30.00
12	96.67	36.67	36.67	33.33	97.78	35.56	35.56	33.33	100.00	28.33	25.00	30.00
14	96.67	36.67	36.67	33.33	100.00	35.56	35.56	33.33	100.00	28.33	25.00	30.00
16	96.67	36.67	36.67	33.33	100.00	35.56	35.56	33.33	100.00	28.33	25.00	30.00
18	96.67	36.67	36.67	33.33	100.00	35.56	35.56	33.33	100.00	28.33	25.00	30.00
20	96.67	36.67	36.67	33.33	100.00	35.56	35.56	33.33	100.00	30.00	25.00	30.00

Table 21 F-score for FOREPOST when $n_u = 10$ and $n_p = 10$

Cut	6 Bottlenecks				9 Bottlenecks				12 Bottlenecks			
points	iter #1	iter #2	iter #3	iter #4	iter #1	iter #2	iter #3	iter #4	iter #1	iter #2	iter #3	iter #4
2	50.00	40.00	45.00	40.00	36.36	36.36	36.36	32.73	28.57	28.57	28.57	28.57
4	80.00	32.00	40.00	32.00	61.54	46.15	46.15	43.08	50.00	37.50	32.50	37.50
6	90.00	30.00	33.33	26.67	80.00	42.67	40.00	40.00	66.67	37.78	31.11	37.78
8	80.00	28.57	28.57	22.86	91.76	37.65	37.65	35.29	80.00	34.00	30.00	36.00
10	72.50	27.50	27.50	25.00	90.53	33.68	33.68	31.58	90.91	30.91	27.27	32.73

Table 21 (continued)

Cut	6 Bottlenecks				9 Bottlenecks				12 Bottlenecks			
points	iter #1	iter #2	iter #3	iter #4	iter #1	iter #2	iter #3	iter #4	iter #1	iter #2	iter #3	iter #4
12	64.44	24.44	24.44	22.22	83.81	30.48	30.48	28.57	100.00	28.33	25.00	30.00
14	58.00	22.00	22.00	20.00	78.26	27.83	27.83	26.09	92.31	26.15	23.08	27.69
16	52.73	20.00	20.00	18.18	72.00	25.60	25.60	24.00	85.71	24.29	21.43	25.71
18	48.33	18.33	18.33	16.67	66.67	23.70	23.70	22.22	80.00	22.67	20.00	24.00
20	44.62	16.92	16.92	15.38	62.07	22.07	22.07	20.69	75.00	22.50	18.75	22.50

Table 22 Precision for FOREPOST when $n_u = 10$ and $n_p = 15$

Cut	6 Bottlenecks				9 Bottlenecks				12 Bottlenecks			
points	iter #1	iter #2	iter #3	iter #4	iter #1	iter #2	iter #3	iter #4	iter #1	iter #2	iter #3	iter #4
2	100.00	100.00	90.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00
4	100.00	60.00	50.00	55.00	100.00	70.00	80.00	80.00	100.00	70.00	75.00	65.00
6	90.00	40.00	33.33	36.67	100.00	53.33	53.33	53.33	100.00	53.33	56.67	50.00
8	67.50	30.00	27.50	30.00	97.50	42.50	40.00	42.50	100.00	42.50	42.50	37.50
10	56.00	24.00	24.00	24.00	84.00	34.00	32.00	34.00	98.00	34.00	34.00	30.00
12	48.33	20.00	20.00	20.00	70.00	28.33	26.67	28.33	91.67	28.33	28.33	25.00
14	41.43	17.14	17.14	17.14	60.00	24.29	22.86	24.29	82.86	25.71	24.29	21.43
16	36.25	15.00	15.00	15.00	52.50	21.25	20.00	21.25	73.75	22.50	21.25	18.75
18	32.22	13.33	13.33	13.33	46.67	18.89	17.78	18.89	66.67	20.00	18.89	16.67
20	29.00	12.00	12.00	12.00	42.00	17.00	16.00	17.00	60.00	18.00	17.00	15.00

Table 23 Recall for FOREPOST when $n_u = 10$ and $n_p = 15$

Cut	6 Bottlenecks				9 Bottlenecks				12 Bottlenecks			
points	iter #1	iter #2	iter #3	iter #4	iter #1	iter #2	iter #3	iter #4	iter #1	iter #2	iter #3	iter #4
2	33.33	33.33	30.00	33.33	22.22	22.22	22.22	22.22	16.67	16.67	16.67	16.67
4	66.67	40.00	33.33	36.67	44.44	31.11	35.56	35.56	33.33	23.33	25.00	21.67
6	90.00	40.00	33.33	36.67	66.67	35.56	35.56	35.56	50.00	26.67	28.33	25.00
8	90.00	40.00	36.67	40.00	86.67	37.78	35.56	37.78	66.67	28.33	28.33	25.00
10	93.33	40.00	40.00	40.00	93.33	37.78	35.56	37.78	81.67	28.33	28.33	25.00
12	96.67	40.00	40.00	40.00	93.33	37.78	35.56	37.78	91.67	28.33	28.33	25.00
14	96.67	40.00	40.00	40.00	93.33	37.78	35.56	37.78	96.67	30.00	28.33	25.00
16	96.67	40.00	40.00	40.00	93.33	37.78	35.56	37.78	98.33	30.00	28.33	25.00
18	96.67	40.00	40.00	40.00	93.33	37.78	35.56	37.78	100.00	30.00	28.33	25.00
20	96.67	40.00	40.00	40.00	93.33	37.78	35.56	37.78	100.00	30.00	28.33	25.00

