FOREPOST: A Tool For Detecting Performance Problems with Feedback-Driven Learning Software Testing

Qi Luo, Denys Poshyvanyk
The College of William and Mary
Williamsburg, VA 23185, USA
{qluo, denys}@cs.wm.edu

Aswathy Nair, Mark Grechanik
University of Illinois at Chicago
Chicago, IL 60601, USA
nair.a.87@gmail.com, drmark@uic.edu

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 to find performance problems in applications automatically.

We present a novel tool, FOREPOST, for finding performance problems in applications automatically using black-box software testing. In this paper, we demonstrate how FOREPOST extracts rules from execution traces of applications by using machine learning algorithms, and then uses these rules to select test input data automatically to steer applications towards computationally intensive paths and to find performance problems. FOREPOST is available in our online appendix (http://www.cs.wm.edu/semeru/data/ICSE16-FOREPOST), which contains the tool, source code and demo video.

CCS Concepts

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

Keywords

Performance testing; machine learning; black-box testing

1. INTRODUCTION

A goal of performance testing is to find performance problems, when an *application under test (AUT)* unexpectedly exhibits worsened characteristics for a specific workload [16, 18]. For example, effective test cases for *load testing*, which is a variant of performance testing, find situations where an AUT suffers from unexpectedly high response time or low throughput [4, 13]. 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 [12]. It is difficult to construct effective performance test cases that can find performance problems in a short period of time, since it requires test engineers to test many combinations of actions and data for nontrivial applications.

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 problems [17, 21]. Naturally, testers want to summarize the behavior of an AUT concisely in terms of its inputs, so that they can select input data that will lead to significantly increased resource consumption thereby revealing performance problems. Unfortunately, finding proper rules that collectively describe properties of such input data is a highly creative process that involves deep understanding of input domains [3, page 152].

Descriptive rules for selecting test input data play a significant role in software testing [5], where 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 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, since 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. Even though real-world systems exhibit much more complex behaviors, useful descriptive rules often enable testers to build effective performance fault revealing test cases.

Rules may provide insight into the behavior of the AUT. For example, a rule may specify that the method `checkFraud` is always invoked when test cases are good for exposing performance bottlenecks and the values of the attribute `SecurityDeposit` of the table `Finances` are frequently retrieved from the back-end 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, and thus these rules can be used to select better test cases automatically.

We demonstrate our tool, FOREPOST, for finding performance problems automatically by learning and using rules that describe classes of input data that lead to intensive computations. FOREPOST implements the feedback-driven performance software testing approach proposed and evaluated on four subject applications, Renters, JPetStore, Dell DVD Store and Agilefnat, in our pervious works [8, 15]. FOREPOST learns rules from AUT execution traces and uses these learned rules to select test input data automatically to find more performance problems in applications as compared to random performance testing. FOREPOST uses runtime monitoring for a short duration of testing together with machine learning techniques and automates test scripts to reduce large amounts of performance-related information collected during AUT runs. Its goal is to select a small number of descriptive rules that provide

insights into properties of test input data that lead to increased computational loads of applications.

In the current version of FOREPOST, it supports the performance testing on Java applications. It takes the binary code and the test input data of the AUTs as inputs, and outputs a set of rules that describe input data to steer AUT towards computationally intensive paths and a ranked list of methods likely to be performance bottlenecks. FOREPOST, its running requirements and the experimental results are publicly available [14]. We also provide a demo video to show how FOREPOST works on one of our subject applications, Agilefant, in our online appendix [14].

2. DEMONSTRATING THE FOREPOST

FOREPOST is a tool of performance testing, which automatically learns rules that describe the AUT behaviors in terms of input data and uses these rules to select inputs and detect performance problems. FOREPOST is built on two key ideas: 1) extracting rules from execution traces that describe relationships between the input data and the performance of the tests that are executed with this data and 2) identifying bottlenecks using these rules. FOREPOST only requires the binary code and the test input data of the AUTs, thus, no source code is needed for performance testing via a FOREPOST. A ranked list of methods is obtained finally. The top methods are likely to be performance bottlenecks, which needs to be further confirmed by test engineers. Currently, FOREPOST supports performance testing of Java applications due to the implementation of the underlying profiler (i.e., TPTP [2]). However, it is straightforward to extend FOREPOST to programs in different programming languages by using different profiling tools. The architecture of FOREPOST is shown in Figure 1. In this section, we introduce our tool and its architecture in details.

