Automating Performance Bottleneck Detection using Search-Based Application Profiling

Du Shen, Qi Luo, Denys Poshyvanyk
Department of Computer Science
College of William and Mary
Williamsburg, Virginia 23185, USA
dshen,qluo,denys@cs.wm.edu

Mark Grechanik
Department of Computer Science
University of Illinois at Chicago
Chicago, Illinois 60607, USA
drmark@uic.edu

ABSTRACT

Application profiling is an important performance analysis technique, when an application under test is analyzed dynamically to determine its space and time complexities and the usage of its instructions. A big and important challenge is to profile nontrivial web applications with large numbers of combinations of their input parameter values. Identifying and understanding particular subsets of inputs leading to performance bottlenecks is mostly manual, intellectually intensive and laborious procedure.

We propose a novel approach for automating performance bottleneck detection using search-based input-sensitive application profiling. Our key idea is to use a genetic algorithm as a search heuristic for obtaining combinations of input parameter values that maximizes a fitness function that represents the elapsed execution time of the application. We implemented our approach, coined as *Genetic Algorithm-driven Profiler* (`GA-Prof`) that combines a search-based heuristic with contrast data mining of execution traces to accurately determine performance bottlenecks. We evaluated `GA-Prof` to determine how effectively and efficiently it can detect injected performance bottlenecks into three popular open source web applications. Our results demonstrate that `GA-Prof` efficiently explores a large space of input value combinations while automatically and accurately detecting performance bottlenecks, thus suggesting that it is effective for automatic profiling.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*; C.4 [**Performance of Systems**]: Performance attributes

General Terms

Performance

Keywords

Application profiling, Performance bottlenecks

ISSTA'15, July 13–17, 2015, Baltimore, MD, USA
© 2015 ACM. 978-1-4503-3620-8/15/07...$15.00
http://dx.doi.org/10.1145/2771783.2771816

1. INTRODUCTION

Improving performance of software applications is one of the most important tasks in software evolution and maintenance [16]. Software engineers make performance enhancements routinely during perfective maintenance [55] when they use exploratory random performance testing [25, 11] to identify methods that lead to performance *bottlenecks* (or *hot spots*), which are phenomena where the performance of the *Application Under Test (AUT)* is limited by one or few components [2, 5]. Developers and testers need performance management tools for identifying performance problems automatically in order to achieve better performance of software while keeping the cost of software maintenance low. In a survey of 148 enterprises, 92% reported that improving application performance was a top priority [84, 70]. The difficulty of comprehending the source code of large-scale applications and their high complexity leads to performance problems that result in productivity losses approaching 20% for different domains due to application downtime [34].

Application profiling is an important performance analysis technique, where an AUT is analyzed dynamically to determine its space and time complexities and the usage of its instructions that reveal performance bottlenecks [73]. Software engineers commonly use *profilers*, *i.e.*, tools that insert instructions into the AUT to obtain frequency, memory usage and elapsed execution time of method calls. When profiling, software engineers perform the following actions: 1) instrument the AUT with a profiler and run the instrumented AUT using some input values and 2) from the collected measurements, they determine what methods are responsible for excessive execution time and resource usage. Simply put, all AUT's methods are sorted in the descending order by their elapsed execution times and top `N` methods on this list are declared bottlenecks and investigated further by engineers. Profilers are widely used at different stages of software development life cycle to analyze runtime performance measurements [33].

A weakness of profiling is that its success for detecting bottlenecks depends on the chosen set of input values for the AUT. A big and important challenge is to profile nontrivial applications with large numbers of combinations of their input parameter values. Many nontrivial applications have complex logic that programmers express by using different control-flow statements, which are often deeply nested. In addition, these control-flow statements have branch conditions that contain expressions that use different variables whose values are computed using some input parameters. In general, it is difficult to choose specific values of input parameters to profile the executions of these applications to obtain bottlenecks.

The full performance analysis can be done if an AUT is profiled with all allowed combinations of values for its inputs. Unfortunately, this is often infeasible because of the enormous number of combi-

nations; for example, 20 integer inputs whose values range from zero to nine give us 10^{20} combinations. To address this problem, *input-sensitive profiling* was introduced where the sizes of their inputs and the values of the input parameters are varied to uncover performance problems in the AUT [86, 20, 51]. Essentially, the standard profiling procedure that we described above is extended with three more steps: 3) study the code of the AUT to understand which methods are specific to certain classes of inputs; 4) construct different combinations of input values to the AUT to find methods involved in bottlenecks, and 5) analyze different execution traces for different combinations of input values to generalize the results. Unfortunately, this procedure is manual, intellectually intensive and laborious, its effectiveness is limited and it increases the cost of application development.

We propose a novel approach for automating performance bottleneck detection using search-based application profiling. Our key idea is to use a *genetic algorithm (GA)* as a search heuristic for obtaining combinations of input parameter values that maximizes a *fitness function* that guides the search process [39]. We implemented our approach, coined as *Genetic Algorithm-driven Profiler* (GA−Prof) that combines a search-based heuristic with contrast data mining [22] from execution traces to automatically and accurately determine bottlenecks. Our paper makes the following noteworthy contributions:

- To the best of our knowledge, GA−Prof is the first fully automatic input-sensitive profiling approach that explores the input parameter space for detecting performance bottlenecks automatically.
- We evaluated GA−Prof on three popular open-source non-trivial web applications. Our results show that GA−Prof effectively explores a large space of possible combinations of inputs while accurately detecting performance bottlenecks.
- GA−Prof and experimental results are publicly available at http://www.cs.wm.edu/semeru/data/ISSTA15-GAProf.

2. PROBLEM STATEMENT

In this section, we provide a background on input-sensitive profiling, discuss peculiarities of execution trace analysis for uncovering bottlenecks and formulate the problem statement.

2.1 Background on Input-Sensitive Profiling

In standard profiling methodology, the input to an application is given as a concrete set of values or as an abstract description from which all values can be generated. Using this input data, profilers instrument and run applications to produce flat or call-graph outputs: the former outputs give a breakdown of resource and time consumption by function while the latter preserve calling contexts by showing caller-callee dependencies among functions. Profilers that are based on the standard methodology are ubiquitous and easy to use; however, their key weakness is based on the assumption that all input data is available in advance, its size is small and finding bottlenecks is orthogonal to the type and the size of the input data. This assumption reduces the effectiveness of profiling for solving performance problems.

Input-sensitive profiling departs from the standard profiling methodology by inferring the size or the type of the input that can pinpoint performance problems in a software application. Consider an example of the pseudocode that is shown in Figure 1. Line 1 specifies that input variables x, y, z and u are initialized with some values. In line 2, the value of the variable v is assigned the result of the

```
input x, y, z, u                                    1
v = A.m( x, y )                                     2
if (v > z) {C.h(B.m(v))} else {D.h(B.m(v)}          3
```

Figure 1: A pseudocode example of input-sensitive profiling.

execution of the method m of A that takes two parameters: x and y and returns their product. In line 3, if the value of v is greater than the value of z, components C and B interact by invoking the method m of B and passing its return value as the parameter to the method h of C. Otherwise, components B and D interact by invoking the method m of B and passing its return value as the parameter to the method h of D. A conclusion that can be inferred from profiling this code depends on specific inputs.

Let us assume that this application is profiled with the input $x \mapsto 5, y \mapsto 2, z \mapsto 3$. The methods of the classes A, C and B are invoked, and the method m of A has the highest elapsed execution time followed by the method m of B. Naturally, these methods are assumed to be bottlenecks; however, while the method m of A and the method m of B are always invoked and they do not depend on the values of the input data, the method m of C and the method m of D depend on the result of the evaluation of the branch condition in line 3. Thus, choosing a different value for the variable z, say 15, may reveal the method m of C and the method m of D as bottlenecks. Also, a different observation is that the input variable u is not used in the invoked methods, and its values do not affect the performance of this program. Thus, knowing how to select input data affects the precision of detecting bottlenecks.

2.2 Analyzing Profile Data for Bottlenecks

Our illustrative example shown in Figure 1 demonstrates two ideas. First, it is not enough to collect performance measurements for some selected input values during profiling – they can be misleading in determining bottlenecks. Consider a situation where a method is invoked many times in different AUT runs for some combinations of input values. In each separate execution trace the total elapsed execution time of the method may not put it on the top of the list of bottlenecks, however, when analyzed across different traces, these methods may be viewed as bottlenecks based on their overall contribution to the total elapsed execution time.