Table 24 F-score for FOREPOST when $n_u = 10$ and $n_p = 15$

Cut	6 Bottlenecks				9 Bottlenecks				12 Bottlenecks			
points	iter #1	iter #2	iter #3	iter #4	iter #1	iter #2	iter #3	iter #4	iter #1	iter #2	iter #3	iter #4
2	50.00	50.00	45.00	50.00	36.36	36.36	36.36	36.36	28.57	28.57	28.57	28.57
4	80.00	48.00	40.00	44.00	61.54	43.08	49.23	49.23	50.00	35.00	37.50	32.50
6	90.00	40.00	33.33	36.67	80.00	42.67	42.67	42.67	66.67	35.56	37.78	33.33
8	77.14	34.29	31.43	34.29	91.76	40.00	37.65	40.00	80.00	34.00	34.00	30.00
10	70.00	30.00	30.00	30.00	88.42	35.79	33.68	35.79	89.09	30.91	30.91	27.27
12	64.44	26.67	26.67	26.67	80.00	32.38	30.48	32.38	91.67	28.33	28.33	25.00
14	58.00	24.00	24.00	24.00	73.04	29.57	27.83	29.57	89.23	27.69	26.15	23.08
16	52.73	21.82	21.82	21.82	67.20	27.20	25.60	27.20	84.29	25.71	24.29	21.43
18	48.33	20.00	20.00	20.00	62.22	25.19	23.70	25.19	80.00	24.00	22.67	20.00
20	44.62	18.46	18.46	18.46	57.93	23.45	22.07	23.45	75.00	22.50	21.25	18.75

Table 25 Precision for FOREPOST when $n_u = 10$ and $n_p = 20$

Cut	6 Bottlenecks				9 Bottlenecks				12 Bottlenecks			
points	iter #1	iter #2	iter #3	iter #4	iter #1	iter #2	iter #3	iter #4	iter #1	iter #2	iter #3	iter #4
2	100.00	90.00	90.00	80.00	100.00	100.00	100.00	90.00	100.00	90.00	100.00	90.00
4	100.00	50.00	55.00	40.00	100.00	70.00	70.00	75.00	100.00	75.00	70.00	70.00
6	86.67	36.67	36.67	26.67	96.67	53.33	50.00	53.33	100.00	53.33	46.67	46.67
8	70.00	30.00	27.50	25.00	87.50	40.00	37.50	40.00	97.50	40.00	35.00	35.00
10	58.00	24.00	24.00	20.00	74.00	32.00	30.00	32.00	92.00	32.00	28.00	28.00
12	48.33	20.00	20.00	16.67	61.67	26.67	25.00	26.67	78.33	26.67	23.33	23.33
14	41.43	17.14	17.14	14.29	54.29	22.86	21.43	22.86	70.00	22.86	20.00	20.00
16	36.25	15.00	15.00	12.50	47.50	20.00	18.75	20.00	61.25	20.00	17.50	17.50
18	32.22	13.33	13.33	11.11	42.22	17.78	16.67	17.78	54.44	17.78	15.56	15.56
20	29.00	12.00	12.00	10.00	39.00	16.00	15.00	16.00	50.00	16.00	14.00	14.00

Table 26 Recall for FOREPOST when $n_u = 10$ and $n_p = 20$

Cut	6 Bottlenecks				9 Bottlenecks				12 Bottlenecks			
points	iter #1	iter #2	iter #3	iter #4	iter #1	iter #2	iter #3	iter #4	iter #1	iter #2	iter #3	iter #4
2	33.33	30.00	30.00	26.67	22.22	22.22	22.22	20.00	16.67	15.00	16.67	15.00
4	66.67	33.33	36.67	26.67	44.44	31.11	31.11	33.33	33.33	25.00	23.33	23.33
6	86.67	36.67	36.67	26.67	64.44	35.56	33.33	35.56	50.00	26.67	23.33	23.33
8	93.33	40.00	36.67	33.33	77.78	35.56	33.33	35.56	65.00	26.67	23.33	23.33
10	96.67	40.00	40.00	33.33	82.22	35.56	33.33	35.56	76.67	26.67	23.33	23.33

Table 26 (continued)

