STEERING SYSTEM EXECUTION TOWARDS INTENSIVE COMPUTATIONS FOR ADAPTIVE SOFTWARE PERFORMANCE TESTING

by

Alim Ul Gias
Registration no.: Ha-602
Session: 2008-2009

A Thesis
Submitted in partial
fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN SOFTWARE ENGINEERING

Institute of Information Technology
University of Dhaka
DHAKA, BANGLADESH

Steering System Execution towards Intensive Computations for Adaptive Software Performance Testing

ALIM UL GIAS

Approved:

Signature *Date*

Supervisor: Dr. Kazi Muheymin-Us-Sakib

Committee Member: Mohd. Zulfiquar Hafiz

Committee Member: Dr. Kazi Muheymin-Us-Sakib

Committee Member: Dr. Muhammad Mahbub Alam

Committee Chair: Dr. Mohammad Shoyaib

To *Hasne Ara Gias*, my mother
who has always been there, waiting for my late night return from the lab

Abstract

Software performance testing is highly essential for responsive web systems to fix several performance issues such as slow response time, throughput, etc. In such systems, usually a subset of interacting URLs construct the performance test cases. The combination of testing URLs and their corresponding parameters creates numerous execution paths to be tested. Thus exhaustive testing of large scale systems is infeasible with respect to time. The goal is to identify as many as performance bottlenecks within a limited period. This will be possible only if the performance hotspots within the system are predicted in advance.

Time intensive tasks within an application are more prone to performance bugs since those tasks involve huge computations. An adaptive approach can be used to steer the system execution towards time intensiveness. Based on this assumption, work has been done where rules are generated to iteratively choose URLs for a test case. However, rule based system can permanently discard URLs based on insufficient data and often creates conflicting rules.

To address those problems, this thesis proposes a *Bayesian Approach for Steering system execution towards Intensive Computations (BASIC)*. The scheme possesses the set of all URLs ($\mathcal{U}$) where a probability (p_i) is assigned to each URL ($u_i \in \mathcal{U}$). This probability gives an intuition regarding how good a URL will be in worsening the application response time. Initially all the URLs have the same probability to be selected, and a subset of those URLs are chosen using the roulette wheel selection algorithm.

On each iteration, the test cases are executed and information is gathered regarding its response time. This information is used to update the probabilities of the URLs, which are considered in the test case, using Bayes rule. However, an updated probability can turn out to be zero and thus a small value ϵ is added to all the probabilities. This prevents a URL from being permanently discarded as

it will still have a small chance of getting picked. Using probabilities also removes the chance of having conflicts because unlike rule based system, URLs now only have a single selection criteria.

The proposed scheme was implemented and tested on certain parts of two web applications - JPetStore and MVC Music Store. As the goal is to steer application execution towards time intensiveness, the performance has been interpreted in terms of time taken for a varying number of transactions. It is seen that with the increase of transactions, BASIC performs similar to existing rule based scheme without sacrificing any URLs. Moreover, the runtime of BASIC is found to be 65% lower than the runtime of rule based technique and this reduced runtime can help to reveal performance bugs more quickly.

Acknowledgments

First and foremost, I would like to thank Dr. Kazi Muheymin-Us-Sakib for his support and guidance during the thesis compilation. He has been relentless in his efforts to bring the best out of me. I would also like to thank Dr. Mohammad Shoyaib and Mr. Shah Mostafa Khaled for their important feedbacks on the intermediate documents and my presentations. They have always provided their generous helping hand towards me to cross the hurdles that came in my way.

I want to express my gratitude to Mr. B. M. Mainul Hossain for providing his suggestions that was pivotal for my success in this work. He was always there, whenever I asked for his help and showed me the right direction. And last but not the least, I would like to convey my special thanks to Ms. Rubaida Easmin, Mr. Md. Saeed Siddik, Mr. Md. Selim and Mr. Mahedi Mahfuj for helping me through out the course of this thesis. It was their support that always motivated me to give my best and complete this work.

It would have been tough to conduct this research without the support of certain grants and for providing those, I am grateful to the following –

- The Ministry of Science and Technology, Bangladesh under the National Science and Technology Fellowship No–39.012.002.01.03.019.2013-280(116).
- The University Grant Commission, Bangladesh under the Dhaka University Teachers Research Grant No–Regi/Admn-3/2012-2013/13190.

List of Publications

Publications during the Master's period

- Alim Ul Gias and Kazi Sakib. An Adaptive Bayesian Approach for URL Selection to Test Performance of Large Scale Web-Based Systems. In Thirty Sixth International Conference on Software Engineering (ICSE) Companion Proceedings, Hyderabad, India. Pages 608–609. ACM/IEEE. June 2014. Accepted.

- Alim Ul Gias, Mirza Rehenuma Tabassum, Amit Seal Ami, Asif Imran, Mohammad Ibrahim, Rayhanur Rahman and Kazi Sakib. A Formal Approach to Verify Software Scalability Requirements using Set Theory and Hoare Triple. In Proceedings of Sixteenth International Conference on Computer and Information Technology (ICCIT), Khulna University, Bangladesh. Pages 7–12. IEEE. March 2014.

- Alim Ul Gias, Rayhanur Rahman, Asif Imran and Kazi Sakib. TFPaaS : Test-first Performance as a Service to Cloud for Software Testing Environment. International Journal of Web Applications, vol. 5, no. 4, pp. 153–167. December 2013.

- Alim Ul Gias, Asif Imran, Rayhanur Rahman and Kazi Sakib. IVRIDIO: Design of a Software Testing Framework to Provide Test-first Performance as a Service. In Proceedings of Third International Conference on Innovative Computing Technology (INTECH), London, UK. Pages 520–525. IEEE. August 2013.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Software performance testing is vital for highly responsive systems like online social networking or E-commerce sites to maintain customer satisfaction. In case of large scale systems, this testing is performed adaptively where test cases are updated on each iteration. Multiple URLs are considered in each script depending on intuitions that those URLs will expose performance bugs. The goal of this research is to propose a technique that will aid in choosing URLs to reveal performance bottlenecks. To achieve such goal, several issues such as URL characterization and selection need to be addressed. These issues are distilled into multiple research questions and a novel solution is proposed by answering those.

1.1 Motivation

The scope of exchanging information is increasing day by day with the development of Internet which is now being used as an alternative to expand businesses by many companies. Many business transactions are being performed through various E-commerce sites. It is convenient for majority of the people to purchase an item from a virtual shop rather than buying it from the store. These E-commerce sites also help different manufacturing businesses to purchase materials required to make goods at lower costs and without the need to speak with someone directly. Since the Internet is always active, companies can also increase their profits by being able to sell products for twenty four hours throughout the week. The success of any E-commerce sites mostly rely on customer satisfaction and one of the ways of achieving it could be minimizing the system response time to minimize customer frustration [1].

It is essential to test a web application for its responsiveness in terms of its

stability that is the capability of handling significant workload. Such test scheme is termed as load testing [2], that can help to build performance within the system structure. Load testing provides the information about the system's behavior when handling specific user requests asking for a number of transactions simultaneously. The system's response times for each of those transactions can also be monitored with load tests within a specific period of time. This type of monitoring can provide useful information regarding application bottlenecks before the application become more problematic.

Software testers construct performance test cases that include actions (GUI interactions or method calls) and data corresponding to these actions. In a web based system, these actions and data are represented by URLs and inputs to those URLs. It is difficult to develop effective performance test cases within a short period as it requires significant amount of time to test all combinations of actions and data. Sometimes it is even not possible to exhaustively test web-based systems like eBay, Flipkart, etc. due to having millions of URLs. Such systems make it difficult to locate bugs because testing majority of the system may not find any bugs whereas testing the other part could find many. Thus choosing the areas that should be tested are extremely difficult for large-scale systems. In such cases, usually the testing is done based on testers' intuition and experience which is referred as Intuitive or Exploratory testing [3] .

Exploratory testing is a scheme where testers run different test cases (initially those are randomly created) and observe the application's behavior. This observation can provide an intuition regarding properties of test cases which guide applications towards more interesting behaviors. These behaviors are likely to reveal bugs which are performance bottlenecks in case of performance tests. To automate such process, an adaptive system is required that will gain knowledge by observing the current iteration and use that knowledge in the next iteration to find bottlenecks. This can be termed as adaptive software performance testing and one

of its major challenges is to steer the application towards behaviors that expose performance issues. The goal of this research is to propose one such automated steering scheme that will assist in adaptive software performance testing.

1.2 Research Questions

One of the common assumptions of adaptive software performance testing is that part of the application having slower response time could reveal more performance bugs. Thus, slower response time could be considered as a criteria that could expose performance bottlenecks. So the main goal of a steering scheme should be guiding application's towards time intensive region and this leads to the primary research question:

RQ How can the application's execution be steered towards its time intensive region?