2.1 Obtaining Rules

As part of the first key idea, the instrumented AUT is initially run using a small number of selected test input data (see step 1 in Figure 1). In our implementation, test engineers write the test scripts to randomly select combinations of test input data and send the selected inputs to the AUT automatically. The input test data comes from existing repositories or databases; it is a common practice in industry as we confirmed it with different performance testing professionals. For example, 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. For web-based applications, we wrote the test scripts that contains all URL requests with different parameters, and used JMeter [1] to simulate users sending URL requests to execute applications. In the test scripts, we defined a set of URL requests that comes from one user as a transaction, and used JMeter to send five transactions concurrently. While initial test input data is selected randomly in the current implementation of FOREPOST, in practice, test engineers can also supply input data that likely reveal bottlenecks or any input data that they prefer to start with.

Once the test script starts executing the application, its execution traces are collected by the profiler, and these traces are forwarded to the execution trace analyzer, which produces the trace statistics (steps 2-3 in Figure 1). We implemented the profiler using the TPTP framework [2], which can inject probes into binary code for collecting running time information, such as the system time, the size of parameters, and the amount of transferred data between the AUTs and databases. Based on collected profiles, trace statistics of each trace is obtained, such as the number of invoked methods, the elapsed time to complete the end-to-end application run, and the number of invocations. Figure 2 (a) shows the results of trace statistics in FOREPOST, where the elapsed time of each trace
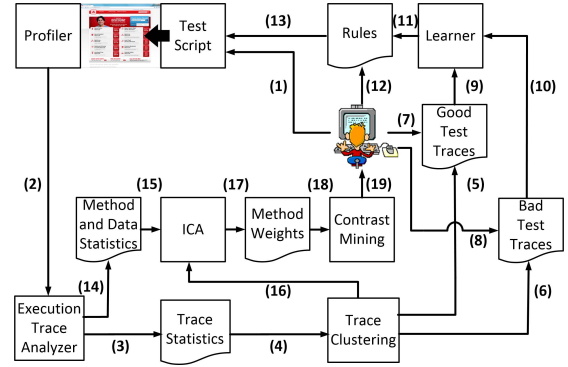

Figure 1: The Architecture of FOREPOST.

is shown in the second column (see ⟨1⟩ in Figure 2). Currently, FOREPOST supports to use elapsed execution time for evaluating the performance behaviors of each trace. We chose elapsed execution time as the performance metric since it is one of the most widely used metrics. Of course, FOREPOST can consider other different types of performance metrics by changing the settings of the profiler. We leave it as the future work.

The trace statistics is supplied to the trace clustering, which clusters collected traces into two different groups that collectively describe different performance of the AUTs. For example, there can be as few as two groups that correspond to good and bad performance test cases (steps 4-8 in Figure 1). The *good* traces make the AUT consume more resources and time, which are "good" to reveal performance bottlenecks. Conversely, the *bad* traces utilize few resources or take less time as compared to the good traces, which are "bad" to expose performance bottlenecks. FOREPOST uses the average elapsed execution time to cluster traces. The traces with longer elapsed execution times than the average value are considered as good traces. The rationale here is that the traces with longer elapsed time would lead to intensive computations and increased workloads, likely to reveal performance bottlenecks. Otherwise, the traces would be assigned as bad ones. As Figure 2 (b) shows, all collected traces are clustered into different groups. Test engineers can review these traces and modify the assignment, like changing "good" to "bad" (see ⟨2⟩ in Figure 2), if necessary.

After clustering traces, each trace is represented as a vector-based form $V_{I_1}, \ldots, V_{I_k} \to T$, where V_{I_m} is the value of the input I_m and $T \in \{G, B\}$, with G and B refering good and bad test cases correspondingly. The format of V_{I_m} can be different depends on the AUT. For example, for a web-based application, it is a vector referring to a set of URL requests. These vectors are used as inputs to the learner, which applies a Machine Learning (ML) algorithm to learn the classification model and output rules (steps 9-11 in Figure 1). FOREPOST uses the class JRip in Weka [20] to implement a propositional rule learner, Repeated Incremental Pruning to Produce Error Reduction (RIPPER) [6]. It combines pre-pruning and post-pruning into the learning process, following a separate-and-conquer strategy. The rules have the form $I_1 \odot V_{I_1} \bullet I_2 \odot V_{I_2} \bullet \ldots \bullet I_k \odot V_{I_k} \to T$, where $\odot$ is one of the relational operators (e.g., $>$ and $=$) and $\bullet$ is one of the logical connectors (i.e., $\wedge$ and $\vee$). For instance, as Figure 2 (c) shows, rules describe the relationships between test input and the performance behavior of the AUT (see ⟨3⟩ in Figure 2). Similarly, the testers can review these rules and mark some of them as erroneous if they have sufficient evidence to do so (step 12 in Figure 1 and ⟨4⟩ in Figure 2). For example, if testers find one rule is incorrect, they can mark its active status as "false" instead of "true".