Cut	6 Bottlenecks				9 Bottlenecks				12 Bottlenecks			
points	iter #1	iter #2	iter #3	iter #4	iter #1	iter #2	iter #3	iter #4	iter #1	iter #2	iter #3	iter #4
12	96.67	40.00	40.00	33.33	82.22	35.56	33.33	35.56	78.33	26.67	23.33	23.33
14	96.67	40.00	40.00	33.33	84.44	35.56	33.33	35.56	81.67	26.67	23.33	23.33
16	96.67	40.00	40.00	33.33	84.44	35.56	33.33	35.56	81.67	26.67	23.33	23.33
18	96.67	40.00	40.00	33.33	84.44	35.56	33.33	35.56	81.67	26.67	23.33	23.33
20	96.67	40.00	40.00	33.33	86.67	35.56	33.33	35.56	83.33	26.67	23.33	23.33

Table 27 F-score for FOREPOST when $n_u = 10$ and $n_p = 20$

Cut	6 Bottlenecks				9 Bottlenecks				12 Bottlenecks			
points	iter #1	iter #2	iter #3	iter #4	iter #1	iter #2	iter #3	iter #4	iter #1	iter #2	iter #3	iter #4
2	50.00	45.00	45.00	40.00	36.36	36.36	36.36	32.73	28.57	25.71	28.57	25.71
4	80.00	40.00	44.00	32.00	61.54	43.08	43.08	46.15	50.00	37.50	35.00	35.00
6	86.67	36.67	36.67	26.67	77.33	42.67	40.00	42.67	66.67	35.56	31.11	31.11
8	80.00	34.29	31.43	28.57	82.35	37.65	35.29	37.65	78.00	32.00	28.00	28.00
10	72.50	30.00	30.00	25.00	77.89	33.68	31.58	33.68	83.64	29.09	25.45	25.45
12	64.44	26.67	26.67	22.22	70.48	30.48	28.57	30.48	78.33	26.67	23.33	23.33
14	58.00	24.00	24.00	20.00	66.09	27.83	26.09	27.83	75.38	24.62	21.54	21.54
16	52.73	21.82	21.82	18.18	60.80	25.60	24.00	25.60	70.00	22.86	20.00	20.00
18	48.33	20.00	20.00	16.67	56.30	23.70	22.22	23.70	65.33	21.33	18.67	18.67
20	44.62	18.46	18.46	15.38	53.79	22.07	20.69	22.07	62.50	20.00	17.50	17.50

Table 28 Precision for FOREPOST when $n_u = 15$ and $n_p = 10$

Cut	6 Bottlenecks				9 Bottlenecks				12 Bottlenecks			
points	iter #1	iter #2	iter #3	iter #4	iter #1	iter #2	iter #3	iter #4	iter #1	iter #2	iter #3	iter #4
2	100.00	60.00	80.00	90.00	100.00	100.00	70.00	100.00	100.00	90.00	100.00	90.00
4	90.00	40.00	50.00	55.00	100.00	75.00	55.00	75.00	100.00	65.00	65.00	75.00
6	80.00	33.33	33.33	36.67	100.00	56.67	40.00	50.00	100.00	50.00	53.33	53.33
8	62.50	25.00	25.00	27.50	100.00	50.00	32.50	37.50	100.00	40.00	40.00	40.00
10	52.00	22.00	20.00	22.00	90.00	40.00	26.00	30.00	98.00	34.00	32.00	32.00
12	43.33	18.33	16.67	18.33	75.00	33.33	21.67	25.00	96.67	28.33	26.67	26.67
14	37.14	15.71	14.29	15.71	64.29	28.57	18.57	21.43	85.71	24.29	22.86	22.86
16	33.75	13.75	12.50	13.75	56.25	25.00	16.25	18.75	75.00	21.25	20.00	20.00
18	30.00	12.22	11.11	12.22	50.00	22.22	14.44	16.67	66.67	18.89	17.78	17.78
20	27.00	11.00	10.00	11.00	45.00	20.00	13.00	15.00	60.00	17.00	16.00	16.00

Table 29 Recall for FOREPOST when $n_u = 15$ and $n_p = 10$

Cut	6 Bottlenecks				9 Bottlenecks				12 Bottlenecks			
points	iter #1	iter #2	iter #3	iter #4	iter #1	iter #2	iter #3	iter #4	iter #1	iter #2	iter #3	iter #4
2	33.33	20.00	26.67	30.00	22.22	22.22	15.56	22.22	16.67	15.00	16.67	15.00
4	60.00	26.67	33.33	36.67	44.44	33.33	24.44	33.33	33.33	21.67	21.67	25.00
6	80.00	33.33	33.33	36.67	66.67	37.78	26.67	33.33	50.00	25.00	26.67	26.67
8	83.33	33.33	33.33	36.67	88.89	44.44	28.89	33.33	66.67	26.67	26.67	26.67
10	86.67	36.67	33.33	36.67	100.00	44.44	28.89	33.33	81.67	28.33	26.67	26.67
12	86.67	36.67	33.33	36.67	100.00	44.44	28.89	33.33	96.67	28.33	26.67	26.67
14	86.67	36.67	33.33	36.67	100.00	44.44	28.89	33.33	100.00	28.33	26.67	26.67
16	90.00	36.67	33.33	36.67	100.00	44.44	28.89	33.33	100.00	28.33	26.67	26.67
18	90.00	36.67	33.33	36.67	100.00	44.44	28.89	33.33	100.00	28.33	26.67	26.67
20	90.00	36.67	33.33	36.67	100.00	44.44	28.89	33.33	100.00	28.33	26.67	26.67