The way of achieving the primary goal is to decide a criteria to add URLs in test cases on each iteration of the adaptive process. That is by observing the execution time for test cases of the current iteration, the scheme should learn that which URLs are helping test case execution times to achieve higher values. Based on that learning each URLs should receive a score regarding their inclusion being a "good" decision. If the score of a URL is higher, it will mean that the URL is more capable to guide the execution towards time intensiveness. One way of assigning this score could be using probability which leads to a breakdown of primary research question:

- Can we quantify that, choosing a URL is being a good decision by means of a probabilistic approach?

On each iteration, multiple test cases have to be created and URLs should be added to those cases. These URLs will be picked based on their probability that

indicate how well they can worsen the system response time. There have to be a way to associate those probability with the possibility of a URL being added in a test case. This leads to our final research question:

- How can we associate the probability of being "good" to the possibility of a URL to get selected in a test case?

All these questions are addressed and this has become the key to all achievements in this research. These achievements are presented in the next section that combined together offer a novel technique to steer system's execution towards computational intensiveness.

1.3 Research Contributions

This research presents a novel solution named as Bayesian Approach for Steering towards Intensive Computations (BASIC) to address the questions that have been raised in the previous section. BASIC is an adaptive scheme which iteratively updates the test cases for performance testing. In each case, multiple URLs of the system are considered depending on intuitions (probability of being time intensive) that those URLs will expose the performance bugs. As the testing goes on the scheme adaptively updates its knowledge regarding a URL using Bayes rule. In a nutshell, the major contributions of BASIC include:

- BASIC offers a novel technique to characterize a URL as time intensive. It uses the Bayes theorem to assign a probability to each of the URL that provides an insight of that URL being being capable of worsening system response time. Initially execution profiles are generated using the test case which are created randomly. These execution profiles are clustered into "good" (slow response time) or "bad" (fast response time) profiles. These

profiles are used to calculate the prior probabilities which are later used to calculate the posterior.

- BASIC uses the roulette wheel selection algorithm [4] to select URLs based on their probabilities of time intensive. Based on the posterior probabilities that will be calculated previously, an area in the roulette wheel is assigned to each of the URLs. If the probability is higher then the possibility of getting selected will also be higher. If the probability of one URL becomes zero a small value is added with that URL and thus no URL will be permanently discarded. This is actually good and this URL will now have a chance to get selected again and may achieve a higher probability.

- It has been demonstrated both theoretically and experimentally that the runtime of BASIC is lower than the existing rule based scheme [5] for steering system execution. The rule based technique also encompasses the problem of creating stalemate situations by generating conflicting rules. This has been illustrated by an experiment that such situation can certainly occur in a rule based system whereas it is not possible in case of BASIC. Moreover, the time growth in each iteration for both the techniques have been illustrated. The key benefit of BASIC is that without discarding any URLs permanently, it performs similar in worsening the system response time like the rule based scheme.

1.4 Organization of the Thesis

Rest of the thesis is organized as follows:

- Chapter 2 provides a taxonomy to give an insight of software testing. It also discusses about the core concepts of performance testing and exploratory testing. Lastly, it illustrates when performance and exploratory testing needs

to be combined and how this can be automated by means of adaptive software performance testing.

- Chapter 3 discusses about the researches which are related to this work. Initially, different learning techniques that have been used in functional testing have been highlighted. The chapter is concluded by presenting different schemes used to automate performance testing.

- Chapter 4 presents the Bayesian Approach for Steering towards Intensive Computations (BASIC). The architecture and work flow of BASIC along with its algorithms has been illustrated. Moreover, it has been theoretically shown that how BASIC prevents URLs being permanently discarded, removes conflicts and minimizes the complexity of rule based system.

- Chapter 5 presents the experimental setup and the results which are obtained by running the experiment. It is shown that BASIC shows similar performance in terms of worsening response time compared to rule based system.

- Chapter 6 includes the concluding remarks about this research. Initially a discussion have been included regarding BASIC and different experimental results and how those results can be improved. Finally, this document is concluded by providing the scope for further research that can be conducted based on BASIC.

Chapter 2

Background

Software testing [6, 7] is an indispensable part of any software development process. The test process appears in many ways based on the size of the software project, available budget, and the desired or acceptable quality of the end product. A variety of techniques and tools exists which are applied during the software test process. These techniques include test design, automation, application, optimization, and adequacy assessment. Adaptation of one or more of these techniques and tools not only leads to improved quality but also reduces cost. Thus to ensure better software quality, this phase requires more emphasis. A significant amount of research effort has already been spent in the development and evaluation of techniques and processes used in software testing. This chapter presents an insight of the current techniques of software testing by providing a taxonomy. Moreover, a viewpoint for adaptive software performance testing in terms of performance testing and exploratory testing is also discussed.

2.1 Taxonomy of Software Testing

Software testing methods are usually divided into black, white and gray box testing. These approaches differ in terms of the view point of a tester over the system structure in times of designing test cases. Figure 2.1 presents a taxonomy of software testing where common software techniques are classified as one of black, white or gray box testing. In order to understand the nature of these techniques, it is required to know the properties of the approaches by which those are performed. These approaches are discussed in the following subsections.

Figure 2.1: A Taxonomy of Software Testing Based on Source Code Exposure

2.1.1 Black Box Testing

With the enhancement of project domain, software testing process is getting more complicated day by day. Black box testing [8, 9] is used to overcome the complexities of structural testing which often increases exponentially with the system volume. Thus, this testing technique is favored and widely used by the software engineers to reveal bugs for large scale systems. This is a technique that allows to test any system without having specific knowledge of the interior mechanism of the application. The tester is oblivious to the system architecture and does not have access to the source code. When conducting a black box test, a tester will interact with the system's user interface by providing inputs and examining outputs without knowing how and where the inputs have worked.

The principle concept for a successful black-box test is to check for the functional correctness of the system. This testing ignores the internal functionalities of a system and focuses solely on the outputs generated in response to selected inputs and execution conditions. Black box testing execute the test cases which exercise all functional requirements of a program. It finds errors like incorrect or missing functions, interface errors, errors in data structures or external database access, and performance errors. In Figure 2.1, the techniques classified under black box testing are intended to find those errors.

Black Box Testing method is applicable to all levels of the software testing process namely unit testing, integration testing, system testing, and acceptance testing. The black box testing comes more into use when the testing level is higher and hence making the box bigger and more complex. This testing is particularly suitable for large code segments and able to efficiently expose bugs within the system. It clearly separates user's perspective from the developer's one through visibly defined roles. Using black box testing, large numbers of moderately skilled testers can test the application without having any knowledge of implementation,

programming language or execution platforms. On the other hand, black box testing has limited coverage since only a selected number of scenarios are actually tested. This testing is inefficient in a sense that the tester only has limited knowledge about an application. Moreover, the test cases are difficult to design since the tester cannot target specific code segments or error prone areas.

2.1.2 White Box Testing

White box testing [10, 11], also known as glass testing, is the detailed exploration of internal logic and structure of the code. In order to conduct white box testing on an application, the tester requires possessing knowledge of the internal working of the code. The tester needs to have a look inside the source code and detect which chunk of the code is behaving improperly. It is a strategy for software debugging in which the tester has excellent knowledge of how the program components interact.

This method can be used for Web services applications but rarely practical for large scale systems and networks. White box testing is highly efficient in detecting and resolving problems, because bugs can often be identified before they cause trouble. Besides, white box testing is widely considered as a security testing (the process to determine that an information system protects data and maintains functionality as specified) method that can be used to validate whether code implementation follows intended design, to validate implemented security functionality, and to disclose exploitable vulnerabilities.

The test cases formulated for white box testing check certain sets of execution paths. In white box testing, all independent paths in a module is have to be executed at least once. All loops should be executed and internal data structures is required to be exercised for ensuring their validity. As the tester has knowledge of the source code, it becomes very easy to identify which type of data can help in testing the application effectively. Extra lines of code can be evacuated that can

trigger hidden defects through white box testing. Moreover, this process can help in achieving maximum coverage during test scenario creation since the tester has sufficient knowledge about the code.

White box testing requires skilled tester to complete the testing procedure. This increases the cost of the development process as skilled testers should be paid accordingly based on their work merit. Sometimes it is impossible to look into every part of an application to find out hidden errors that may create problems as many paths may go untested. It is difficult to maintain white box testing as the use of specialized tools like code analyzers and debuggers are required.

2.1.3 Gray Box Testing

Gray Box testing [12, 13] is an approach to test a system with limited knowledge of its internal mechanism. In the time of testing an application, having sound knowledge of the system domain always gives the tester an advantage over someone with limited domain knowledge. Unlike black box testing, where the tester only tests the user interface, in this technique the tester has access to design documents and databases. This knowledge helps the tester to prepare better test plans that involves test data and scenarios.