A feedback loop is formed by supplying these learned rules back into the test script to automatically guide selection of test inputs
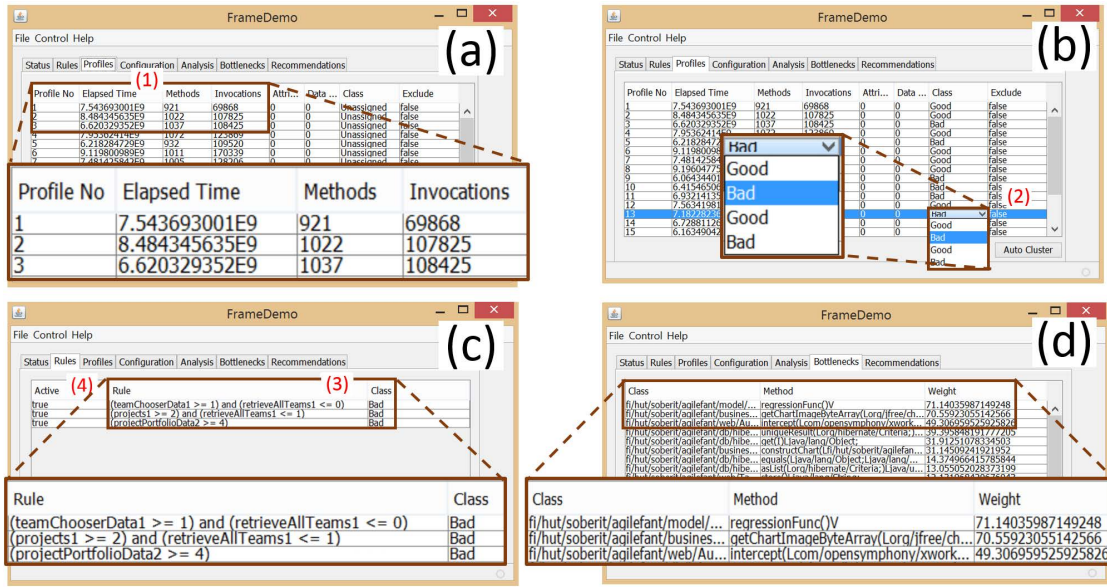
Figure 2: The four views of FOREPOST: (a) the view showing trace statistics, (b) the view showing automatic clustering, (c) the view showing learned rules, and (d) the view showing identified performance bottlenecks.

(step 13 in Figure 1). Once the test script receives a new set of rules, it partitions the input space into blocks according to these rules and starts selecting test inputs from each block. The profiler collects traces of these new test runs, and thus the cycle repeats. The new rules can be *re*learned at each several passes and the input space is repartitioned adaptively according to new rules. When no new rules are learned after some time of testing, the partition of test inputs is stable with a high probability. At this point instrumentation can be removed and testing can continue with these selected inputs.

2.2 Identifying Bottlenecks

Our goal is to help testers to identify bottlenecks automatically 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 to check to see if the content of some file is modified. While this method may be one bottleneck, it is invoked in both good and bad traces, thus its contribution to 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 traces and that are not invoked or have little significance in bad traces, where the significance of a method is a function of the resource consumption that its execution triggers. Thus, once the input space is partitioned into clusters that lead to good and bad traces, we want to find methods that are specific to good traces and that are most likely to contribute to bottlenecks.