Table 30 F-score for FOREPOST when $n_u = 15$ and $n_p = 10$

Cut	6 Bottlenecks				9 Bottlenecks				12 Bottlenecks			
points	iter #1	iter #2	iter #3	iter #4	iter #1	iter #2	iter #3	iter #4	iter #1	iter #2	iter #3	iter #4
2	50.00	30.00	40.00	45.00	36.36	36.36	25.45	36.36	28.57	25.71	28.57	25.71
4	72.00	32.00	40.00	44.00	61.54	46.15	33.85	46.15	50.00	32.50	32.50	37.50
6	80.00	33.33	33.33	36.67	80.00	45.33	32.00	40.00	66.67	33.33	35.56	35.56
8	71.43	28.57	28.57	31.43	94.12	47.06	30.59	35.29	80.00	32.00	32.00	32.00
10	65.00	27.50	25.00	27.50	94.74	42.11	27.37	31.58	89.09	30.91	29.09	29.09
12	57.78	24.44	22.22	24.44	85.71	38.10	24.76	28.57	96.67	28.33	26.67	26.67
14	52.00	22.00	20.00	22.00	78.26	34.78	22.61	26.09	92.31	26.15	24.62	24.62
16	49.09	20.00	18.18	20.00	72.00	32.00	20.80	24.00	85.71	24.29	22.86	22.86
18	45.00	18.33	16.67	18.33	66.67	29.63	19.26	22.22	80.00	22.67	21.33	21.33
20	41.54	16.92	15.38	16.92	62.07	27.59	17.93	20.69	75.00	21.25	20.00	20.00

Table 31 Precision for FOREPOST when $n_u = 15$ and $n_p = 15$

Cut	6 Bottlenecks				9 Bottlenecks				12 Bottlenecks			
points	iter #1	iter #2	iter #3	iter #4	iter #1	iter #2	iter #3	iter #4	iter #1	iter #2	iter #3	iter #4
2	100.00	70.00	60.00	60.00	100.00	90.00	80.00	80.00	100.00	100.00	100.00	90.00
4	100.00	55.00	60.00	40.00	100.00	75.00	80.00	55.00	100.00	80.00	80.00	80.00
6	90.00	36.67	40.00	26.67	96.67	53.33	53.33	43.33	100.00	76.67	76.67	76.67
8	72.50	30.00	30.00	22.50	97.50	40.00	40.00	32.50	100.00	60.00	60.00	62.50
10	58.00	24.00	24.00	18.00	86.00	32.00	32.00	26.00	98.00	48.00	48.00	50.00
12	48.33	20.00	20.00	15.00	71.67	26.67	26.67	21.67	93.33	40.00	41.67	41.67
14	41.43	17.14	17.14	12.86	62.86	22.86	22.86	18.57	82.86	34.29	35.71	35.71
16	36.25	15.00	15.00	11.25	55.00	20.00	20.00	16.25	75.00	31.25	31.25	31.25
18	32.22	13.33	13.33	10.00	48.89	17.78	17.78	14.44	66.67	27.78	27.78	27.78
20	29.00	12.00	12.00	9.00	44.00	16.00	16.00	13.00	60.00	25.00	25.00	25.00

Table 32 Recall for FOREPOST when $n_u = 15$ and $n_p = 15$

Cut	6 Bottlenecks				9 Bottlenecks				12 Bottlenecks			
points	iter #1	iter #2	iter #3	iter #4	iter #1	iter #2	iter #3	iter #4	iter #1	iter #2	iter #3	iter #4
2	33.33	23.33	20.00	20.00	22.22	20.00	17.78	17.78	16.67	16.67	16.67	15.00
4	66.67	36.67	40.00	26.67	44.44	33.33	35.56	24.44	33.33	26.67	26.67	26.67
6	90.00	36.67	40.00	26.67	64.44	35.56	35.56	28.89	50.00	38.33	38.33	38.33
8	96.67	40.00	40.00	30.00	86.67	35.56	35.56	28.89	66.67	40.00	40.00	41.67
10	96.67	40.00	40.00	30.00	95.56	35.56	35.56	28.89	81.67	40.00	40.00	41.67
12	96.67	40.00	40.00	30.00	95.56	35.56	35.56	28.89	93.33	40.00	41.67	41.67
14	96.67	40.00	40.00	30.00	97.78	35.56	35.56	28.89	96.67	40.00	41.67	41.67
16	96.67	40.00	40.00	30.00	97.78	35.56	35.56	28.89	100.00	41.67	41.67	41.67
18	96.67	40.00	40.00	30.00	97.78	35.56	35.56	28.89	100.00	41.67	41.67	41.67
20	96.67	40.00	40.00	30.00	97.78	35.56	35.56	28.89	100.00	41.67	41.67	41.67