Gray box testing is based on the internal structures of code and algorithms for preparing the test cases more than black box testing but less than white box testing. This process does significant tasks when conducting integration testing between two modules of code written by two different developers, where only interfaces are exhibited for test. This is only possible as gray box testing does not require the whole source code to be hidden, testers can develop effective test cases on certain aspects. The method can also incorporate reverse engineering to determine boundary values.

Being an combination of black and white box, gray box testing offers the

benefits of both. A tester can design effective test scenarios especially around communication protocols and data type handling based on the extra information that are provided. The test is done from the viewpoint of a user instead of a designer. Gray box testing is well suited for web applications that encompasses a distributed network. In such cases, white box testing may not be possible due to the absence of source code. Black box testing is also not used due to just contract between customer and developer. Thus it is better to use gray box testing to reveal the performance bugs for web applications.

Due to the unavailability of the source code, there is no scope to inspect the internal implementation and thus test coverage is limited in gray box testing. The tests can be redundant if the software designer has already run a test case. Testing every possible input stream is also infeasible because usually it would take huge amount of time. This is the reason many program paths could go untested especially for large scale applications.

2.2 Performance Testing

Performance testing [1, 14] is performed using the black box approach and is a non-functional type of testing. Even though by being a non-functional testing, performance testing is considered almost as important as functional testing before launching any business critical applications. This is because performance has become extremely important in this rapidly growing technological era. It is extremely important to keep the clients happy by means of a highly responsive system. Studies have revealed that user conversion rate of an application starts to suffer when its response time exceeds more than 5 seconds [1]. Dissatisfied users may never come back again and can also influence others for not to use the application. This may result in large proportionate business loss and thus it is highly

[1] agileload.com/agileload/blog/2012/11/09/application-performance-testing-basics

essential to test the system performance for avoiding such situations. This is the reason for which performance testing is drawing the attention of the organizations and getting a high priority in testing phase. Currently organizations prefer to conduct an entire performance testing over their applications before going to production phase. This ensure that their application is fast enough in responding the intended client's requests.

In order to accurately measure performance, there are a number of key indicators that must be taken into consideration. These indicators are classified into two types: service-oriented (availability and response time) and efficiency-oriented (throughput and utilization). Service-oriented indicators measure how well an application is providing a service to the end users. As said earlier, it can be measured by lack of availability which is significant because many applications can have a substantial business loss for even a small out-of-service period. Efficiency-oriented indicators measure how well an application uses the resources provided to it. It can also be measured by means of throughput which is defined as the rate of number of requests served successfully.

All these indicators, taken together, can provide an accurate indication of how an application is performing and its specific impacts. However, the most critical measure among the performance indicators is response time. It is defined as the total time from the first byte of the request sent by the user to the last byte received to complete any interaction. Even the most appealing user interface cannot guarantee user satisfaction if the responsiveness of the application is not up to the mark. It is required to study the average response time of similar systems and rigorously test the current system to see whether the performance is satisfactory to keep the clients happy.

In case of large-scale system it is usually not possible to exhaustively test the whole application to find performance bugs. In such cases, testers use their intuitions and experiences to identify regions which could be prone to performance

bottlenecks. Such scheme is referred to exploratory testing. For effective software performance testing, it is essential to integrate the idea of exploratory testing to quickly reveal performance issues. The next section highlights the properties of exploratory testing.

2.3 Exploratory Testing

Exploratory testing can be defined in terms of three words: simultaneous learning, test design and test execution [3]. This means that testers should learn about the interesting properties of the system, that could expose bugs, along with running the test cases. This learning should be applied when a new test case will be designed. This is also referred as intuitive testing and mostly depends on the creativity and the experience of the testers. This kind of testing is usually effective when the application under test is too large and cannot be tested exhaustively. In such cases, the testers initially create a test case based on their experience and try to guide the test case execution towards bug prone region.

The primary advantage of exploratory testing is that important bugs could be identified quickly during the execution time with a comparatively less amount of preparation. This is because the approach tends to progress more intellectually based on tester's creativity and experience rather than the process of executing predefined test scripts. The testers can use logical reasoning based on the results of previous test cases to guide their future testing on the fly. However, this technique is highly depended on the ability of individual testers and the outcome will vary from person to person.

In terms of performance testing, performance bottlenecks are the issues for which testers are usually looking for. Considering web systems like social networking and E-commerce sites, it is hard to reveal bugs within a short amount of time. Due to this, testers create an initial test case that include system URLs

and observe the execution time. One of the goal of performance testing is to make the system response time worse to locate performance bugs. Testers look for those URLs that plays a vital role in worsening the system response time. Those URLs become the clue to reveal bottlenecks within the system. This whole notion of performance and exploratory testing can be automated by means of Adaptive Random Testing (ART) which is discussed in the next section.

2.4 Adaptive Random Testing

Adaptive Random Testing (ART) is an enhancement of random testing which depicts a group of algorithms for generating random test cases. It has been illustrated that these test cases have greater fault-detection capacity than simple random testing [15]. This is because the test cases that were initially prepared, get continuously updated on each iteration. The research trends in adaptive random testing is shown in Figure 2.2 and one of those trend is to use the concept in testing software performance. This actually combines the idea of performance and exploratory testing and automates the whole procedure. The learning is usually performed on analyzing the execution traces generated by test case on each iteration.

ART originates from Random testing (RT) which is one of the most commonly used software testing procedure to assess the software reliability and detect software errors. ART by being an improvement of RT, yields better results than original RT in terms of fault detection capability. Nonetheless, a few work have been done on analyzing the effectiveness of ART in different test scenarios. Previous studies have shown that adaptive random testing can use fewer test cases than random testing to detect the early application failure. The researchers of [16] presented a group of ART algorithms for generating test cases which are based on fixed-size-candidate set [17] and restricted random testing [17]. The authors used

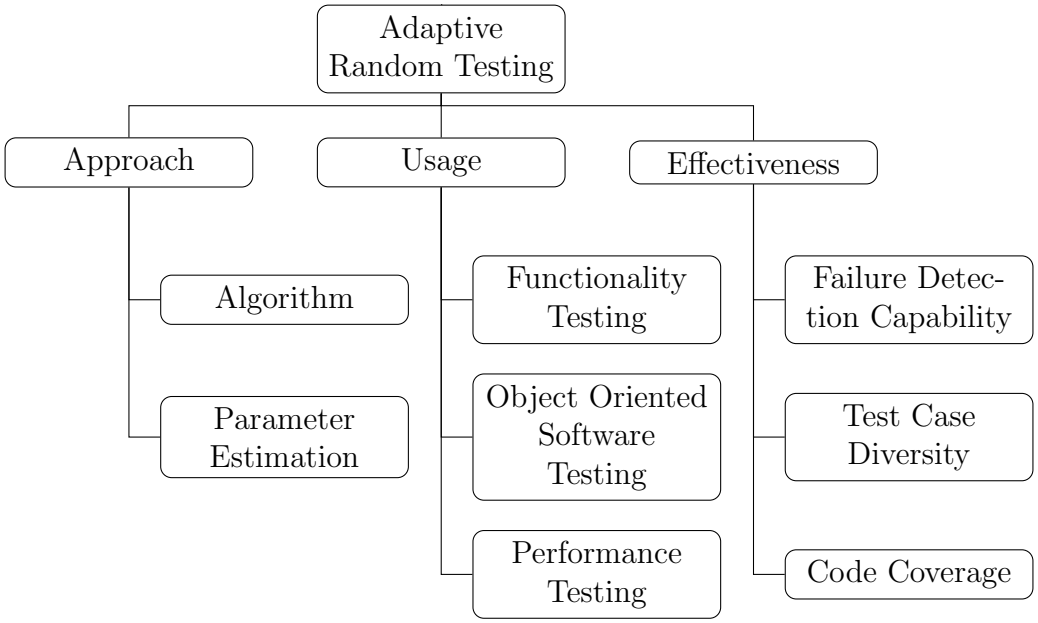

Figure 2.2: A Taxonomy of Current Research Trends on Adaptive Random Testing

an empirical methodology for comparing the effectiveness of test cases obtained by their proposed methods and random selection strategy. Experimental data illustrate that the ART algorithms cover the possible combinations at a given time more quickly than the random selection scheme. Moreover, their proposed scheme often detected more faults earlier and with fewer test cases in simulations.

The adaptive random testing procedure has recently been improved and examined in the research conducted by Zhou et al [18]. The authors have utilized adaptive random testing to prioritize regression test cases based on code coverage by applying the concepts of Jaccard Distance (JD) [19] and Coverage Manhattan Distance (CMD) [20]. The problem with previous research was that the researchers did not provide the comparison between JD and CMD to clarify their relative advantages. Moreover, the frequency information of CMD was also not utilized. This gap was filled by the current research [18] by first inspecting the fault-detection capabilities due to using frequency information for test case prioritization and later the comparison of JD and CMD was also made. Their research findings illustrate that "coverage" is more useful than "frequency" though the latter can sometimes

complement the former. Moreover, these results show that CMD is superior to JD. It is also revealed that, for certain faults, the conventional "additional" algorithm which is widely accepted as one of the best algorithms for test case prioritization, could perform much worse than random testing on large test suites.