Second, it is important to distinguish bottlenecks based on their *generality* versus their *specificity* for different input values when using input-sensitive profiling. Some methods are computationally intensive, they implement some important requirements and they are invoked for most of the combinations of input data. The method main in Java applications is an example of a generally invoked method. In our illustrative example that is shown in Figure 1, these are the method m of A and the method m of B. Even though profilers easily put these methods on top of the list of bottlenecks, there is often little that software engineers can do to fix these bottlenecks, since these methods are general-purpose. Another example of such general-purpose bottleneck is a logging facility that records execution events on a persistent media. While some improvements can be performed to make a logging facility more efficient, it is often a necessary overhead. Throughout this paper we call these bottlenecks *natural* as opposed to artificially injected or those that result from incorrect implementation of some requirements. The former bottlenecks are rarely fixed while the latter ones are often considered performance related bugs.

On the other hand, specific bottlenecks are methods that are invoked in response to certain combinations of input values. These bottlenecks are most difficult to find, since they involve an exploration of the enormous space of combinations of the input values

that collectively are a small ratio of the total input values space. As it often happens, these bottlenecks remain undetected until the application performance worsens significantly when deployed in the field and used by customers. An important goal of input-sensitive profiling is to increase the specificity of determined bottlenecks by finding a small number of combinations of input values that lead to exposing worsened performance in certain methods of the AUT.

2.3 The Problem Statement

In this paper, we address a fundamental problem of software maintenance and evolution – *how to increase the effectiveness of input-sensitive profiling efficiently*. The root of this fundamental problem is that profiling applications as part of random exploratory performance testing results in a large number of execution traces, many of which are not effective (or useful) in determining specific bottlenecks. Selecting a small subset of input values often results in a skewed distribution of performance measurements, leading to decreased accuracy and low recall for bottlenecks. That is, the output of an input-sensitive profiler is a list of methods that are sorted in the descending order using some performance criteria (e.g., elapsed execution time). If the order of the methods on this list varies significantly from run to run using different input parameter values, the effectiveness of such profiling is low, since engineers cannot easily zero in on performance bottlenecks.

It is equally important to ensure that the exploration of the input parameter space is not done indiscriminately, since many generated input values may not be contributing anything to measuring the effectiveness of the bottleneck detection algorithm. Consider our motivating example in Figure 1, where the input variable `u` may have many values thereby magnifying the input space. Clearly, this parameter does not affect the methods in lines `2-3` and profiling this application with different values for the input variable `u` reduces the efficiency of detecting bottlenecks. Thus, not only is it ineffective to explore the input parameter value space indiscriminately, but it is also highly inefficient (if feasible at all) to profile applications on all combinations of input values. The core problem is how to guide the search process for input values, so that profilers keep extracting useful information for determining and converging on bottlenecks eventually.

Related to the problem of effectiveness and efficiency of input-sensitive profiling is a problem of detecting specific bottlenecks, *i.e.*, those bottleneck methods that become visible only for a small number of combinations of input values. Automatically detecting highly specific bottlenecks is undecidable and very expensive in general. However, multiple evidence show that performance engineers use contrast analysis on collected execution traces, where they analyze correlations among various performance counters with respect to different load profiles [45]. We partially address the problem of determining highly specific bottlenecks in this paper.

3. OUR APPROACH

In this section, we explain key ideas behind our approach, give background on *genetic algorithms*, provide an overview and describe the architecture and workflow of `GA-Prof`.

3.1 Overview of GA-Prof

Search-based algorithms are at the core of `GA-Prof` to automate application profiling for detecting performance bottlenecks. There are two key phases in `GA-Prof`: 1) generating test inputs to automate application profiling and 2) identifying performance bottlenecks.

Automating application profiling. A goal of our approach is to automate application profiling by relying on evolutionary algorithms to explore different combinations of the input parameter values. While exploring these combinations a goal is to maximize a *fitness function* that maps input values to the elapsed execution times of the AUT that is run with these input values. Initially, the instrumented AUT is run with randomly chosen input values; after collecting execution traces and performance measurements for these runs `GA-Prof` evaluates a fitness function for every trace and selects a few sets of inputs that are more likely to lead to performance bottlenecks (*i.e.*, they increase elapsed execution times of the AUT). Subsequently, using the GA terminology, `GA-Prof` evolves to choose combinations of the input parameter values and run the AUT with them. This process is repeated continuously, and the collected profiles are analyzed to detect performance problems in the AUT.

To identify potential performance problems, evolutionary algorithms are used to find *good* inputs that are likely to steer the application's execution towards more computationally expensive paths, especially the paths that contain methods whose executions contribute to performance problems. Conversely, we define *bad* combinations of AUT's inputs as those that take less time for AUT to execute. Note that definition of good and bad inputs may be counter-intuitive. By selecting good combinations of inputs and discarding bad ones, `GA-Prof` keeps evolving the inputs that trigger more intensive workloads in the AUT. The conjecture is that traces that correspond to these good input sets are more likely to be informative at identifying performance bottlenecks.

Identifying performance bottlenecks. Potential performance bottlenecks are detected by using information extracted from multiple traces. Our approach focuses on specific performance problems (not general performance bottlenecks appearing in every application run), which affect AUT's performance significantly. Since the traces are clustered into *good traces* that consume more resources (*e.g.*, execution time) and the *bad traces* that consume less resources, `GA-Prof` marks a method as a performance bottleneck if it has significant contribution to good traces but less significant contribution to bad traces (see Section 3.2.3). A conjecture is that an AUT's specific bottleneck will manifest itself only in a few computationally expensive executions for specific inputs. By extracting these specific performance bottlenecks from collected traces automatically, we make `GA-Prof` favor the highly specific rather than general bottleneck methods.

3.2 Using Genetic Algorithms in GA-Prof

We introduce *Genetic Algorithms (GAs)*, explain why we use GAs and discuss how we utilize GAs in `GA-Prof`.

3.2.1 Background on Genetic Algorithms

GAs are based on the mechanism of natural selection [41] and they use stochastic search techniques to generate solutions to optimization problems. GAs have been widely used in applications where optimization is required but a solution cannot be easily found. The advantage of GA is in having multiple individuals evolve in parallel to explore a large search space of possible solutions. An individual/solution is represented by *chromosome*, *i.e.* a sequence of *genes*.

There are different variations of GAs, but the core idea is that new individuals (*i.e.*, *offspring*) are generated using fitter existing individuals (*i.e.*, *parents*). A pre-defined *fitness function* [41] is used to evaluate the fitness of each individual based on some *fitness value*. Fitter individuals have a better chance to survive. In order to create a new generation, new individuals are created by applying several operators to existing individuals. These operators include (i) a selection operator, (ii) a crossover operator and (iii) a mutation operator.

The selection operator selects *parents* based on fitness values. The crossover operator recombines a pair of selected individuals and generates two new individuals. The mutation operator produces a mutant of one individual by randomly altering its *gene*.

3.2.2 Why We Use Genetic Algorithms in GA-Prof

GAs are based on heuristic and optimization-based search over solution spaces. An alternative to GAs is to use pattern recognition, such as *machine learning (ML)* algorithms. Specifically, our previous work on FOREPOST showed that it is possible to obtain performance bottlenecks for nontrivial applications with a high degree of precision using feedback-directed learning system [32]. With FOREPOST, execution traces for the AUT are collected, they are assigned to different performance classes (*i.e.*, Good and Bad), and then ML algorithms are used to learn the model of the AUT that maps classes of inputs to different performance behaviors of the AUT (*e.g.*, Good and Bad). Our hypothesis is that `GA-Prof` is more effective than FOREPOST because determining what combinations of input values reveal performance bottleneck is inherently a search and optimization problem for which GA algorithms are suited the best. Given the complexity of a nontrivial application, it is difficult to learn a precise model from a limited set of execution traces. We confirm this hypothesis with our experimental results in Section 5.3. In future work, we will explore a combination of GA and ML approaches to the problem of input-sensitive profiling.

3.2.3 Automating Profiling Using GAs

A *gene representation* introduces how we represent AUT's test inputs. For any AUT, one test input is usually a combination of multiple input parameters with specified values. Considering that one *chromosome* is actually a sequence of *genes*, we use chromosome to represent test input. Naturally, each gene of the chromosome represents one input parameter. The value of each gene could be primary types, such as integers, float or boolean, or other well defined types. For a specific type of AUT, *e.g.*, a web-based application, an input test case is a set of URLs. Therefore, we assign an integer ID to each URL so that each gene is has an integer value. Naturally, a chromosome of a sequence of integers actually represents a sequence of URLs.

A *fitness function* evaluates an individual by computing its fitness value. These fitness values are used to guide selection and evolution processes. Since performance problems are more likely to be exposed when it takes longer for the AUT to execute, we favor sets of input values which trigger more computationally intensive runs of the AUT. As a result, the fitness value that we use to evaluate each combination of inputs is measured as the total elapsed time for executing AUT.