Table 33 F-score for FOREPOST when $n_u = 15$ and $n_p = 15$

Cut	6 Bottlenecks				9 Bottlenecks				12 Bottlenecks			
points	iter #1	iter #2	iter #3	iter #4	iter #1	iter #2	iter #3	iter #4	iter #1	iter #2	iter #3	iter #4
2	50.00	35.00	30.00	30.00	36.36	32.73	29.09	29.09	28.57	28.57	28.57	25.71
4	80.00	44.00	48.00	32.00	61.54	46.15	49.23	33.85	50.00	40.00	40.00	40.00
6	90.00	36.67	40.00	26.67	77.33	42.67	42.67	34.67	66.67	51.11	51.11	51.11
8	82.86	34.29	34.29	25.71	91.76	37.65	37.65	30.59	80.00	48.00	48.00	50.00
10	72.50	30.00	30.00	22.50	90.53	33.68	33.68	27.37	89.09	43.64	43.64	45.45
12	64.44	26.67	26.67	20.00	81.90	30.48	30.48	24.76	93.33	40.00	41.67	41.67
14	58.00	24.00	24.00	18.00	76.52	27.83	27.83	22.61	89.23	36.92	38.46	38.46
16	52.73	21.82	21.82	16.36	70.40	25.60	25.60	20.80	85.71	35.71	35.71	35.71
18	48.33	20.00	20.00	15.00	65.19	23.70	23.70	19.26	80.00	33.33	33.33	33.33
20	44.62	18.46	18.46	13.85	60.69	22.07	22.07	17.93	75.00	31.25	31.25	31.25

Table 34 Precision for FOREPOST when $n_u = 15$ and $n_p = 20$

Cut	6 Bottlenecks				9 Bottlenecks				12 Bottlenecks			
points	iter #1	iter #2	iter #3	iter #4	iter #1	iter #2	iter #3	iter #4	iter #1	iter #2	iter #3	iter #4
2	90.00	50.00	70.00	80.00	100.00	80.00	90.00	100.00	100.00	80.00	80.00	90.00
4	90.00	50.00	50.00	50.00	95.00	60.00	70.00	70.00	100.00	80.00	75.00	70.00
6	73.33	33.33	33.33	33.33	93.33	46.67	46.67	50.00	100.00	53.33	53.33	46.67
8	57.50	25.00	25.00	25.00	87.50	35.00	35.00	37.50	95.00	40.00	40.00	35.00
10	46.00	20.00	20.00	20.00	72.00	28.00	28.00	30.00	86.00	32.00	32.00	28.00
12	38.33	16.67	16.67	16.67	60.00	23.33	23.33	25.00	80.00	26.67	26.67	23.33
14	34.29	14.29	14.29	14.29	51.43	20.00	20.00	21.43	71.43	22.86	22.86	20.00
16	30.00	12.50	12.50	12.50	45.00	17.50	17.50	18.75	66.25	20.00	20.00	17.50
18	26.67	11.11	11.11	11.11	41.11	15.56	15.56	16.67	61.11	17.78	17.78	15.56
20	25.00	10.00	10.00	10.00	37.00	14.00	14.00	15.00	55.00	16.00	16.00	14.00

Table 35 Recall for FOREPOST when $n_u = 15$ and $n_p = 20$

Cut	6 Bottlenecks				9 Bottlenecks				12 Bottlenecks			
points	iter #1	iter #2	iter #3	iter #4	iter #1	iter #2	iter #3	iter #4	iter #1	iter #2	iter #3	iter #4
2	30.00	16.67	23.33	26.67	22.22	17.78	20.00	22.22	16.67	13.33	13.33	15.00
4	60.00	33.33	33.33	33.33	42.22	26.67	31.11	31.11	33.33	26.67	25.00	23.33
6	73.33	33.33	33.33	33.33	62.22	31.11	31.11	33.33	50.00	26.67	26.67	23.33
8	76.67	33.33	33.33	33.33	77.78	31.11	31.11	33.33	63.33	26.67	26.67	23.33
10	76.67	33.33	33.33	33.33	80.00	31.11	31.11	33.33	71.67	26.67	26.67	23.33
12	76.67	33.33	33.33	33.33	80.00	31.11	31.11	33.33	80.00	26.67	26.67	23.33
14	80.00	33.33	33.33	33.33	80.00	31.11	31.11	33.33	83.33	26.67	26.67	23.33
16	80.00	33.33	33.33	33.33	80.00	31.11	31.11	33.33	88.33	26.67	26.67	23.33
18	80.00	33.33	33.33	33.33	82.22	31.11	31.11	33.33	91.67	26.67	26.67	23.33
20	83.33	33.33	33.33	33.33	82.22	31.11	31.11	33.33	91.67	26.67	26.67	23.33