This task is accomplished in parallel to computing rules, and it starts when the trace analyzer produces the method statistics used to construct two matrices $\| \mathbf{x_B} \|$ and $\| \mathbf{x_G} \|$ for bad and good traces correspondingly (steps 14-16 in Figure 1). While FOREPOST currently considers the elapsed execution time of each method as its resource consumption, users can easily extend FOREPOST by measuring the consumptions as a normalized weighted sum of the times of method's invocations, the total elapsed time, the number of attributes accessed in the databases, and the amount of data transferred between the AUTs and the databases. Once these matrices are constructed, *Independent Component Analysis (ICA)* decomposes them into the matrices $\| \mathbf{s_B} \|$ and $\| \mathbf{s_G} \|$ for bad and good traces correspondingly. FOREPOST uses a Java library FastICA to implement the Fast and Robust Fixed-Point Algorithms for I-

CA [11]. Crossreferencing the matrices $\| \mathbf{s_B} \|$ and $\| \mathbf{s_G} \|$, FOREPOST specifies method weights for high-level requirements for different groups (i.e., good or bad), and does the contrast mining (see details in our previous work [15]) to assign larger weights to the methods most significant in good traces but not invoked or have little significance in bad traces. All methods are ranked based on their weights, as Figure 2 (d) shows. The advisor can determine whether the performance testers should look at the top methods to debug possible performance problems (steps 17-19 in Figure 1).

2.3 The Availability

More information about FOREPOST can be found in our online appendix, which contains (i) a video demonstrating FOREPOST, (ii) links for downloading the source code of FOREPOST, (iii) links for downloading the used tools, such as Weka and TPTP framework, (iv) examples of detected performance bottlenecks, and (v) the architecture of FOREPOST.

2.4 Usage Examples

FOREPOST was evaluated on one nontrivial application at an insurance company, Renters, and three open-source applications, JPetStore, Dell DVD Store and Agilefant. JPetStore and Dell DVD Store are open-source simulations of e-commerce sites, and Agilefant is an enterprise-level project management system. All the subjects are available at our online appendix [14]. The results are shown in our previous works [8, 15]. In this section, we show some examples of rules and detected performance bottlenecks.

Obtaining Rules. FOREPOST uses ML algorithm to extract the rules that describe relationship between test input data and the AUT behaviors. These rules are used to generate input data that steers AUT towards computationally intensive execution paths. Here are four examples of rules for Renters, JPetStore, Dell DVD Store and Agilefant respectively:

- (childOrAdultCareDetails.numberPersonsCaredForChild $\geqslant 3$) and (personalInjuryDetails.nationalEntertainerAthlete $= Y$) $\rightarrow$ Good
- (browse_title_ACADEMY_AFRICAN_2 $\geqslant 5$) and (addToCart_5_1 $\leqslant 0$) Bad
- (viewPrdct_K9RT01 $\geqslant 5$) and (viewItem_EST16 $\geqslant 5$) $\rightarrow$ Bad
- (storeStory3 > 100) and (storeProject6 < 25) $\rightarrow$ Bad

Detecting Performance bottlenecks. FOREPOST provides a ranked list of methods likely to be performance bottlenecks. Testers can look into the top methods to check whether they are real performance bottlenecks. For instance, FOREPOST obtained a ranked list of methods from Renters, and these methods were reviewed by the most experienced developers and testers for Renters. After reviewing the methods and checking source code, they confirmed around 20 methods were bottlenecks. To illustrate, the identified method *checkWildFireArea* was confirmed as a true bottleneck, where an incorrect invocation of *checkWildFireArea* occurs due to the implementation of *Visitor* pattern. Even though it did not contribute to computing the insurance quote, it took significant computational resources to execute. Some methods were considered bottlenecks, since they processed too much XML data. In all, FOREPOST pointed out methods as bottlenecks, but a deeper analysis was needed. For open-source subjects, we injected artificial bottlenecks to analyze the effectiveness and accuracy of FOREPOST. The detailed results can be found in our previous works [8, 15].

3. RELATED WORK

Partition testing is a set of strategies that divides the program's input domain into subdomains from which test cases can be derived to cover each subset at least once [3]. Closely related is the work by Dickinson et al [7], which uses clustering analysis execution profiles to find failures among the executions induced by a set of potential test cases. Although we both used clustering techniques, FOREPOST differs in that it clusters the execution profiles based on the length of the execution time, and we target the performance bugs instead of functional errors.