A *termination criterion* determines when to stop evolution. Usually, there is a maximum limit for the number of generations, meaning that evolution will be terminated when maximum allowed number of generation is reached, which we choose experimentally. Also, in order to improve the efficiency of the GA, the evolution process can also be terminated when the results converge, *i.e.*, their changes among generations become infinitesimal. In `GA-Prof` we monitor the average fitness value of every individual in one generation and we terminate the evolution when results converge.

Our GA implementation includes the following steps: (i) randomly generate an initial set of AUT's inputs, (ii) use them to execute AUT and collect execution traces, (iii) calculate the fitness value of each execution trace, and (iv) use fitness values to guide the evolution and choose new sets of input values. `GA-Prof` takes in the complete set of input ranges for the subject application and the GA configurations, including crossover rate, mutation rate, fitness function and termination criterion. Then, the algorithm generates an initial population by randomly sampling the gene pool of complete input set. Here is when the evolution begins. The crossover operator takes in a pair of *parent* chromosomes, randomly selects a crossover (cutting) point and exchanges the remaining gene sequence, thus creating two *offsprings* for a new generation. The total number of parent pairs is dependent on crossover rate. After that, the mutation operator takes in an offspring chromosome and changes the value of genes with another value within the specified range, thus generating a mutant of the offspring chromosome. The probability of genes being changed is so-called mutation rate. All newly generated individuals are considered a temporary pool and need to evaluated by the pre-defined fitness function. Each one is assigned with a fitness value and fitter individuals are selected to form a new generation. The selection is based on tournament selection. To select one individual, a tournament is run among a random subset of temporary individuals and the winner is selected, while other individuals are put back to the temporary pool. Multiple tournaments are needed until the new generation meets required population. Thus, a new generation is created. This cycle repeats until termination criterion is satisfied and the final population is returned.

Algorithm 1 `GA-Prof`'s algorithm for automating application profiling

```
1: Inputs: GA Configuration Ω, Input Set I
2: P ← Initial Population(I)
3: while Terminate() == FALSE do
4:    P ← Crossover(P, Ω)
5:    P ← Mutation( P, Ω,I)
6:    for all p ∈ P do
7:       F ← FitnessFunction( p)
8:    end for
9:    P ← Selection( F,P)
10: end while
11: return P
```

The algorithm of automating application profiling is shown in Algorithm 1. `GA-Prof` takes in the complete set of input ranges for the subject application and the GA configurations Ω, including crossover rate, mutation rate, fitness function and termination criterion. In Step 2, the algorithm randomly generates an initial population. Starting from Step 3, the evolution process begins. In Step 4, the crossover operator randomly selects a crossover point and exchanges the remaining genes for selected patent individuals, thus creating two new offspring individuals for a new generation. In Step 5, the mutation operator changes the value of one random gene with another value within the specified range, thus creating a new (updated) individuals if mutation is triggered. In Step 6-8, the fitness of each individual is evaluated using the pre-defined fitness function, which is introduced above. The selection of individuals participating in producing offsprings for a new generation is guided via the fitness values (Step 9). The cycle of Step 3-11 repeats until termination criterion is satisfied. The final population is returned in Step 11 as the algorithm terminates.

3.3 Identifying Performance Bottlenecks

Our goal is to identify specific bottleneck methods automatically. Recall that bottlenecks with a high degree of specificity are more valuable to fix during maintenance than natural or general bottlenecks. Our idea is to detect bottlenecks that are more significant in good execution profiles and are less significant in bad execution profiles.

In order to contrast methods in good/bad execution profiles we rely on the *Independent Component Analysis (ICA)* algorithm that can be used to break large execution traces into sets of orthogonal sets of methods relating to different features of an AUT [42, 32, 31]. ICA algorithm is a computational method that is used to extract components from mixed signals if these components are independent and satisfy the non-Gaussian distribution. ICA has been previously used to address concept location [31] and performance testing problems [32].

The decomposition process is described by the equation $\|\mathbf{x}\| = \|\mathbf{A}\| \cdot \|\mathbf{s}\|$, where $\|\mathbf{A}\|$ is the transformation matrix that is applied to signal matrix $\|\mathbf{s}\|$ to obtain signal mixture matrix $\|\mathbf{x}\|$. In `GA-Prof` context, each row in $\|\mathbf{x}\|$ corresponds to an execution trace and each column corresponds to a method invoked in each trace. Therefore, each element in x_i^j reflects the contribution of method i in trace j. Now we solve this reverse problem by decomposing $\|\mathbf{x}\|$. The elements in $\|\mathbf{s}\|$, s_p^k indicate the contribution of method k to implementing a feature q. Our conjecture is that methods having higher contribution in given features are likely to be involved in performance problems.

$$D_{e_g} = \sqrt{\sum_{i=0}^{N_{M_g}} \sum_{j=0}^{N_{R_g}} (S_{Good}^{ij} - S_{Bad}^{kl})^2} \quad (1)$$

Since execution traces are clustered into good and bad categories, matrix $\|\mathbf{s}\|$ are generated for both of these two clusters, *i.e.* $\|\mathbf{s}_{Good}\|$ and $\|\mathbf{s}_{Bad}\|$. Based on these two matrices, we rely on the Equation 1 to compute specificity weight for each method, where D_{e_g} is the distance for each method, $N_{M_{Good}}$ is the number of good methods, $N_{R_{Good}}$ is the number of features. Since we consider the distance as the weight for each method, we favor potential performance bottlenecks that are significant in good execution traces but not invoked or not significant in bad execution traces. As a result, `GA-Prof` generates a ranked list of methods based on their weights. Higher ranked methods are identified as bottlenecks with a higher degree of specificity.

3.4 GA-Prof's Architecture and Workflow

The architecture of `GA-Prof` is shown in Figure 2. Solid arrows indicate command and data flows between components and the numbers in parentheses indicate the sequence of operations in the workflow.

Initial input value combinations are chosen at random (1). For each of the input sets, AUT's methods are invoked and *Profiler* collects (2) the execution trace for each individual solution. We implemented Profiler component in `GA-Prof` using TPTP framework[1]. The execution traces are passed (3) to *Execution Trace Analyzer*, which uses these traces to produce (4) *Trace Statistics*, containing information about method calls, such as the total number of invocations and the total elapsed self-time for each method. *GA analyzer* computes (5) the fitness value for each input is based on the *Trace Statistics* of its corresponding execution trace. Then the population is evolved using cross-over and mutation operators and new individuals/offsprings are generated (6).

When the termination criterion is satisfied, potential bottlenecks are identified using the last generation of individuals (input combinations). However, it should be noticed that the bottlenecks can be also produced `GA-Prof` for any given generation. *Traces Statistics* are passed (7) to *Trace Clustering*, and all traces are divided into two groups: *good* (8) and *bad* (9) execution traces. Clustering is done based on computing the median value of the elapsed execution time.

[1]https://www.eclipse.org/tptp/

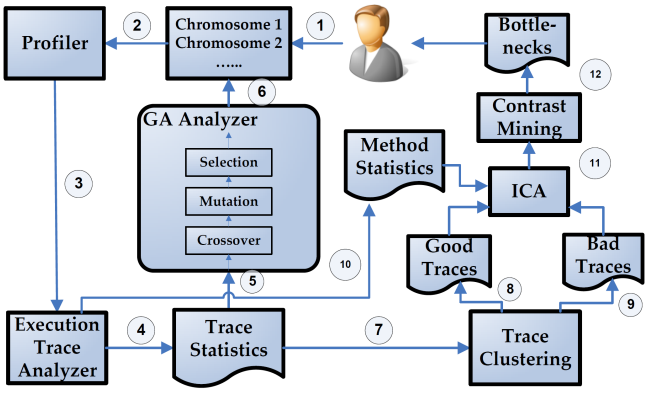

Figure 2: The architecture and workflow of `GA-Prof`.

Combining this with *Method and Data Statistics* produced (10) by *Execution Trace Analyzer*, ICA algorithm computes (11) *Method Weights* for each method using Equation 1. The higher the method's weight in good execution traces the higher the possibility that a method is a AUT's bottleneck. A ranked list of potential bottleneck methods is generated (12) using their weights and is given to the engineer for further evaluation.

4. EMPIRICAL EVALUATION

This section describes the design of the empirical study to evaluate `GA-Prof`. We pose the following three *Research Questions (RQs)*:

***RQ*$_1$:** How effective is `GA-Prof` in finding sets of inputs that steer profiling applications towards more computationally intensive executions?

***RQ*$_2$:** How effective is `GA-Prof` in identifying performance bottlenecks for specific sets of inputs?

***RQ*$_3$:** Is `GA-Prof` more effective than competitive approach in identifying performance bottlenecks?

We introduce the null hypothesis $\boldsymbol{H}_0$ (and consequently alternative hypothesis $\boldsymbol{H}_A$) to evaluate the statistical significance of the difference in the mean value of elapsed execution time between random input and `GA-Prof` generated input for subject applications, designed to answer RQ_1:

$\boldsymbol{H}_0$: There is no statistical difference in the mean values of elapsed execution times triggered by input combinations generated randomly and by `GA-Prof`, for subject applications.