Table 36 F-score for FOREPOST when $n_u = 15$ and $n_p = 20$

Cut	6 Bottlenecks				9 Bottlenecks				12 Bottlenecks			
points	iter #1	iter #2	iter #3	iter #4	iter #1	iter #2	iter #3	iter #4	iter #1	iter #2	iter #3	iter #4
2	45.00	25.00	35.00	40.00	36.36	29.09	32.73	36.36	28.57	22.86	22.86	25.71
4	72.00	40.00	40.00	40.00	58.46	36.92	43.08	43.08	50.00	40.00	37.50	35.00
6	73.33	33.33	33.33	33.33	74.67	37.33	37.33	40.00	66.67	35.56	35.56	31.11
8	65.71	28.57	28.57	28.57	82.35	32.94	32.94	35.29	76.00	32.00	32.00	28.00
10	57.50	25.00	25.00	25.00	75.79	29.47	29.47	31.58	78.18	29.09	29.09	25.45
12	51.11	22.22	22.22	22.22	68.57	26.67	26.67	28.57	80.00	26.67	26.67	23.33
14	48.00	20.00	20.00	20.00	62.61	24.35	24.35	26.09	76.92	24.62	24.62	21.54
16	43.64	18.18	18.18	18.18	57.60	22.40	22.40	24.00	75.71	22.86	22.86	20.00
18	40.00	16.67	16.67	16.67	54.81	20.74	20.74	22.22	73.33	21.33	21.33	18.67
20	38.46	15.38	15.38	15.38	51.03	19.31	19.31	20.69	68.75	20.00	20.00	17.50

References

Achenbach M, Ostermann K (2009) Engineering abstractions in model checking and testing. IEEE Intl Workshop SCAM:137–146

Aguilera MK, Mogul JC, Wiener JL, Reynolds P, Muthitacharoen A (2003) Performance debugging for distributed systems of black boxes. In: SOSP, pp 74–89

Ammann P, Offutt J (2008) Introduction to software testing. Cambridge University Press

Ammons G, Choi JD, Gupta M, Swamy N (2004) Finding and removing performance bottlenecks in large systems. In: ECOOP, pp 170–194

Arcuri A, Briand LC (2011) A practical guide for using statistical tests to assess randomized algorithms in software engineering. In: ICSE, pp 1–10

Ashley C (2006) Application performance management market offers attractive benefits to european service providers. The Yankee Group

Avritzer A, Weyuker EJ (1994) Generating test suites for software load testing. In: ISSTA, pp 44–57

Avritzer A, Weyuker EJ (1996) Deriving workloads for performance testing, vol 26. Wiley, New York, pp 613–633

Avritzer A, de Souza e Silva E, Leão RMM, Weyuker EJ (2011) Automated generation of test cases using a performability model. Software IET 5(2):113–119

Barna C, Litoiu M, Ghanbari H (2011) Autonomic load-testing framework. In: roceedings of the 8th ACM international conference on autonomic computing, ICAC '11. ACM, USA, pp 91–100

Bayan M, Cangussu JaW (2008) Automatic feedback, control-based, stress and load testing. In: Proceedings of the 2008 ACM symposium on applied computing, SAC 08. ACM, USA, pp 661–666

Beck K (2003) Test-driven development: by example. The Addison-Wesley Signature Series. Addison-Wesley

Bird DL, Munoz CU (1983) Automatic generation of random self-checking test cases. IBM Syst J 22:229–245

Bishop CM (2006) Pattern Recognition and Machine Learning (Information Science and Statistics). Springer, Secaucus

Briand LC, Labiche Y, Shousha M (2005) Stress testing real-time systems with genetic algorithms. In: Proceedings of the 7th annual conference on genetic and evolutionary computation, GECCO 05. ACM, USA, pp 1021–1028

Cohen J (2013) Statistical power analysis for the behavioral sciences. Academic Press

Cohen WW (1995) Fast effective rule induction. In: Twelfth ICML, pp 115–123

Cornelissen W, Klaassen A, Matsinger A, van Wee G (1995) How to make intuitive testing more systematic. IEEE Softw 12(5):87–89

Dickinson W, Leon D, Podgurski A (2001) Finding failures by cluster analysis of execution profiles. In: ICSE, pp 339–348

Dijkstra EW (1976) A discipline of programming, vol 1. Englewood Cliffs: prentice-hall

Dustin E, Rashka J, Paul J (1999) Automated software testing: introduction, management, and performance. Addison-Wesley Professional

Fewster M, Graham D (1999) Software test automation: effective use of test execution tools. ACM Press/Addison-Wesley Publishing Co.