Authors in [21] have evaluated and compared the performance of adaptive random testing and random testing from another perspective, which is code coverage. It has been shown in various studies that higher code coverage can bring a higher failure-detection capability. It can also improve the effectiveness in estimating software reliability. The researchers performed a number of experiments depending on two categories of code coverage criteria that include structure-based and fault-based coverage. Adaptive random testing can attain higher code coverage than random testing with the same number of test cases. Their experimental results demonstrate that ART have a better failure-detection capability than random testing. Moreover, it provides higher effectiveness in measuring software reliability and confidence in the reliability of the software even if no failure is detected [21].

Recently, work has been done on using the resources of cloud to run test cases adaptively [22]. Using the resources of cloud provides better results in finding the bottlenecks of the application under test. However, the application under test should be instrumented to keep track of system's traces. The components which are responsible for this task are the Test Task Controller, Estimator, Test Report Handler and repository store system traces. An illustration of the adaptive testing procedure is shown in Figure 2.3. The test controller will pass the commands to the execution environment and traces of those executions will be stored. These traces will be used by the estimators to provide guidelines for performing the test in the next iteration. The results of the test script execution will be sent to the report handler who will forward it to the client.

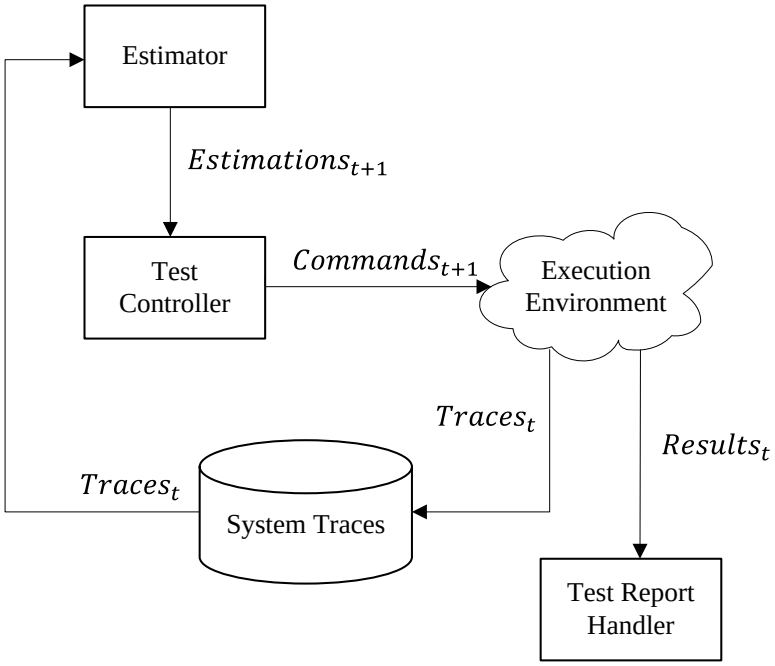

Figure 2.3: Illustration of Adaptive Software Performance Testing in the Cloud

2.5 Summary

This chapter discusses about the preliminaries of software testing. A taxonomy of software testing is provided based on three widely used approaches of software testing: black, white and gray box testing. A discussion has been provided on performance and exploratory testing which needs to be combined to test large scale systems. Finally a notion of adaptive random testing is provided that can aid in automating the combination of performance and exploratory testing.

Chapter 3

Literature Review

To the best of author's knowledge, guiding system's execution towards time intensiveness is yet to be fully addressed by the researchers. However, plenty of related work can be found that used learning strategies to locate functional bugs and to automate performance testing. This chapter presents the state of the art works regarding software testing methodologies that involves machine learning paradigm. Initially methods that involves functional testing using machine learning techniques are discussed. Later the techniques used to automate software performance testing, based on data analyses are also discussed.

3.1 Machine Learning Techniques in Functional Testing

This section discusses about functional testing schemes that used different learning strategies. The works mainly involves analyzing program execution or stack traces. Work have also been done on static code analysis to locate functional bugs. These techniques are discussed in the following subsection.

3.1.1 Analyzing Program Execution

Dickinson et al [23] have used the concept of cluster analysis in observation based testing [24]. Observation based technique analyzes a subset of program execution traces to find failures within the execution. For choosing a subset it can use filtering schemes like random sampling techniques. The authors proposed a novel scheme to filter out a subset to reduce the tester's effort. In their work they have

used the cluster filtering analysis proposed by Podgurski et al [25]. A cluster filtering scheme includes two steps: choosing a clustering technique and a strategy for sampling from clusters. The authors have used Agglomerative hierarchical clustering [26] to fulfill the first step. As a sampling strategy they have tested multiple ones including: simple random sampling, one-per cluster sampling, adaptive sampling and n-per cluster sampling strategy. Their results suggests that adaptive sampling strategy is the most effective one in the context of cluster filtering to detect execution failures.

3.1.2 Static Code Analysis

The authors in [27] presented a way to detect object usage anomalies. They have highlighted the fact that the objects often requires to follow a protocol for interacting with each other. One such protocol is that they need to follow a specific sequence of method call. For example, one has to call the method "hasNext()" before calling the "has()" method. These protocols may not be always specified and their violations can lead to program failures. The authors have proposed an approach that takes code examples to automatically infer legal sequences of method calls. The resulting patterns after the code analysis was used to detect anomalies within the system. Their work is novel in a sense that this work is the first fully automatic defect detection approach that learns and checks method call sequences.

3.2 Automated Performance Testing

This section presents different methodologies used to automate software performance testing. Some of those work includes designing tools and frameworks for software performance testing. Work has also been done on performance debugging

and performance regression testing using statistical methods. These schemes are discussed in the following subsections.

3.2.1 Tools and Frameworks

A tool for analyzing Java performance named as Jinsight EX has been presented in [28]. It has been designed to address the problem of diagnosing performance and memory problems of Java program that involves the analysis of large and complex data sets describing a program's execution or dynamic behavior. The tool is aimed to help the user to find the right organization of the data unveiling useful information and to work with the data through a long and uncertain fact-finding process.

Jinsight EX incorporates the concept of execution slices to provide the flexibility of organizing execution information along lines that are useful for the current analysis task. A defined execution slice can be used to represent an area of interest throughout the analysis. A number of execution slices can also be utilized as a user-defined dimension to measure resource usage more precisely with respect to user-defined aspects of a program execution. In addition, the tool allows slices to be organized into workspaces that assists in larger analyses in the context of certain phases. This feature allows to develop alternative hypotheses that can be used for further experimentation and leads towards incremental discovery.

However, some scopes for further enhancement of Jinsight EX have been identified by the authors. These includes incorporating information from additional sources such as high level design information from software design tools to define execution slices. Moreover, they have shown their concern on addressing large stack traces especially in case of longer running programs.

3.2.2 Performance Debugging

An approach of Performance Debugging for large-scale distributed systems including multiple communicating components is presented by Aguilera et al [29]. Debugging of those large systems becomes much harder when systems are composed of "black-box" components. This means the source code are not available. The main goal of their research is to design tools that help programmers isolate performance bottlenecks in black-box distributed systems. The tools was supposed to be based on application-independent passive tracing of communication between the nodes in a distributed system, combined with off-line analysis of these traces. The authors proposed two very different algorithms for inferring the dominant causal paths through a distributed system for tracing performance issues. Their solution approach involves three phases: Exposing and tracing communication, Inferring causal paths and patterns and Visualization of result.

3.2.3 Statistical Methods

An automated framework for performance regression testing is presented in [30]. The authors proposed a mechanism that is based on the gathering knowledge from previous regression test record and correlating different performance matrices with new test results. The paper also discusses the difference of their works with existing techniques [31, 32]. These techniques work on assuming the Service Level Objectives (SLO) that can be effectively used to determine an anomaly occurrence. However, the problem is that during the development the SLO is usually not available. The authors have compared their test results against correlations among performance metrics taken from performance regression testing repositories.

Jiang et al [33] presented an approach that uses load testing logs to discover execution sequences and deviations from those sequences. They have highlighted

that though techniques like [34, 35, 36] are capable of detecting programming errors, these are not applicable for load testing. The authors focuses on automatically detecting performance problems by analyzing load test results. This was done by executing load test on same operations for a large number of times. The execution sequences are identified by log decomposition and log abstraction [37]. The test results are analyzed to find deviations from the execution sequence and deviated behaviors are marked as anomalies. This approach could be useful in saving time for a software tester but the challenge here is huge amount of logs are required to be created by load test for a multiple number of times.