$\boldsymbol{H}_A$: There is statistically significant difference in the mean values of elapsed execution times triggered by input combinations generated randomly and by `GA-Prof`, for subject applications.

In the rest of this section, we first introduce the subject applications used in the study. Then, we describe the methodology, inputs and variables. Finally, we discuss the threats to validity with specific strategies on how we minimized those.

4.1 Subject Applications

We evaluated `GA-Prof` on three subject applications: JPetStore[2], DellDVDStore[3] and Agilefant[4].

[2]http://sourceforge.net/projectss/ibatisjpetstore/
[3]http://linux.dell.com/dvdstore/
[4]http://agilefant.com/

These three applications are all web-based open-source database-centric applications. In these systems, users rely on a web-based *Graphical User Interface (GUI)* front-end to communicate with back-end that accepts URLs as inputs. We deploy JPetStore and DellDVDStore on Apache Tomcat[5] server 6.0.35 and Agilefant on 7.0.47. JPetStore is a Java implementation of the benchmark, PetStore. In our empirical study, we used iBatis JPetStore 4.0.5. The system consists of 2,139 lines of code, 384 methods, 36 classes in 8 packages. JPetStore uses Apache Derby[6] as its back-end database and contains 125 URLs. DellDVDStore is an open-source simulation of an online e-commerce site, which has been used in a number of industrial performance-related studies similarly to JPetStore [46, 45, 14, 18, 72]. DellDVDStore uses MySQL[7] as its back-end database and contains 117 URLs. Agilefant is an enterprise-level backlog product and project management system. It also uses MySQL as its back-end database and contains 124 URLs. We used Agilefant 3.5.1 in our experiments. It consists of 10,848 lines of code, 2,528 methods and 254 classes in 21 packages.

4.2 Methodology

Since we use web-based subject applications, the inputs for these applications are URL requests. For instance, JPetStore has a web-based client-server architecture. Its GUI front-end communicates with the J2EE-based back-end that accepts HTTP requests in the form of URLs. Its back-end can serve multiple URL requests from multiple users concurrently. Each URL exercises different components of the application. For each subject application, we traversed the web interface and source code of these systems and recorded all unique URLs sent to the back-end, in order to obtain a complete set of URL requests.

We define a *transaction* as a set of URLs that are submitted by a single user. To answer RQ_1, we issued multiple transactions in parallel collecting profiling traces and computing the total elapsed execution time for the back-end to execute the transactions. Our goal is to evaluate if `GA−Prof` can automatically find combinations of URLs that cause increase in elapsed execution time. In our experiments, we set the number of concurrent users to five and the number of URLs in one transaction to 50. To answer RQ_2, we randomly selected nine methods in each subject application and injected time delays into them to test whether `GA−Prof` can correctly identify them. In order to answer RQ_3, we chose FOREPOST [32] as competitive approach (see Section 3.2.2). We conducted comparison experiments on subject applications, with artificial delays injected, and compared the effectiveness of both approaches identifying them.

To choose the delay length and methods to inject bottlenecks into, we ran the subject applications without injected bottlenecks and obtained a ranked list of methods. On top of this list we obtained natural bottlenecks. Then, we randomly chose nine methods which all ranked very low on the list of profiled methods to avoid natural bottlenecks of the system and injected artificial delays of five milliseconds into the chosen methods. This delay was chosen experimentally, so that these methods will become bottlenecks for a small subset of combinations of the input values.

Since `GA−Prof` relies on GAs, which are based on randomized algorithms, we had to conduct our experiments multiple times to ensure statistical significance of the results. We followed the guidelines for statistical tests for assessing randomized algorithms [6, 7] when designing the methodology for our empirical study. We repeated the experiments for each subject application for 30 times.

[5]http://tomcat.apache.org/

[6]http://db.apache.org/derby/

[7]http://www.mysql.com/

The experiments for JPetStore and Agilefant were carried out using two Dell PowerEdge R720 servers each with two eight-core Intel Xeon CPUs E5-2609 2.40GHz,10M Cache, 6.4GT/s QPI, No Turbo, 4C, 80W, Max Mem 1066MHz with 32GB RAM that consists of two 16GB RDIMM, 1333 MT/s, Low Volt, Dual Rank, x4 Data Width. The experiments for DellDVDStore were carried out using one Lenovo Y530 laptop with Intel Core2 Duo processor P7350, 2.0 GHz, 3 GB RAM. It typically takes three hours to finish one run for JPetStore and DellDVDStore, and approximately one day for Agilefant. All comparison experiments were conducted on the same experimental platforms to ensure fair comparison.

The GA is implemented using the JGAP library, which provides a collection of methods for a wide range of GA purposes[8]. We used the following GA settings for `GA−Prof`: a crossover rate of 0.3, a mutation rate of 0.1, a population of 30 individuals and a tournament selection of size five. We used the total elapsed time as our fitness function, as described in Section 3.2.3. The evolution is terminated if the results do not improve for ten generations. The maximum number of generations is set to 30 – we chose this value experimentally based on the duration of AUTs' runs and the limits of our experimental platform.

4.3 Variables

Dependent variables include the average number of transactions that subject applications can sustain under the load and the average time that it takes to execute a transaction. There is one main independent variable, that is, bottlenecks. We are interested in two main indicators of the search process: the variance in the position of the bottleneck method relative to the top `N` methods on the list of all profiled methods and the convergence rate to the ultimate position on the list for the bottleneck method among generations of running the GA.

Consider a situation when an engineer is asked to run a profiler on the AUT. When selecting input values randomly, a specific execution path can be taken that may not result in a long elapsed execution time for a bottleneck method to be listed as top `N` method on the profile method list. Depending on the selected input data, this method may enter the top `N` methods on the list and leave it seemingly randomly, as the input data are selected at random. Doing so contributes to the large variance in the position of a given method on the profiled methods list. In contrast, when using a stochastic approach like the GA, we should observe a trend when the variance gets smaller as the bottleneck method moves closer to the top of the list. A long term trend should show this direction for a bottleneck method in our experiments.

4.4 Threats to Validity

A threat to validity for our empirical study is that our experiments were performed on only three open-source web-based applications, which makes it difficult to generalize the results to other types of applications that may have different logic, structure, or input types. However, JPetStore and DellDVDStore were used in other empirical studies on performance testing [46, 45, 72, 18, 14] and Agilefant is representative of enterprise-level applications, we expect our results to be generalizable to at least this type of web-based software applications.

Our current implementation of `GA−Prof` deals with only one type of inputs - URLs, whereas other programs may have different input types. While this is a potential threat, in our opinion, this is not a major one, since `GA−Prof` can be easily adapted to encode inputs of other types. There is no theoretical limitation that prevents

[8]http://jgap.sourceforge.net/

GA−Prof from profiling other types of applications. In order to apply GA−Prof to other applications, one only needs to modify gene representation approach so that GA−Prof recognizes other types of input, such as numbers, strings and booleans. However, GA−Prof currently does not support complex input types, such as inputs with varying lengths. Additionally, it is possible that GA−Prof generates invalid URL sequences through the GA operators. This can be solved by extracting special constraints of inputs for each AUT to ensure generated URL sequences are valid, however, it is currently out of the scope of this paper. Moreover, there may be cases where some methods are naturally computationally intensive, yet they are not performance problems. Our current implementation cannot distinguish these cases with the real performance problems, since we only used elapsed execution time to measure method performance. We are planning on addressing these limitations in the future work.

Artificial delays were injected into randomly chosen methods. This may be a threat for two reasons. First, performance bottlenecks of web-based applications may result from external sources, such as network communication and database queries. Second, real world bottlenecks do not necessarily exist in random spots. However, understanding the locations of performance bottlenecks within applications is currently out of scope for this work.

A different threat is that we perform experiments with a fixed number of users and fixed size of transactions. Using multiple users may lead to discovering new bottlenecks where multithreading, synchronization, and database transactions may expose new types of delays. Experimenting with large workloads is a subject of future work and it is orthogonal to the RQs that we pose, since large workloads will introduce complex interactions among software components, which is outside the scope of this paper.

In spite of these threats, this empirical study design allowed us to evaluate GA−Prof in a controlled setting. Thus, we are confident that the threats have been minimized and our results are reliable.

5. EMPIRICAL RESULTS

This section describes and analyzes the results of our experiments on three software systems in order to answer the research questions stated in Section 4.