Foo KC, Jiang ZM, Adams B, Hassan AE, Zou Y, Flora P (2010) Mining performance regression testing repositories for automated performance analysis. In: 10th international conference on quality software (QSIC), IEEE, pp 32–41

Freeman S, Mackinnon T, Pryce N, Walnes J (2004) Mock roles, objects. In: Companion to OOPSLA '04, pp 236–246

Furnkranz J, Widmer G (1994) Incremental reduced error pruning. In: International conference on machine learning, pp 70–77

Garbani JP (2008) Market overview: the application performance management market. Forrester Research

Glenford JM (1979) The art of software testing. Wiley. ISBN 10:0471043281

Grant S, Cordy JR, Skillicorn D (2008) Automated concept location using independent component analysis. In: WCRE '08, pp 138–142

Grechanik M, Fu C, Xie Q (2012) Automatically finding performance problems with feedback-directed learning software testing. In: 34th international conference on software engineering (ICSE), pp 156–166

Grindal M, Offutt J, Andler SF (2005) Combination testing strategies: a survey. Software Testing, Verification, and Reliability 15:167–199

Group TY (2005) Enterprise application management survey. The Yankee Group

Hamlet D (2006) When only random testing will do. In: Proceedings of the 1st international workshop on random testing, RT '06. ACM, USA, pp 1–9. doi:10.1145/1145735.1145737

Hamlet R (1994) Random testing. In: Encyclopedia of Software Engineering. Wiley, pp 970–978

Haran M, Karr A, Orso A, Porter A, Sanil A (2005) Applying classification techniques to remotely-collected program execution data. In: ESEC/FSE-13, pp 146–155

Hull E, Jackson K, Dick J (2005) Requirements engineering. Springer

Hyvärinen A, Oja E (2000) Independent component analysis: algorithms and applications. Neural Netw 13(4–5):411–430

IEEE (1991) IEEE standard computer dictionary: a compilation of ieee standard computer glossaries

Isaacs R, Barham P (2002) Performance analysis in loosely-coupled distributed systems. In: 7th CaberNet Radicals Workshop

Jiang ZM, Hassan AE, Hamann G, Flora P (2009) Automated performance analysis of load tests. In: ICSM, pp 125–134

Jin G, Song L, Shi X, Scherpelz J, Lu S (2012) Understanding and detecting real-world performance bugs. In: Proceedings of the 33rd ACM SIGPLAN conference on programming language design and implementation, pp 77–88

Jovic M, Adamoli A, Hauswirth M (2011) Catch me if you can: performance bug detection in the wild. ACM SIGPLAN Not 46(10):155–170

Kaner C (1997) Improving the maintainability of automated test suites. Software QA 4(4)

Kaner C (2003) What is a good test case? In: Software Testing Analysis & Review Conference (STAR) East

Koziolek H (2005) Operational profiles for software reliability. In: Seminar on Dependability Engineering, Germany, Citeseer

Linares-Vásquez M, Mcmillan C, Poshyvanyk D, Grechanik M (2014) On using machine learning to automatically classify software applications into domain categories. Empirical Softw Engg 19(3):582–618. doi:10.1007/s10664-012-9230-z

Lowry R (2014) Concepts and applications of inferential statistics. R. Lowry

Malik H, Adams B, Hassan AE (2010) Pinpointing the subsystems responsible for the performance deviations in a load test. In: IEEE 21st international symposium on software reliability engineering (ISSRE), IEEE, pp 201–210

Malik H, Hemmati H, Hassan AE (2013) Automatic detection of performance deviations in the load testing of large scale systems. In: Proceedings of the 2013 international conference on software engineering, IEEE Press, pp 1012–1021

McMillan C, Linares-Vasquez M, Poshyvanyk D, Grechanik M (2011) Categorizing software applications for maintenance. In: Proceedings of the 2011 27th IEEE international conference on software maintenance, ICSM 11. IEEE Computer Society, USA, pp 343–352. doi:10.1109/ICSM.2011.6080801

Menascé DA (2002) Load testing, benchmarking, and application performance management for the web. In: International CMG Conference, pp 271–282

Molyneaux I (2009) The art of application performance testing: help for programmers and quality assurance. O'Reilly Media, Inc

Murphy TE (2008) Managing test data for maximum productivity. Tech. rep

Musa JD (1993) Operational profiles in software-reliability engineering, vol 10. IEEE Computer Society Press, Los Alamitos, pp 14–32

Nistor A, Jiang T, Tan L (2013) Discovering, reporting, and fixing performance bugs. In: Proceedings of the 10th international workshop on mining software repositories, pp 237–246

Nistor A, Chang PC, Radoi C, Lu S (2015) Caramel: detecting and fixing performance problems that have non-intrusive fixes. ICSE