A method for automatically capturing, indexing, clustering, and retrieving system history from a running system is presented in [38]. The authors showed that the construction of an effective signature that captures the essential state of an enterprise system is nontrivial. The naive approach yields poor clustering and retrieval behaviors. However, good results are obtained with an approach based on their prior successful usage of statistical methods for capturing relationships between low-level system metrics and high-level behaviors. They also demonstrated how the representation and clustering can be used across different sites to diagnosis for problems. Their experimental validation is conducted on a realistic testbed with injected performance faults, and on production traces from several weeks of operation of a real customer facing Web application in our organization.

Ammons et al [39] illustrated a novel approach to find bottlenecks in large scale software systems. In their approach, for each kind of profile (for example, call- tree profiles), a tool developer implements a simple profile interface that exposes a small set of primitives for selecting summaries of profile measurements and querying how summaries overlap. Next, an analyst uses a search tool, which is written to the profile interface and thus independent of the kind of profile to find bottlenecks. In one case study, after using Bottlenecks for half an hour, they reported 14 bottlenecks in IBM's WebSphere Application Server. By optimizing

some of these bottlenecks, they obtained a throughput improvement of 23% on the Trade3 benchmark [40]. The optimizations include novel optimizations of Java Enterprise Edition and Java security, which exploit the high temporal and spatial redundancy of security checks.

Woodside et al [41] presented a way to keep the performance model up-to-date as the software system changes. They have demonstrated an Extended Kalman Filter [42] can be constructed for a performance model and investigated its usefulness on an example. Moreover, the authors have identified potential pitfalls to its application, which this is the first use of Kalman filtering to track changes in a performance model. The filter converged remarkably quickly following a step change in the system parameters, which was the condition investigated here. A convenient aspect of the Kalman Filter is its capability to fuse data of different types, such as delay, throughput, and utilization, as demonstrated in their experimental section of this paper. Since the filter depends on sensitivities, and bottlenecked systems have low sensitivity to parameters away from the bottleneck, one drawback of their work could be a bottlenecked system or model might get "stuck" away from the correct values of parameters.

An automated approach to automatically detect possible performance problems in a load test is presented in [43]. The authors have identified the issue that , filtering performance problems from load test log is time consuming because of the very large size of log files. Their proposed problem detection technique is the extended version of [33] where dominant execution patterns are used as an indicator for performance problem analysis. Their proposed scheme used the results of prior load tests as an informal baseline and compared against the current test result. The performance of prior run and current run was summarized and an statistical analysis was conducted. If the current run possessed scenarios that followed different response time distribution than the baseline (prior load test response time distribution), the current run was identified as problematic.

3.2.4 Rule based Approach: FOREPOST

To the best of authors' knowledge, the most recent scheme for finding performance issues in large web-based systems is FOREPOST [5]. FOREPOST relies on the assumption that guiding a system's execution towards time intensive region will expose performance bugs. Thus it has a guiding strategy which is based on rules. On each iteration FOREPOST analyzes the execution profiles generated by the test cases to create rules. These rules are used in the next iteration to take decision on URL inclusion.

FOREPOST starts the adaptive testing procedure with m out of n number of URLs. After collecting and analyzing execution profiles, it generates rules saying which URLs (g) must be considered in the next iteration and which must not (b). In the next iteration, those g URLs are kept and the rest $(m - g)$ positions are filled up from the $(n - b)$ URLs. If no rule is generated, m URLs are again selected randomly.

Problems with FOREPOST are that rule generation is a complex process and can create conflicting situations. Moreover, if a URL is labeled as bad, it will be permanently discarded whereas it could turn out to be a good one if it were picked with another set of URLs. These issues are related to uncertainties which should be resolved to expose more accurate performance bugs.

3.3 Summary

This section presents some of the works that are either based on learning scheme or directly performance testing. The most closest work to our research goal, FOREPOST is also presented in this chapter. The flaws of FOREPOST is also illustrated and the next Chapter focuses on resolving those problems of FOREPOST.

Chapter 4

BASIC: Bayesian Approach for Steering system execution towards Intensive Computations

Reviewing the literature shows that guiding an application execution towards its time intensive region can expose performance bugs. Work has been done based on this assumption using certain rules which have been adaptively generated during the execution. However, rules involve a complex generation procedure, can be misleading and create conflicting situations by being rigid in nature. An alternative way around to resolve those issues could be using a probabilistic approach where probabilities will be updated adaptively. This section proposes one such steering scheme named *Bayesian Approach for Steering system execution towards Intensive Computations (BASIC)*. Moreover, the potential benefit of the proposed technique is also illustrated.

4.1 Overview of BASIC

BASIC is a methodology of adaptively adding or removing URLs from test scripts which is performed based on a learning scheme. The scheme is based on Bayes theorem that calculates the posterior probability of choosing a URL being a good decision given some observed evidence. To be more precise, URLs are added or removed from test scripts on the basis of information i.e. probability of being a good decision, gathered from analyzing execution profiles generated by test scripts of previous iteration.

Let us consider that there are $\mathcal{U} = \{u_1, u_2, u_3, ..., u_n\}$ URLs in a web-based system. Each URL (u_i) has a probability (p_i) of being good associated with

it which is mapped by the function $\phi : \mathcal{U} \rightarrow \mathcal{P}$. In the current iteration of the adaptive testing procedure, $\mathcal{S} \subset \mathcal{U}$ URLs have been chosen to create w test scripts. These w test scripts will produce w profiles. Each of these profiles will have a execution time and based on their execution time, those can be labeled as one of the element from the set $\mathcal{L} = \{good, bad, unassigned\}$. Based on the occurrence of each $u_i \in \mathcal{S}$ in one of the profiles, their probabilities are being updated using Equation 4.1-

$$P(l_j = good|u_i) = \frac{P(u_i|l_j = good)P(l_j = good)}{\sum_{l_j \in \mathcal{L}} P(u_i|l_j)P(l_j)}, \forall_{u_i \in \mathcal{S} \text{ and } l_j \in \mathcal{L}} \tag{4.1}$$

Where,

$$P(u_i|l_j) = \frac{\text{frequency of } u_i \text{ in } l_j \text{ profiles}}{\text{frequency of profiles labeled as } l_j}, \forall_{u_i \in \mathcal{S} \text{ and } l_j \in \mathcal{L}} \tag{4.2}$$

$$P(l_j) = \frac{\text{frequency of } l_j \text{ profiles}}{\text{total number of profiles } w}, \forall_{l_j \in \mathcal{L}} \tag{4.3}$$

BASIC uses the updated probability of these $|S|$ URLs to choose URLs in the next iteration. On each iteration certain amount of profiles are generated which changes the belief regarding each URL of being time intensive. The main aspects of BASIC is discussed below from a high point of view:

- BASIC selects each URL using the Roulette wheel selection algorithm similar to [4]. The probability of being "good" is used to associate the probability of selection for each URL i.e a proportion of the wheel is assigned to each URL based on their probability. Thus there is no chance of having a conflict during URL selection like FOREPOST. Moreover, it is very much likely that URLs with higher probability will get selected though URLs with lower probability may also get the chance. This is actually good as those particular URLs may turn out to be time intensive by being selected with a other set

of URLs. This is how BASIC eradicates the problem of getting permanently discarded.

- After collecting w amount of profiles $\mathcal{E} = \{E_1, E_2, E_3, ..., E_w\}$ BASIC labels each of those profiles. A profile can be defined as a set of URLs with cardinality m, execution time and label i.e $E_k = \{\{u_1, u_2, u_3, ..., u_m\}, t_k, l_k\}$. The labeling is performed based on clustering scheme using execution time. Initially the mean (μ_t) and standard deviation (σ_t) is sorted out $\forall_{t_k \in E_k}$. Each profile E_k then receives the value for its label (l_k) from $\mathcal{L}$ based on the following condition-

$$l_k \in E_k = \begin{cases} good & \text{if } t_k \geq (\mu_t + \sigma_t) \\ bad & \text{if } t_k \leq (\mu_t - \sigma_t) \\ unassigned & \text{otherwise} \end{cases} \tag{4.4}$$

- At each iteration, BASIC updates its belief regarding which URLs are actually "good" i.e the probability of being time intensive. This is performed using Equation 4.1. During the update procedure, the posterior probability may turn out to be zero. Thus it will not have the chance of being selected in the next selection due to not having any area in the roulette wheel. This scenario is exactly the same as of when a rule is generated in FOREPOST stating that a URL is bad. To avoid such situation, after updating the probability of all URLs in the current iteration, a negligible value is added with all those probabilities. This discards the chance of having a zero probability and the probabilities remain relatively same as the value is added with all.