Learning rules helps stakeholders to reconfigure distributed systems to optimize for dynamically changing workloads [9, 19]. One work [19] is similar to FOREPOST that uses learning methodology to learn rules, but it focuses on learning a set of hardware configurations with better performance under current workload. In contrast, FOREPOST uses feedback-directed adaptive test scripts to locate most computationally intensive executions and bottlenecks.

A technique automatically classifies execution data, collected in the field, as coming from either passing or failing program runs [10]. It attempts to learn a classification model to predict if an application run failed using execution data. Conversely, FOREPOST learns rules to select input test data that steer applications towards computationally intensive runs to expose performance problems.

In their recent work, Pingyu et al, generate performance test cases using dynamic symbolic execution [21]. Similar to FOREPOST, they use heuristics to generate test cases by determining paths of executions to introduce higher workloads. Unlike FOREPOST, white-box testing approach is used thus requiring access to source code while FOREPOST is an entirely black-box driven approach. It is also unclear how white-box approaches will scale up to large industrial subjects. We view the approach as complementary, where a hybrid approach may combine the benefits of both approaches in a grey-box testing. We leave it as the future work.

4. CONCLUSION

In this paper, we demonstrate a novel tool, FOREPOST, which finds performance problems in applications automatically using black-box software testing. Our tool is an adaptive, feedback-directed learning testing system that learns rules from execution traces of applications and uses these rules to select test input data for these applications automatically to find more performance problems when compared with random testing. FOREPOST takes the binary code and the test input data of the AUTs as inputs, and outputs a set of rules for steering the AUT towards computation-ally intensive execution paths and a ranked list of methods likely to be performance bottlenecks. The evaluation demonstrated that FOREPOST is effective in finding input data to trigger intensive computations and in identifying performance bottlenecks.

5. REFERENCES

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

[2] Tptp, http://www.eclipse.org/tptp.

[3] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.

[4] A. Avritzer and E. J. Weyuker. Generating test suites for software load testing. In *ISSTA*, pages 44–57, 1994.

[5] K. Beck. *Test-Driven Development: By Example*. The Addison-Wesley Signature Series. Addison-Wesley, 2003.

[6] W. W. Cohen. Fast effective rule induction. In *Twelfth ICML*, pages 115–123. Morgan Kaufmann, 1995.

[7] W. Dickinson, D. Leon, and A. Podgurski. Finding failures by cluster analysis of execution profiles. In *ICSE*, 2001.

[8] M. Grechanik, C. Fu, and Q. Xie. Automatically finding performance problems with feedback-directed learning software testing. In *ICSE'12*, pages 156–166, 2012.

[9] M. Grechanik, Q. Luo, D. Poshyvanyk, and A. Porter. Enhancing rules for cloud resource provisioning via learned software performance models. In *ICPE'16*, 2016.

[10] M. Haran, A. Karr, A. Orso, A. Porter, and A. Sanil. Applying classification techniques to remotely-collected program execution data. In *FSE'13*, pages 146–155, 2005.

[11] A. Hyvärinen. Fast and robust fixed-point algorithms for independent component analysis. *Neural Networks, IEEE Transactions on*, 10(3):626–634, 1999.

[12] IEEE. *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries*. Institute of Electrical and Electronics Engineers, January 1991.

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

[14] Q. Luo. Forepost website, http://www.cs.wm.edu/semeru/data/ICSE16-FOREPOST.

[15] Q. Luo, A. Nair, M. Grechanik, and D. Poshyvanyk. Forepost: Finding performance problems automatically with feedback-directed learning software testing. *EMSE*, 2016.

[16] I. Molyneaux. *The Art of Application Performance Testing: Help for Programmers and Quality Assurance*. O'Reilly Media, Inc., 2009.

[17] D. Shen, Q. Luo, D. Poshyvanyk, and M. Grechanik. Automating performance bottleneck detection using search-based application profiling. In *ISSTA'15*, 2015.

[18] E. J. Weyuker and F. I. Vokolos. Experience with performance testing of software systems: Issues, an approach, and case study. *IEEE Trans. Softw. Eng.*, 26(12):1147–1156, 2000.

[19] J. Wildstrom, P. Stone, E. Witchel, and M. Dahlin. Machine learning for on-line hardware reconfiguration. In *IJCAI'07*.

[20] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 2005.

[21] P. Zhang, S. G. Elbaum, and M. B. Dwyer. Automatic generation of load tests. In *ASE*, pages 43–52, 2011.