Park S, Hossain BMM, Hussain I, Csallner C, Grechanik M, Taneja K, Fu C, Xie Q (2012) Carfast: Achieving higher statement coverage faster. In: Proceedings of the ACM SIGSOFT 20th international symposium on the foundations of software engineering, FSE 12. ACM, USA, pp 35:1–35:11. doi:10.1145/2393596.2393636

Parnas DL (1972) On the criteria to be used in decomposing systems into modules. Commun ACM 15:1053–1058

Parsons S (2005) Independent component analysis: a tutorial introduction. Knowl Eng Rev 20(2):198–199

Schwaber C, Mines C, Hogan L (2006) Performance-driven software development: How it shops can more efficiently meet performance requirements. Forrester Research

Shapiro SS, Wilk MB (1965) An analysis of variance test for normality (complete samples). Biometrika 52(3/4):591–611

Shen D, Luo Q, Poshyvanyk D, Grechanik M (2015) Automating performance bottleneck detection using search-based application profiling. In: Proceedings of the 2015 international symposium on software testing and analysis, ISSTA 15. ACM, USA, pp 270–281. doi:10.1145/2771783.2771816

Syer MD, Jiang ZM, Nagappan M, Hassan AE, Nasser M, Flora P (2013) Leveraging performance counters and execution logs to diagnose memory-related performance issues. In: 29th IEEE international conference on software maintenance (ICSM), IEEE, pp 110–119

Tarr PL, Ossher H, Harrison WH Jr (1999) SMS, Degrees of separation: Multi-dimensional separation of concerns. In: ICSE, pp 107–119

Tian K, Revelle M, Poshyvanyk D (2009) Using latent dirichlet allocation for automatic categorization of software. In: Proceedings of the 2009 6th IEEE international working conference on mining software repositories, MSR 09. IEEE Computer Society, USA, pp 163–166. doi:10.1109/MSR.2009.5069496

Westcott MR (1968) Toward a contemporary psychology of intuition: a historical, theoretical, and empirical inquiry. Holt, Rinehart and Winston

Weyuker EJ, Vokolos FI (2000) Experience with performance testing of software systems: Issues, an approach, and case study. IEEE Trans Softw Eng 26(12):1147–1156

Wilcoxon F (1945) Individual comparisons by ranking methods. Biom Bull 1(6):80–83

Wildstrom J, Stone P, Witchel E, Dahlin M (2007) Machine learning for on-line hardware reconfiguration. In: IJCAI'07, pp 1113–1118

Witten IH, Frank E (2005) Data mining: practical machine learning tools and techniques. Morgan Kaufmann

Yuhanna N (2009) Dbms selection: look beyond basic functions. Forrester Research

Zaman S, Adams B, Hassan AE (2011) Security versus performance bugs: a case study on firefox. In: Proceedings of the 8th working conference on mining software repositories, ACM, pp 93–102

Zaman S, Adams B, Hassan AE (2012) A qualitative study on performance bugs. In: 9th IEEE working conference on mining software repositories (MSR), pp 199–208
Zaparanuks D, Hauswirth M (2012) Algorithmic profiling. ACM SIGPLAN Not 47(6):67–76
Zhang P, Elbaum SG, Dwyer MB (2011) Automatic generation of load tests. In: ASE, pp 43–52

Qi Luo is a Ph.D. candidate in the department of Computer Science at The College of William and Mary, advised by Dr. Denys Poshyvanyk. She received her Bachelor degree from Beihang University and her Master degree from Tsinghua University. Her research interests are in software engineering, performance testing and regression testing. She is a student member of ACM and IEEE.

Aswathy Nair is a Technology Analyst at Bank of America Merrill Lynch. She received her B.Tech in Information Technology from Anna University in 2008 and her MS in Computer Science from University of Illinois at Chicago in 2011. Her MS thesis focused on Automatically Finding abstractions for Input Space Partitioning for Software Performance Testing.

Mark Grechanik is an Assistant Professor at the department of computer science of the University of Illinois at Chicago. He received his Ph.D. in Computer Sciences from the University of Texas at Austin. Dr. Grechanik has served as a member of ACM SigSoft Executive Committee since 2009. His research area is software engineering in general, with particular interests in software testing, evolution, and reuse. He is also interested in problems that lie at the intersection of software engineering and data privacy. Dr.Grechanik has a unique blend of strong academic background and long-term industry experience. He is a Senior Member of IEEE and a Senior Member of ACM.

Denys Poshvanyk is an Associate Professor at the College of William and Mary in Virginia. He received his Ph.D. degree in Computer Science from Wayne State University in 2008. He also obtained his M.S. and M.A. degrees in Computer Science from the National University of Kyiv-Mohyla Academy, Ukraine and Wayne State University in 2003 and 2006, respectively. He served as a program co-chair for the ICSME 2016, ICPC 2013, WCRE 2012 and WCRE 2011. His research interests are in software engineering, software maintenance and evolution, program comprehension, reverse engineering, software repository mining, source code analysis, and metrics. He is member of the IEEE and ACM.