Rest of the chapter discusses about steps of BASIC methodology in details. Section 4.2 illustrates the algorithm for URL selection. Section 4.3 discusses about the profile classification scheme. Section 4.4 illustrates the evidence collection methodology for updating the belief. Section 4.5 discusses in details about the

overall BASIC approach. Section 4.6 gives an insight for further enhancement of BASIC scheme to achieve accurate results. And finally Section 4.7 illustrates that how BASIC resolves the problem associated with a rule based system.

4.2 URL Selection

The URLs for adding in test scripts are chosen using a roulette wheel or fitness proportionate selection algorithm. Each URL has a probability assigned with it. This acts as the basis on which a URL will get an area on the wheel. URLs with high probabilities will have larger areas which eventually increase their chance of getting picked. This selection algorithm is illustrated in Algorithm 1. The symbols used in the algorithm are presented in Table 4.1.

Table 4.1: Symbols for URL Selection Algorithm

$\mathcal{U}$	The set of all URLs of the System under Test
u_i	An individual element of the set $\mathcal{U}$
$\mathcal{P}$	The set of probabilities corresponding to each URL
p_i	An individual element of the set $\mathcal{P}$
ϕ	The mapping function between $\mathcal{U}$ and $\mathcal{P}$
$\mathcal{C}$	The set of cumulative probabilities corresponding to each URL
c_i	An individual element of the set $\mathcal{C}$

The Algorithm takes the set of all URLs ($\mathcal{U}$), their associated probabilities ($\mathcal{P}$) and the mapping function (ϕ) as an input. It returns a URL in a typical roulette wheel fashion. For getting multiple URLs one have to call the algorithm for multiple times. The algorithm can return a particular URL more than once. Thus during the test script creation one should use a Set to store URLs for ignoring repetitive values.

Initially the scheme calculates the total probability of the set $\mathcal{P}$. Besides it creates a temporary set $\mathcal{C} = \{c_1, c_2, c_3, ..., c_n\}$ to store the cumulative probability up to each URL (u_i). Finally, a random number is generated between 0 and total probability and the URL (u_i) is returned for which the cumulative probability (c_i)

Algorithm 1 URL Selection

Input: The set of URLs $\mathcal{U}$, the set of URLs' associated probabilities $\mathcal{P}$ and the mapping function $\phi : \mathcal{U} \rightarrow \mathcal{P}$

Output: A URL $u_i \in \mathcal{U}$

1: **Begin**
2: $sum = 0$
3: $\mathcal{C} \leftarrow \emptyset$
4: **for each** Probability $p_i \in \mathcal{P}$ **do**
5: $\quad sum+ = p_i$
6: $\quad c_i = sum$
7: $\quad \mathcal{C} \cup \{c_i\}$
8: **end for**
9: $number = Random[0, sum]$
10: **for each** URL $u_i \in \mathcal{U}$ **do**
11: $\quad$ **if** $number < c_i$ **then**
12: $\quad\quad$ **return** u_i
13: $\quad$ **end if**
14: **end for**
15: **End**

is greater than the random number. This algorithm can be called for a multiple number of times to select URLs for a test case.

4.3 Clustering Execution Profile

Executing a test script created by the URL Selection algorithm generates an execution profile. After collecting a set of execution profile $\mathcal{E} = \{E_1, E_2, ..., E_w\}$, it is required to classify those to estimate the impact of involved URLs towards time intensiveness. Each execution profile $E_k \in \mathcal{E}$ will contain a set of m URLs, profile execution time t_k and an empty label l_k. This label will have an assigned value from the set of label $\mathcal{L} = \{good, bad, unassigned\}$ after classifying all profiles based on the execution time. This procedure is illustrated in Algorithm 2 and its corresponding symbols are described in Table 4.2.

Initially the mean (μ_t) and standard deviation (σ_t) for all the profile execution time is being calculated. All the profiles are then clustered using these two values. If the execution time of a profile is greater than or equals to $\mu_t + \sigma_t$ then it is labeled

as a good profile. The profile is labeled as a bad one if the opposite happens i.e. the profile execution time is less than or equals to $\mu_t - \sigma_t$. If none of these two cases hold, then the profile is labeled as unassigned.

Table 4.2: Symbols for Execution Profile Clustering Algorithm

$\mathcal{E}$	The set of current execution profiles
E_k	An individual element of the set $\mathcal{E}$
t_k	The profile execution time of the profile E_k
l_k	The profile label of the profile E_k
$\mathcal{L}$	The set of all labels
μ_t	The mean of all profile execution time
σ_t	The standard deviation of all profile execution time

Algorithm 2 Execution Profile Clustering

Input: The set of execution profiles $\mathcal{E} = \{E_1, E_2, ..., E_w\}$ where each execution profile (E_k) will contain a set of m URLs, profile execution time (t_k) and an empty label (l_k) i.e. $E_k = \{\{u_1, u_2, ..., u_m\}, t_k, l_k\}$, The set of labels $\mathcal{L} = \{good, bad, unassigned\}$

Output: $l_k = good \vee bad \vee unassigned, \forall_{l_k \in E_k}$

1: **Begin**
2: Find the mean μ_t and standard deviation σ_t, $\forall_{t_k \in E_k}$
3: **for each** Execution Profile $E_k \in \mathcal{E}$ **do**
4: **if** $(t_k \in E_k) \geq (\mu_t + \sigma_t)$ **then**
5: $(l_k \in E_k) \leftarrow good$
6: **else if** $(t_k \in E_k) \leq (\mu_t - \sigma_t)$ **then**
7: $(l_k \in E_k) \leftarrow bad$
8: **else**
9: $(l_k \in E_k) \leftarrow unassigned$
10: **end if**
11: **end for**
12: **End**

Figure 4.1 shows the different regions corresponding to each label. These regions will change depending on the values of the profile execution times of the current iteration. Form the figure it could be seen that the extreme values will be marked as either good or bad and other values will fall in unassigned zone.

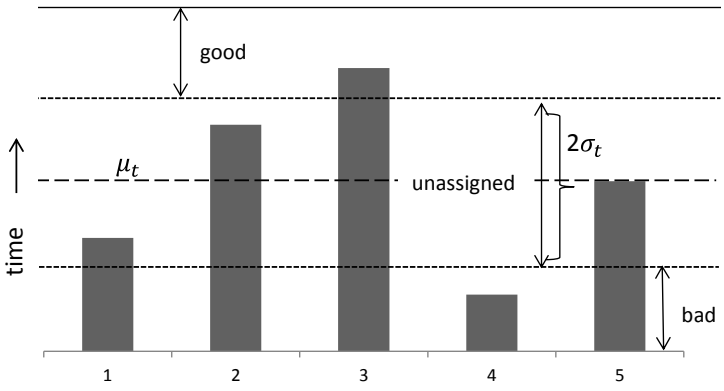

Figure 4.1: Illustration of the clustering scheme

4.4 Evidence Collection

BASIC needs to update its knowledge regarding the probability of a URL being time intensive. The scheme updates its knowledge based on the collected evidence during each iteration. The evidence includes-

- The probability of each event i.e. label (good, bad or unassigned)
- The prior probabilities

As soon as the task of clustering execution profile ends, BASIC starts to collect those evidence. Initially it calculates the probability for each label and then the prior probabilities.

Algorithm 3 illustrates the steps involving label's probability calculation. The symbols used in the algorithm are presented in Table 4.3.

Table 4.3: Symbols for Calculate Label's Probability Algorithm

$\mathcal{E}$	The set of current execution profiles
E_i	An individual element of the set $\mathcal{E}$
$\mathcal{L}$	The set of all labels
l_k	Any element from the set $\mathcal{L}$

The algorithm takes the set of execution profile $\mathcal{E}$ all the labels $\mathcal{L}$ as inputs. Each of those profiles must be labeled using one of the element of the set $\mathcal{L}$. Initially the algorithm counts the number of profiles labeled as good or bad or

unassigned. The probability of each label is then achieved by dividing those numbers by the total number of profiles.