5.1 Searching Through Input Combinations

The results for JPetStore with injected artificial delays are shown in the box-and-whisker plots in Figure 3(a), which summarizes the elapsed execution times for the application for given sets of inputs. In this figure, we are only comparing the first and the last generations of the evolution, that is, the resulting running times while profiling JPetStore with random sets of inputs (*i.e.*, the first generation) and evolved input combinations (*i.e.*, the last generation). For the first generation, where each individual is a randomly generated transaction, the average elapsed execution time to execute the system using given sets of inputs is ≈ 4.9 seconds. For the last generation, the average time is ≈ 8.3 seconds, which shows 69.4% increase. The average elapsed times for JPetStore to execute inputs in one transaction across every generation is shown in Figure 4(a). The results demonstrate that GA−Prof is effective in finding combinations of input values that trigger more intensive workloads.

This conclusion is confirmed by the results for DellDVDStore shown in Figure 3(b). The average elapsed execution time is $\approx$ 8.1 seconds in the first generation and ≈ 9.3 seconds in the last generation. We can observe the increase in average elapsed time of approximately 14.8%. This increase is smaller as compared to JPetStore, because DellDVDStore has a relatively smaller and simpler structure, which means that even with randomly generated individuals, significant part of the bottleneck methods are triggered

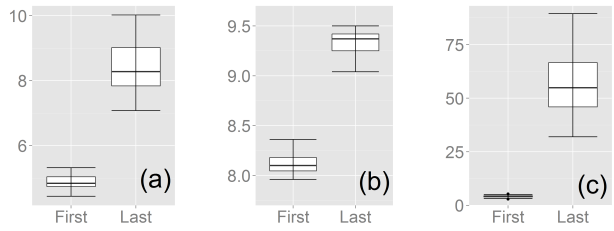

Figure 3: Execution elapsed time measured in seconds for subject AUTs. We compare average elapsed times of each transaction in first and last generations for each application. The x-axis corresponds to the first and last generations, and y-axis corresponds to systems' average elapsed time. The results for all three subject applications are averaged over 30 runs. Subfigure (a), (b) and (c) corresponds to JPetStore, DellDVDStore and Agilefant, respectively.

in the first generation, leaving relatively small part of the search space for GA−Prof to explore. However, for those applications with a large input set (*i.e.*, large search space), we expect to see a significant increase in elapsed time.

This conjecture is confirmed by the results of Agilefant, shown in Figure 3(c). For the first generation, the mean value of elapsed execution time is ≈ 4.13 seconds, and for the last generation, the average time is ≈ 58.22 seconds. The increase in mean value of elapsed execution time is significant because Agilefant is a much larger system as compared to JPetStore and DellDVDStore, and has a much larger input space. Thus, it is more likely that randomly generated combinations of inputs in the first generations may not necessarily be able to focus on the hot spots. Also, the average elapsed times for DellDVDStore and Agilefant to execute one transaction across every generation is shown in Figure 4(b) and 4(c). As the populations evolve, GA−Prof was consistently able to find combinations of inputs that steer applications toward more computationally intensive executions.

To test the null hypothesis $\boldsymbol{H}_{0,JPetStore}$, we applied *t-test* for paired sample mean of the first and last generations from all 30 runs of JPetStore. The p value is $p = 1.5e-21$, allowing us to reject the null hypothesis and accept the alternative hypothesis $\boldsymbol{H}_{A,JPetStore}$ with strong statistical significance ($p < 0.05$) that GA−Prof is effective in finding the combinations of inputs and steering JPetStore towards more computationally intensive executions. Similarly, the *t-test* results for DellDVDStore and Agilefant are $p = 2.9e-30$ and $p = 6.4e-17$. We reject null hypotheses $\boldsymbol{H}_{0,DellDVDStore}$ and $\boldsymbol{H}_{0,Agilefant}$, and accept the alternative hypotheses $\boldsymbol{H}_{A,DellDVDStore}$ and $\boldsymbol{H}_{A,Agilefant}$, thus **positively answering** $\boldsymbol{RQ}_1$ that GA−Prof is effective in finding sets of inputs that steer profiling applications towards more computationally intensive executions.

5.2 Understanding Performance Bottlenecks

As stated in Section 3, GA−Prof ranks methods in a descending order and generates a list of potential bottlenecks. Higher ranking indicates the higher probability of being a performance bottleneck. Since we inserted artificial delays into selected methods, we expect these methods (injected bottlenecks) to be ranked higher on the list. We tracked the ranks of each injected bottleneck across generations and we performed linear fitting analysis in order to understand variation and trends in rankings of known bottlenecks.

The *standard deviation* indicates the variation of rankings across generations. For a given injected bottleneck, we take as input the sequence of its ranks. We calculate the standard deviation at each

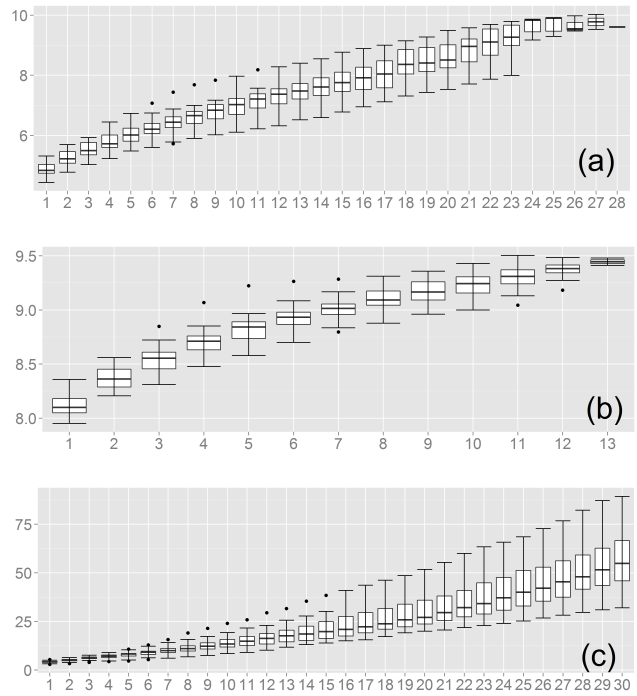

Figure 4: The results for elapsed execution time across every generation for each application, measured in seconds. The x-axis corresponds to generations, and y-axis corresponds to average elapsed time. Subfigure (a), (b) and (c) corresponds to JPetStore, DellDVDStore and Agilefant, respectively.

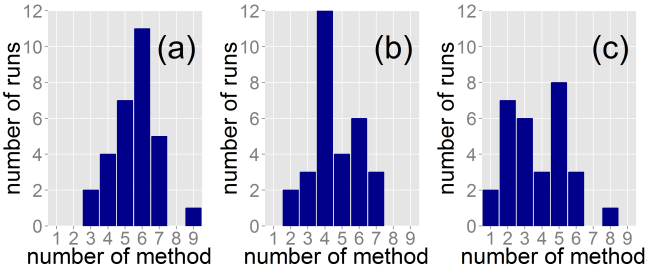

Figure 5: Distribution of the quantity of captured injected bottlenecks. The x-axis corresponds to the number of injected bottlenecks that are captured by one certain `GA−Prof` run. The y-axis corresponds to the number of `GA−Prof` runs. Subfigure (a), (b) and (c) corresponds to JPetStore, DellDVDStore and Agilefant, respectively.

Table 1: Comparing GA-Prof and FOREPOST for detecting performance bottlenecks in JPetStore (JP) and DellDVDStore (DS). All numbers are averaged over multiple runs. "# of Methods" indicates the number of injected bottlenecks that are captured by one certain technique. "Final Ranks" indicates the ranks of injected bottlenecks in the final ranked list.

		GA-Prof		FOREPOST	
				config1	config2
# of Methods	JP	5.6	>	1.8	2.2
	DS	4.6	>	4.2	2.6
Final Ranks	JP	13.78	<	241.67	145.98
	DS	10.94	<	12.67	14.80

generation using the segment of successive five generations, consisting of the ranks at previous two generations, the current generation and next two generations. However, for the first two generations and the last two generations, the value of the standard deviation is assigned to zero because we do not have respective data for generations before and after respectively.

The *linear fitting* reflects the trend of rankings as `GA−Prof` evolves. For each run and method, we take the sequence of rankings as input and perform linear fitting. A negative slope shows that a method is converging to the top of the list; a positive slope shows that a method ends up in lower positions.

If `GA−Prof` yields a negative slope for the fit straight line for one injected bottleneck, `GA−Prof` is considered to "capture" this method. If the slope is positive, `GA−Prof` is considered to "miss" this method. We run `GA−Prof` multiple times for each subject application, and every `GA−Prof` run can capture injected bottlenecks. Figure 5 shows the distribution of the quantity of captured injected bottlenecks. In experiments with JPetStore (see Figure 5(a)), for most of the time, `GA−Prof` can capture five or six bottlenecks. The probability of capturing five or more bottlenecks is 80%. The similar distribution pattern can be observed for DellDVDStore and Agilefant, shown in Figure 5(b) and 5(c). To sum up, the average number (expectation) of injected bottlenecks that `GA−Prof` can capture is 5.6, 4.6, and 3.7 for JPetStore, DellDVDStore and Agilefant, respectively.

One example of `GA−Prof` run on JPetStore is shown in Figure 6. We can see that at most times, injected bottlenecks ranked within top 20 of the descending list, which means that `GA−Prof` 's output is stable and reliable. However, there are some cases where the rank of a bottleneck method is ranked as low as taking the position on the list below 200 and then comes back to the top of the list, for example, Figure 6(b). This phenomenon is expected, since our approach is search-based and it can choose input values for some generations that are not optimal. `GA−Prof` approaches to the target (the bottlenecks) by continuous self-correction. It is expected that sometimes `GA−Prof` experiences some "over-correction", which is when we observe a very low ranking of a method. This is inevitable, however, it is not a concern. The method will come back later on top of the list in future generations, as proved by the figures. As a result, `GA−Prof` will eventually yield a reliable list of methods where injected bottlenecks are ranked on top. This can be demonstrated by the fit linear line (blue dashed lines in the figures). In the example in Figure 6, we observe a negative slope for all nine methods, which means that the ranking of all nine injected bottlenecks are converging to the top of the list as the `GA−Prof` evolves. However, we do not expect that `GA−Prof` would always be able to capture every single injected bottleneck. A positive slope does not always mean that the method is missed. Sometimes a method is ranked on top of the list at every generation, leaving no space for improvement, thus, the slope can not be negative. Sometimes a method may give way to another method but still stay within top positions of the list. These two cases do not impair the reliability of the ranked list at all. In summary, results demonstrate that `GA−Prof` is effective in identifying injected bottlenecks, thus, **positively addressing *RQ*$_2$**.

5.3 Comparing GA-Prof to FOREPOST

Recall from Section 3.2.2 that FOREPOST is the closest competitive approach to `GA−Prof` that uses machine learning to obtain models that map classes of inputs to performance behaviors of the AUT [32]. Like `GA−Prof`, FOREPOST outputs a descending list of potential bottlenecks.

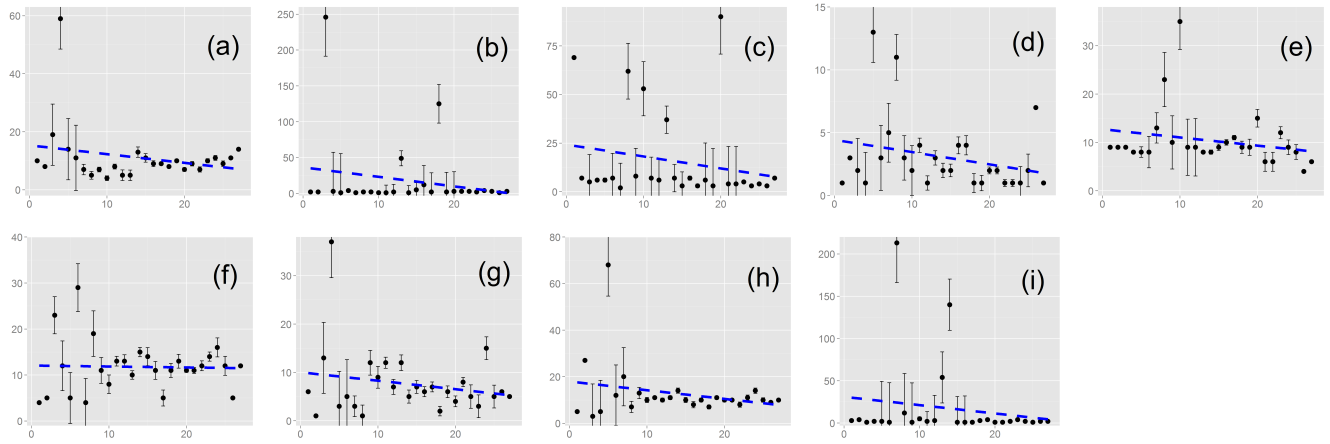

Figure 6: Understanding the trend of ranks of injected bottlenecks. The x-axis corresponds to generations, and y-axis corresponds to the rank of bottlenecks. In each subfigure, the rank of the method is shown in black circles. The standard deviation at each generations is shown in black vertical lines and whiskers. The fit straight line is shown is blue dashed lines.

In our comparison experiments, we used two configurations for FOREPOST. In `config1`, we used four iterations of learning rules and ten execution traces in between. In `config2`, we used four iterations and 15 execution traces. Since FOREPOST experiments are very time-consuming, we repeated FOREPOST experiments five times for only two subject applications: JPetStore and DellDVDStore. The results are shown in Table 1, where we compared the following: 1) how many injected bottlenecks are captured (titled as"# of Method"), and 2) final ranks of injected bottlenecks (titled as"Final Ranks"). Capturing a bottleneck is defined in Section 5.2. By "final ranks", we mean the average of all injected bottlenecks rankings in last generation (`GA-Prof`) or last iteration (FOREPOST).

Table 1 shows that `GA-Prof` was able to capture, on average, 5.6 injected bottlenecks in JPetStore, while FOREPOST captured only 1.8 and 2.2 bottlenecks in two respective configurations. Similarly, for DellDVDStore, `GA-Prof` also captured more bottlenecks. Final ranks are injected bottlenecks' rankings over multiple runs. Smaller numbers represent higher positions in the list, indicating higher probability of being performance problems. For JPetStore, the injected bottlenecks have an average rankings of 13.78 in the list by `GA-Prof`, and 241.67 and 145.98 by FOREPOST. For DellDVDStore, injected bottlenecks are also ranked higher by `GA-Prof`. In summary, `GA-Prof` finds more bottlenecks than FOREPOST, confirming our initial conjecture, and, thus, **positively addressing** $\boldsymbol{RQ_3}$ that `GA-Prof` is more effective than FOREPOST in identifying performance bottlenecks.

6. RELATED WORK

Profiling, a form of dynamic program analysis, is widely used in software testing, such as test generation [23, 74, 50], functional fault detection [8, 66, 88, 9, 19, 49], and non-functional fault detection [81, 20, 62, 21, 56, 35, 83]. Korel provided an approach that generates test cases based on actual executions of AUT to search for the values of input variables, which influence undesirable execution flow, by using function minimization methods [50]. Schur *et al.* provided an tool ProCrawl, which mined an extended finite-state machine as a behavior model and generated test scripts for regression testing [68, 69]. Artzi *et al.* used the Tarantula algorithm to localize source codes which lead to failures in web application by combining the concrete and symbolic execution information [9]. Chilimbi *et al.* provided a tool, HOLMES, to instrument the selected parts of the application, which are likely to contain the root causes of bug reports, and then used statistical analysis to identify the paths that predict the failures strongly by assigning score to these paths [19]. An approach provided by Jiang *et al.* utilizes execution profilers that possibly contain faults to simplify the program and scale down its complexity for in-house testing [44]. But these works only focused on functional faults. Coppa *et al.* provided an approach to measure how the performance scales with increasing size of input, and used it to find out performance faults by analyzing the profiles [20, 24]. Liu *et al.* designed an innovative system, AutoAnalyzer, to identify existence of performance bottlenecks using clustering algorithms and to locate performance bottlenecks by searching algorithm [56]. However, these two papers only paid attention to some specific problems, whereas `GA-Prof` is aimed at exploring and detecting all possible performance bottlenecks. Han *et al.* proposed an approach, StackMine, which applied a costly-pattern mining algorithm on callstack traces, and then extracted impactful subsequences of function calls to help the performance debugging [35]. However, they only extracted callstack patterns that lead to response delay, instead of detecting method-level performance bottlenecks. Our approach applies genetic algorithms to generate test cases, which are likely to reveal performance problems by analyzing execution information.