Algorithm 3 Calculate Label's Probability

Input: The set of execution profiles $\mathcal{E}$ and the set of labels $\mathcal{L}$
Output: $P(l_j), \forall_{l_j \in \mathcal{L}}$

1: **Begin**
2: $numberOfGoodProfiles = 0$
3: $numberOfBadProfiles = 0$
4: $numberOfUnassignedProfiles = 0$
5: **for each** $E_i \in \mathcal{E}$ **do**
6: **if** $l_i \in E_i = good$ **then**
7: $numberOfGoodProfiles++$
8: **else if** $l_i \in E_i = bad$ **then**
9: $numberOfBadProfiles++$
10: **else**
11: $numberOfUnassignedProfiles++$
12: **end if**
13: **end for**
14: $P(l_j = good) = \frac{numberOfGoodProfiles}{|\mathcal{E}|}$
15: $P(l_j = bad) = \frac{numberOfBadProfiles}{|\mathcal{E}|}$
16: $P(l_j = unassigned) = \frac{numberOfUnassignedProfiles}{|\mathcal{E}|}$
17: **End**

Table 4.4: Symbols for Prior Probability Calculation Algorithm

$\mathcal{S}$	The set of URLs used in the current iteration
u_i	An individual element of the set $\mathcal{S}$
$\mathcal{E}$	The set of current execution profiles
E_i	An individual element of the set $\mathcal{E}$
u_j	An individual URL of the URL set within E_i
$\mathcal{L}$	The set of all labels
l_j	Any element from the set $\mathcal{L}$

The procedure for prior probability calculation is shown in Algorithm 4 and the symbols used in the algorithm are presented in Table 4.4. The algorithm takes the set of URLs ($\mathcal{S}$) used in the current iteration as an input. Moreover, the total number of profiles labeled as good, bad or unassigned is also passed to the algorithm. The algorithm also requires the set of execution profiles ($\mathcal{E}$) and the set of labels ($\mathcal{L}$). At the end the probability of each URL, within $\mathcal{S}$, residing in one of good, bad or unassigned profile is provided.

Algorithm 4 Prior Probability Calculation

Input: The set of URLs in the current iteration $\mathcal{S} \subset \mathcal{U}$, the set of execution profiles $\mathcal{E}$, $numberOfGoodProfiles$, $numberOfBadProfiles$, $numberOfUnassignedProfiles$ and the set of labels $\mathcal{L}$

Output: $P(u_i|l_j), \forall_{u_i \in \mathcal{S} \text{ and } l_j \in \mathcal{L}}$

1: **Begin**
2: **for each** $u_i \in \mathcal{S}$ **do**
3: $inGoodProfiles_i = 0$
4: $inBadProfiles_i = 0$
5: $inUnassignedProfiles_i = 0$
6: **for each** $E_i \in \mathcal{E}$ **do**
7: **for each** $u_j \in E_i$ **do**
8: **if** $u_i = u_j$ **then**
9: **if** $l_k \in E_i = good$ **then**
10: $inGoodProfiles_i++$
11: **else if** $l_k \in E_i = bad$ **then**
12: $inBadProfiles_i++$
13: **else**
14: $inUnassignedProfiles_i++$
15: **end if**
16: **end if**
17: **end for**
18: **end for**
19: **if** $numberOfGoodProfiles \neq 0$ **then**
20: $P(u_i|l_j = good) = \frac{inGoodProfiles_i}{numberOfGoodProfiles}$
21: **else**
22: $P(u_i|l_j = good) = 0$
23: **end if**
24: **if** $numberOfBadProfiles \neq 0$ **then**
25: $P(u_i|l_j = bad) = \frac{inBadProfiles_i}{numberOfBadProfiles}$
26: **else**
27: $P(u_i|l_j = bad) = 0$
28: **end if**
29: **if** $numberOfUnassignedProfiles \neq 0$ **then**
30: $P(u_i|l_j = unassigned) = \frac{inUnassignedProfiles_i}{numberOfUnassignedProfiles}$
31: **else**
32: $P(u_i|l_j = unassigned) = 0$
33: **end if**
34: **end for**
35: **End**

Initially, it is calculated that the number of times each URL (u_i), in the current URL set ($\mathcal{S}$), is residing in a good, bad or unassigned profile. The prior probabilities are then calculated by dividing those numbers by the total number of profiles labeled as good, bad or unassigned. If any of the profile number turn out to be 0, then that corresponding prior probability is also set to 0.

4.5 BASIC Algorithm

The BASIC scheme is illustrated in Algorithm 5 that will guide the application's execution towards its time intensive region. The symbols used in the algorithm is presented in Table 4.5. On each iteration, the scheme updates it knowledge regarding which URL will be time intensive. It takes the set of all URLs ($\mathcal{U}$) and the set of their associated probabilities ($\mathcal{P}$) as inputs.

Table 4.5: Symbols for BASIC Algorithm

$\mathcal{U}$	The set of all URLs of the System under Test
$\mathcal{P}$	The set of probabilities corresponding to each URL
p_i	An individual element of the set $\mathcal{P}$
ϕ	The mapping function between $\mathcal{U}$ and $\mathcal{P}$
$\mathcal{S}$	The set of URLs used in the current iteration
u_i	An individual element of the set $\mathcal{S}$
$\mathcal{E}$	The set of current execution profiles
E_k	An individual element of the set $\mathcal{E}$
t_k	The profile execution time of the profile E_k
l_k	The profile label of the profile E_k
$\mathcal{L}$	The set of all labels
l_j	Any element from the set $\mathcal{L}$
r_i	Posterior probability of the i^{th} URL

The algorithm starts by creating w number of profiles which will later be analyzed. Each of those profiles are created by choosing m number of URLs from the set $\mathcal{U}$. All the URLs for the current iteration are stored in $\mathcal{S}$. The URLs are chosen using the roulette selection algorithm. After collecting the profiles, those are clustered and labeled as good, bad or unassigned using the clustering

Algorithm 5 BASIC Algorithm

Input: The set of URLs $\mathcal{U}$, the set of URLs' associated probabilities $\mathcal{P}$ and the mapping function $\phi : \mathcal{U} \rightarrow \mathcal{P}$

Output: Updated $\mathcal{P}$

1: **Begin**
2: $\mathcal{E} \leftarrow \emptyset$
3: $\mathcal{S} \leftarrow \emptyset$
4: **while** w profiles are not created **do**
5: Select m URLs $\{u_1, u_2, ..., u_m\}$ using Algorithm 1.
6: $\mathcal{S} \cup \{u_1, u_2, ..., u_m\}$
7: Run test script with these m URLs to create execution profile $E_k = \{\{u_1, u_2, ..., u_m\}, t_k, l_k\}$
8: $\mathcal{E} \cup \{E_k\}$
9: **end while**
10: Label all execution profiles in $\mathcal{E}$ using Algorithm 2.
11: Calculate each label's probability and prior probabilities using Algorithm 3 and 4.
12: $prevTotal = 0$
13: **for each** $u_i \in S$ **do**
14: Calculate $r_i = P(l_j = good|u_i)$ using corresponding label's probability and prior probabilities.
15: $r_i += \epsilon$
16: $prevTotal += p_i$
17: **end for**
18: **for each** $u_i \in \mathcal{S}$ **do**
19: $p_i = prevTotal \times \frac{r_i}{\sum r_i}$
20: **end for**
21: **End**

algorithm. Moreover, the prior probabilities are also calculated using the algorithm for prior probability calculation.

Based on the label's probability and the prior probabilities, BASIC updates its knowledge of what it belief as good. That is the probability of each URL being time intensive is updated. Moreover, the area of corresponding URLs, i.e the URLs selected in the current iteration ($\mathcal{S}$), is readjusted in the roulette wheel. The areas of other URLs, i.e not chosen in this iteration, are kept the same. This phenomenon is shown in Figure 4.2.

In Figure 4.2 a system with 8 URLs is considered. On each iteration 3 URLs are chosen. The dotted lines indicate the previous area borders and the solid lines

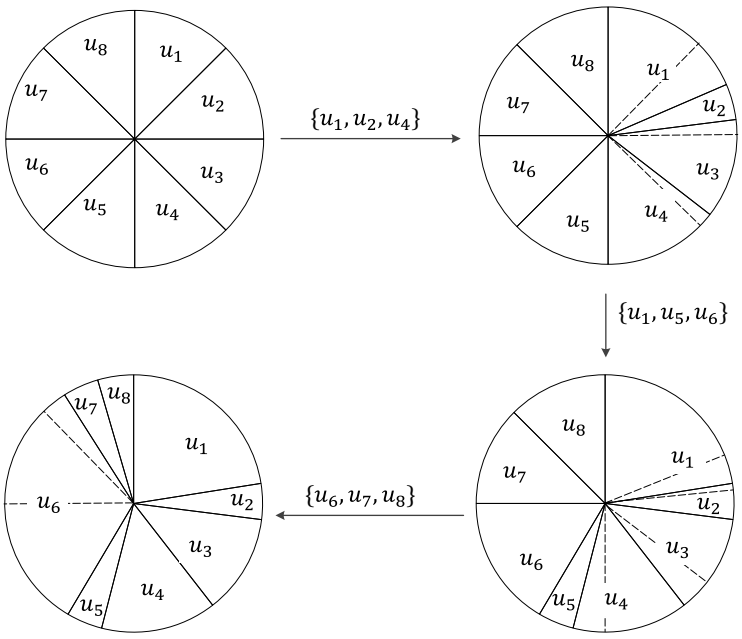

Figure 4.2: Illustration of area readjustment of multiple URLs on each iteration

indicate the current ones. It could be seen that the area is readjusted between the URLs that are chosen in the current iteration and the other remains the same. This keeps the loss-gain scenario relative which is important as we do not have any information regarding the other elements in the current iteration.