Genetic Algorithms (GAs) is widely used in many areas of software engineering [36], such as software maintenance [54, 61, 64], textual analysis [65], cloud computing [28, 38] and testing [4, 3, 37, 60, 15, 10, 29, 82, 59, 57, 58]. Test generation is a key point in software testing. Alshahwan *et al.* used dynamically mined value seeding into search space to target branches and generate the test data automatically [4]. To achieve higher branch coverage, McMinn *et al.* used a hybrid global-local search algorithm, which extended the Genetic Algorithm with a Memetic algorithm, to generate the test cases [40, 27]. Harman *et al.* designed an approach by using the dynamic symbolic execution and search-based algorithms to generate test data, which can kill both the first order and higher order mutants for mutation testing [37]. Ali *et al.* provided a systematic review for the search-based test case generation, which built a framework to evaluate the empirical search-based test generation techniques by measuring cost and effectiveness [3]. Briand *et al.* applied GAs to

stress testing. They developed a method for automatically deriving test cases to maximize the probability of critical deadline misses [12]. Iqbal *et al.* used GAs in testing of real-time embedded systems and their empirical study proved effectiveness in detecting system faults [43]. Schwarz *et al.* applied GAs to mutation testing [71], where GAs were used to produce mutations of a program aiming at improving the quality of test case suits. Genetic Algorithms have been successfully used in coverage-oriented testing. Jones *et al.* [48] used GAs to automatically generate test data to execute every branch in several subject applications written in Ada. They used branch predicate as the basis of fitness function. Similarly, Watkins [75] attempted to obtain full path coverage by assigning a small fitness value to an individual that follows previously covered path. Genetic Programming (GP), which is a variation of GAs, considers an individual as the abstract syntax tree of a program that evolves in a genetic way. Fitness is usually measured using results-based approach that seeks to find a program that is best adapted to its specification, such as mutation testing [52] and bug fixing [79, 26, 53, 30]. Wasif *et al.* demonstrated that genetic programming is helpful for software fault prediction [1]. However, these approaches did not consider the non-functional properties.

In Wegener *et al.*'s work [76, 78, 77], GAs were shown to find unknown execution times, which also used GAs for selecting test input data and exposing performance problems. However, they looked for the longest as well as the shortest execution times. Moreover, they did not repeat their experiments to account for the randomness of GAs. Also, their decision about when to stop evolution was rather arbitrary. Finally, `GA-Prof` uses contrast mining to detect specific bottlenecks across different sets of inputs and profiles.

Performance Testing. Finding and fixing performance problems was shown to be even more challenging than identifying functional problems [85]. Thus, one critical goal in performance testing is to automatically generate test cases which may invoke performance problems. Burnim *et al.* provided a complexity testing algorithm for the symbolic test generation tool, to construct the inputs that lead to the worst-case computational complexity of the program [13]. Jin *et al.* extracted efficiency-related rules from 109 real-world performance bugs, and used them to detect performance bugs [47]. Chen *et al.* detected performance anti-patterns (object-relational mapping) from global call and data flow graphs and ranked them [17]. Nguyen *et al.* designed an approach for mining the software regression-causes repositories, and used machine learning algorithms to identify regression causes automatically based on the results of prior tests [63]. Xiao *et al.* propose an approach that predicts workload-dependent performance bottlenecks by using complexity models [80]. Zhang *et al.* proposed an approach for exposing performance bottlenecks using test cases generated by a symbolic-execution based approach [87]. However, unlike `GA-Prof`, they did not utilize execution information to identify performance problems. Pradel *et al.* provided a performance-guided test generation technique to identify pairs of events whose execution time may gradually increase [67]. Grechanik *et al.* proposed FOREPOST, a feedback-directed black-box approach for generating test data, finding performance problems and identifying bottlenecks [32]. Generating test case inputs was guided by rules which were derived from execution traces using a machine learning algorithm. Both FOREPOST and `GA-Prof` approaches are aiming at finding specific combinations of input sets that steer application execution to hot paths. However, `GA-Prof` uses genetic algorithms for exploring a large space of input combinations in the context of automating application profiling. Moreover, our experimental results confirm that `GA-Prof` demonstrate superior results as compared to those by FOREPOST, which is rooted in our original conjecture - it is difficult to learn a precise model from a limited set of execution traces as currently done in FOREPOST. In summary, `GA-Prof` is more effective than FOREPOST because determining what combinations of input values reveal performance bottlenecks is an inherently search and optimization problem for which GAs are best suited for.

7. CONCLUSION

We propose a novel approach for automating performance bottleneck detection using search-based application profiling. Our key idea is to use a genetic algorithm as a search heuristic for obtaining combinations of input parameter values that maximizes a fitness function that represents the elapsed execution time of the application with these input values. We implemented our approach, coined as *Genetic Algorithm-driven Profiler (`GA-Prof`)* that combines a search-based heuristic with contrast data mining from execution traces to accurately determine performance bottlenecks. We evaluated `GA-Prof` in the empirical study to determine how effectively and efficiently it detects injected performance bottlenecks into three popular open source web applications: two popular performance benchmarks and one enterprise-level application. Our results demonstrate that `GA-Prof` effectively explores a large space of the combinations of the input values while automatically and accurately detecting performance bottlenecks.

8. ACKNOWLEDGMENTS

This work is supported by NSF CCF-0916139, NSF CCF-1017633, NSF CCF-1217928, NSF CCF-1218129 grants and Microsoft SEIF. Any opinions, findings, and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

9. REFERENCES

[1] W. Afzal and R. Torkar. On the application of genetic programming for software engineering predictive modeling: A systematic review. *Expert Syst. Applications*, 38(9):11984–11997, 2011.

[2] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *SOSP '03*, pages 74–89, 2003.

[3] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege. A systematic review of the application and empirical investigation of search-based test case generation. *TSE*, 36(6):742–762, 2010.

[4] N. Alshahwan and M. Harman. Automated web application testing using search based software engineering. In *ASE '11*, pages 3–12, 2011.

[5] G. Ammons, J.-D. Choi, M. Gupta, and N. Swamy. Finding and removing performance bottlenecks in large systems. In *ECOOP '04*, pages 172–196. 2004.

[6] A. Arcuri and L. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *ICSE '11*, pages 1–10, 2011.

[7] A. Arcuri and L. Briand. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *STVR*, 2012.

[8] S. Artzi, J. Dolby, F. Tip, and M. Pistoia. Directed test generation for effective fault localization. In *ISSTA '10*, pages 49–60, 2010.

[9] S. Artzi, J. Dolby, F. Tip, and M. Pistoia. Practical fault localization for dynamic web applications. In *ICSE '10*, pages 49–60, 2010.

[10] A. Baars, M. Harman, Y. Hassoun, K. Lakhotia, P. McMinn, P. Tonella, and T. Vos. Symbolic search-based testing. In *ASE '11*, pages 53–62, 2011.

[11] J. Bach. What is exploratory testing? *stickyminds.com*.

[12] L. C. Briand, Y. Labiche, and M. Shousha. Stress testing real-time systems with genetic algorithms. In *GECCO '05*, pages 1021–1028, 2005.

[13] J. Burnim, S. Juvekar, and K. Sen. Wise: Automated test generation for worst-case complexity. In *ICSE '09*, pages 463–473, 2009.

[14] Y. Cai, J. Grundy, and J. Hosking. Synthesizing client load models for performance engineering via web crawling. In *ASE '07*, pages 353–362, 2007.

[15] H. W. Cain, B. P. Miller, and B. J. Wylie. A callgraph-based search strategy for automated performance diagnosis. In *Euro-Par '00*, pages 108–122, 2000.

[16] N. Chapin, J. E. Hale, K. M. Kham, J. F. Ramil, and W.-G. Tan. Types of software evolution and software maintenance. *J. of Softw. Maint. and Evo. R. P.*, 13(1):3–30, 2001.

[17] T.-H. Chen, W. Shang, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora. Detecting performance anti-patterns for applications developed using object-relational mapping. In *ICSE '14*, pages 1001–1012, 2014.

[18] X. Chen, C. P. Ho, R. Osman, P. G. Harrison, and W. J. Knottenbelt. Understanding, modelling, and improving the performance of web applications in multicore virtualised environments. In *ICPE '14*, pages 197–207, 2014.

[19] T. M. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani. Holmes: Effective statistical debugging via efficient path profiling. In *ICSE '09*, pages 34–44, 2009.

[20] E. Coppa, C. Demetrescu, and I. Finocchi. Input-sensitive profiling. In *PLDI '12*, pages 89–98, 2012.

[21] D. C. D'Elia, C. Demetrescu, and I. Finocchi. Mining hot calling contexts in small space. In *PLDI '11*.

[22] G. Dong and J. Bailey. *Contrast Data Mining: Concepts, Algorithms, and Applications*. 1st edition, 2012.

[23] S. Elbaum and M. Hardojo. An empirical study of profiling strategies for released software and their impact on testing activities. In *ISSTA '04*, pages 65–75, 2004.

[24] I. F. Emilio Coppa, Camil Demetrescu. Input-sensitive profiling. *TSE*, 40(12):1185–1205, 2014.

[25] D. R. Faught. Exploratory load testing. *stickyminds.com*.

[26] S. Forrest, T. Nguyen, W. Weimer, and C. Le Goues. A genetic programming approach to automated software repair. In *GECCO '09*, pages 947–954, 2009.

[27] G. Fraser, A. Arcuri, and P. McMinn. A memetic algorithm for whole test suite generation. *JSS*, 2014.

[28] S. Frey, F. Fittkau, and W. Hasselbring. Search-based genetic optimization for deployment and reconfiguration of software in the cloud. In *ICSE '13*, pages 512–521, 2013.

[29] P. Godefroid and S. Khurshid. Exploring very large state spaces using genetic algorithms. *STTT*, 6(2):117–127, 2004.

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

[31] S. Grant, J. R. Cordy, and D. Skillicorn. Automated concept location using independent component analysis. In *WCRE'08*, pages 138–142, 2008.

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

[33] B. Gregg. *Systems Performance: Enterprise and the Cloud*. Prentice Hall Press, 2013.

[34] Y. Group. Enterprise application management survey. 2005.

[35] S. Han, Y. Dang, S. Ge, D. Zhang, and T. Xie. Performance debugging in the large via mining millions of stack traces. In *ICSE '12*, pages 145–155, 2012.

[36] M. Harman. Search based software engineering for program comprehension. In *ICPC '07*, pages 3–13, 2007.

[37] M. Harman, Y. Jia, and W. B. Langdon. Strong higher order mutation-based test data generation. In *FSE '11*, pages 212–222, 2011.

[38] M. Harman, K. Lakhotia, J. Singer, D. R. White, and S. Yoo. Cloud engineering is search based software engineering too. *JSS*, 86(9):2225–2241, 2013.

[39] M. Harman, S. A. Mansouri, and Y. Zhang. Search-based software engineering: Trends, techniques and applications. *CSUR*, 45(1):11:1–11:61, Dec. 2012.

[40] M. Harman and P. McMinn. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *TSE*, 36(2):226–247, Mar. 2010.

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

[42] A. Hyvärinen and E. Oja. Independent component analysis: algorithms and applications. *Neural networks*, 13(4):411–430, 2000.

[43] M. Z. Iqbal, A. Arcuri, and L. Briand. Empirical investigation of search algorithms for environment model-based testing of real-time embedded software. In *ISSTA '12*, pages 199–209. ACM, 2012.

[44] L. Jiang and Z. Su. Profile-guided program simplification for effective testing and analysis. In *FSE '08*, pages 48–58, 2008.

[45] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora. Automatic identification of load testing problems. In *ICSM '08*, pages 307–316, 2008.

[46] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora. Automated performance analysis of load tests. In *ICSM '09*, pages 125–134, 2009.

[47] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and detecting real-world performance bugs. *PIDI '12*, pages 77–88, 2012.

[48] B. F. Jones, H.-H. Sthamer, and D. E. Eyres. Automatic structural testing using genetic algorithms. *Softw. Eng. J.*, 11(5):299–306, 1996.

[49] P. Joshi, C.-S. Park, K. Sen, and M. Naik. A randomized dynamic program analysis technique for detecting real deadlocks. *SIGPLAN Not.*, 44(6):110–120, 2009.

[50] B. Korel. Automated software test data generation. *TSE*, 16(8):870–879, 1990.

[51] T. Küstner, J. Weidendorfer, and T. Weinzierl. Argument controlled profiling. In *Euro-Par '09*, pages 177–184.

[52] W. B. Langdon, M. Harman, and Y. Jia. Multi objective higher order mutation testing with genetic programming. In *TAIC PART '09*, pages 21–29, 2009.

[53] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *ICSE '12*, pages 3–13, 2012.

[54] Z. Li, M. Harman, and R. M. Hierons. Search algorithms for regression test case prioritization. *TSE*, 33(4):225–237, 2007.

[55] B. P. Lientz and E. B. Swanson. *Software Maintenance Management*. Addison-Wesley, 1980.

[56] X. Liu, J. Zhan, K. Zhan, W. Shi, L. Yuan, D. Meng, and L. Wang. Automatic performance debugging of spmd-style parallel programs. *JPDC*, 71(7):925–937, 2011.

[57] P. McMinn. Search-based software test data generation: A survey: Research articles. *STVR*, 14(2):105–156, June 2004.

[58] P. McMinn. Search-based software testing: Past, present and future. In *ICSTW '11*, pages 153–163, 2011.

[59] P. McMinn, M. Harman, K. Lakhotia, Y. Hassoun, and J. Wegener. Input domain reduction through irrelevant variable removal and its effect on local, global, and hybrid search-based structural test data generation. *TSE*, 38(2):453–477, 2012.

[60] A. Memon, A. Porter, C. Yilmaz, A. Nagarajan, D. Schmidt, and B. Natarajan. Skoll: Distributed continuous quality assurance. In *ICSE '04*, pages 459–468, 2004.

[61] B. S. Mitchell and S. Mancoridis. Using heuristic search techniques to extract design abstractions from source code. In *GECCO '02*, pages 1375–1382, 2002.

[62] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Evaluating the accuracy of java profilers. In *PLDI '10*, pages 187–197, 2010.

[63] T. H. D. Nguyen, M. Nagappan, A. E. Hassan, M. Nasser, and F. Parminder. An industrial case study of automatically identifying performance regression-causes. In *MSR '14*.

[64] M. O'Keeffe and M. Ó. Cinnéide. Search-based software maintenance. In *CSMR '06*, pages 249–260. IEEE, 2006.

[65] A. Panichella, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia. How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms. In *ICSE '13*, pages 522–531.

[66] M. Pradel and T. R. Gross. Leveraging test generation and specification mining for automated bug detection without false positives. In *ICSE '12*, pages 288–298, 2012.

[67] M. Pradel, P. Schuh, G. Necula, and K. Sen. Eventbreak: Analyzing the responsiveness of user interfaces through performance-guided test generation. In *OOPSLA '14*, pages 33–47, 2014.

[68] M. Schur, A. Roth, and A. Zeller. Mining behavior models from enterprise web applications. In *FSE '13*, pages 422–432, 2013.

[69] M. Schur, A. Roth, and A. Zeller. Procrawl: Mining test models from multi-user web applications. In *ISSTA '14*, pages 413–416, 2014.

[70] C. Schwaber, C. Mines, and L. Hogan. Performance-driven software development: How it shops can more efficiently meet performance requirements. *Forrester Research*, 2006.

[71] B. Schwarz, D. Schuler, and A. Zeller. Breeding high-impact mutations. In *ICSTW '11*, pages 382–387. IEEE, 2011.

[72] A. Shankar, M. Arnold, and R. Bodik. Jolt: Lightweight dynamic analysis and removal of object churn. In *OOPSLA '08*, pages 127–142, 2008.

[73] A. Srivastava and A. Eustace. Atom: A system for building customized program analysis tools. In *PLDI '94*, pages 196–205. ACM, 1994.

[74] V. Vangala, J. Czerwonka, and P. Talluri. Test case comparison and clustering using program profiles and static execution. In *FSE '09*, pages 293–294, 2009.

[75] A. L. Watkins. The automatic generation of test data using genetic algorithms. In *SQC*, pages 300–309, 1995.

[76] J. Wegener, K. Grimm, M. Grochtmann, H. Sthamer, and B. Jones. Systematic testing of real-time systems. In *EuroSTAR '96*, 1996.

[77] J. Wegener and M. Grochtmann. Verifying timing constraints of real-time systems by means of evolutionary testing. *Real-Time Systems*, 15(3):275–298, 1998.

[78] J. Wegener, H. Sthamer, B. F. Jones, and D. E. Eyres. Testing real-time systems using genetic algorithms. *Software Quality Journal*, 6(2):127–135, 1997.

[79] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *ICSE '09*, pages 364–374, 2009.

[80] X. Xiao, S. Han, D. Zhang, and T. Xie. Context-sensitive delta inference for identifying workload-dependent performance bottlenecks. In *ISSTA '13*, pages 90–100, 2013.

[81] G. Xu and A. Rountev. Precise memory leak detection for java software using container profiling. In *ICSE '08*, pages 151–160, 2008.

[82] Z. Xu, M. B. Cohen, and G. Rothermel. Factors affecting the use of genetic algorithms in test suite augmentation. In *GECCO '10*, pages 1365–1372, 2010.

[83] D. Yan, G. Xu, and A. Rountev. Uncovering performance problems in java applications with reference propagation profiling. In *ICSE '12*, pages 134–144, 2012.

[84] N. Yuhanna. Dbms selection: Look beyond basic functions. *Forrester Research*, 2009.

[85] S. Zaman, B. Adams, and A. E. Hassan. A qualitative study on performance bugs. In *MSR '12*, pages 199–208, 2012.

[86] D. Zaparanuks and M. Hauswirth. Algorithmic profiling. In *PLDI '12*, pages 67–76, 2012.

[87] P. Zhang, S. Elbaum, and M. B. Dwyer. Automatic generation of load tests. In *ASE '11*, pages 43–52, 2011.

[88] S. Zhang and M. D. Ernst. Automated diagnosis of software configuration errors. In *ICSE '13*, pages 312–321, 2013.