The BASIC algorithm can be called iteratively until a desire time limit to steer the application's execution. As the number of iteration will increase, profiles with more execution time will be generated. This means that the execution will tend towards the computationally intensive region which is the prime goal of BASIC. The architecture and the work flow of BASIC is illustrated in Figure 4.3 to further clarify its working procedure.

From Figure 4.3, it is seen that there are two core components of BASIC: the Profiler and the Analyzer. Besides there are repositories to store all the URLs and their probabilities, current set of URLs and the current profiles. Moreover there is also an Initiator that will initiate the whole process by asking the Profiler to create w number of profiles.

The profiler has four components: the Controller, the Selector, the Runner

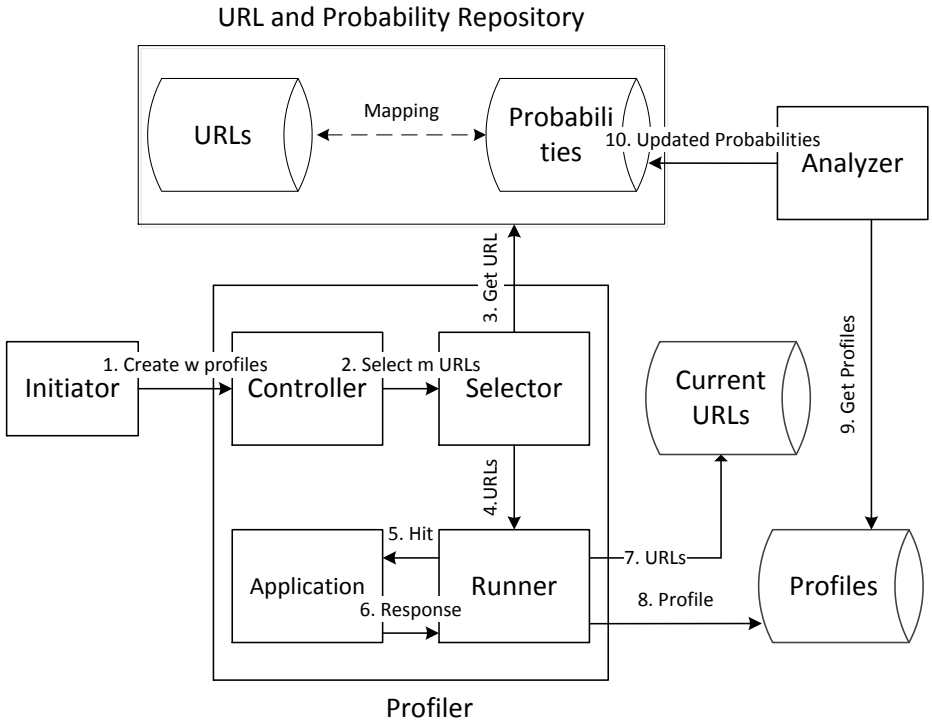

Figure 4.3: The architecture and work flow of BASIC

and the Application that will be tested. The controller will receive the request from the Initiator and ask the Selector to select m URLs from the repository. The Selector should select the URLs using Algorithm 1. The URLs will be then passed to the Runner which will create and run the test scripts. After running the test scripts, it will create a profile using Algorithm 2. The profile should be stored in the profile repository. The controller will repeat these procedures until w profiles are created.

The analyzer will analyze those profiles to determine the label's probability and the prior probabilities. These probabilities will be used to calculate the posterior probability of each URL in the current iteration of being time intensive. The probabilities in the probability repository are then updated using these posterior probabilities.

4.6 Incorporating Odds Ratio

Readjusting the area in the roulette wheel for each URL is an important step of BASIC. However, on readjusting the area only the probability of being good i.e. time intensive is considered. It should be remembered that a URL may also have a probability of being bad i.e. not time intensive. In that case, we may achieve better result if we could incorporate the probability of being bad for each URL. The posterior probability of a URL of being bad can be calculated using Equation 4.5.

$$P(l_j = bad|u_i) = \frac{P(u_i|l_j = bad)P(l_j = bad)}{\sum_{l_j \in \mathcal{L}} P(u_i|l_j)P(l_j)} \tag{4.5}$$

The probability of having profiles labeled as good, bad or unassigned and the prior probabilities can be calculated using Algorithm 3 and 4 respectively. However, the question is how to incorporate this posterior probability of being bad with the posterior probability of being good. The answer to this problem could be using Odds ratio [44].

Odds Ratio (OR) is used to quantify how closely the presence or absence of one property is associated with the presence or absence of another property. In this case, these properties are probability of being good and probability of being bad. Each URL from the URL set will have both of those probabilities and a ratio can be formed which quantitatively describes the association between good and bad. This ratio is the odds ratio which can be computed using Equation 4.6.

$$r_i = \frac{P(u_i|l_j = good)}{P(u_i|l_j = bad)} \tag{4.6}$$

In Algorithm 5 r_i is determined based on the value of $P(l_j = good|u_i)$. To incorporate the OR one should also calculate the value of $P(l_j = bad|u_i)$. The ratio of these two posterior probabilities can be used to assign the value of r_i. This r_i will then work as the basis of readjusting area in the roulette wheel.

4.7 Resolving problems of Rule Based System

State of the art techniques for guiding system's execution are based on rules which introduce issues such as complexity, permanently discarding URLs, statemate situations, etc. Firstly, rules generating is a complicated procedure and may require extra costs in terms of time. Secondly, the current rule based system generates rule which may permanently discard a URL even though the data based on which the decision was made is relative to present iteration. And lastly conflicting situations may arise as rules are rigid in nature. All those issues are addressed by BASIC which is illustrated in the following subsections.

4.7.1 Minimizing Complexity

The complexity of rule generation procedure can vary from $O(n^4)$ to $O(mlog^2m)$ [45, 46], where n is the number of attributes and m is the number of examples. Even if we choose algorithms like [45, 46] the complexity will be $O(mlog^2m)$. However, in our context the complexity will depend on creating the data structure for analyzing and creating the rules. For rule generation, the data structure should have all the attributes, in this case all the URLs, and the number of their occurrence on each example i.e execution profile.

To create the data structure, one has to check all the URLs with the URLs present in the current profiles which will require three loops. Let the number of URLs be n, the number of profiles be w and the number of URLs present in the profiles be km where k is a factor of non-uniqueness. In this case we can at most select n URLs in a test script and thus k could have any value from the range $[\frac{m}{n}, 1]$. So the complexity becomes $O(wnm)$. Now in the worst case the value of w an m can turn out to be n resulting the complexity to $O(n^3)$.

As the context is about large scale system's, the number of URLs (n) will be

much much greater than the number of examples (m). So the bottleneck of the system will be the part involving data structure creation instead of rule generation. So in the worst case, it can be said that the complexity of a rule based system in the context of guiding system's execution is $O(n^3)$.

Seeing the algorithms presented in Section 4.2-4.5, it can be said that the bottleneck of BASIC will be the prior probability calculation scheme. The prior probability calculation will require three for loops similar to rule based system. However, instead of going through all the URLs, BASIC will check the URLs of the current iteration. Let the number of URLs in the current iteration be p. Then the complexity of BASIC turns out to be $O(wpm)$. In the worst case, all these values could be equal to the number of total URLs (n), thus making the worst case complexity $O(n^3)$.

It should be noted that in case of adaptive software performance testing one has to handle a large number of URL set. The URL set is so large that it is not possible to test all of them exhaustively. Thus usually one should proceed by iteratively choosing a smaller subset of the total URL set. Thus if the cardinality of the overall URL set is n and subset on each iteration is m then $m \ll n$. Now in case of rule based system one has to go through all the URLs to create the data structure whereas BASIC only analyzes the URLs used in the current set. Thus even if the worst case complexity for both is $O(n^3)$, in real life BASIC should perform much more faster.

4.7.2 Preventing URls being Discarded

Consider a URL u resides in a bad profile for multiple number of times. Thus a rule is generated: $(u \geq 1) \implies$ class=bad. The meaning of the rule is that if u is selected in a test script then the profile generated by that test script will be labeled as bad. Now let us define an event E where URL u is selected in a test

script. As a rule has been generated, the system will interpret it as following:

$$l_k \in E_k = \begin{cases} good & \text{if event } E \text{ is false} \\ bad & \text{if event } E \text{ is true} \end{cases} \tag{4.7}$$

As the goal of the system is to yield good profiles, the system will discard the URL (u) involved in that event. Eventually this URL will never be selected in a test script which might be a wrong decision. This is because the data based on which this decision was made is relative to the current iteration. There may be a case that URL (u) was selected with the most time consuming URLs of the system. Thus it was labeled as a bad one. However, being selected with other URLs may turn that URL as a good one. But a rule based system will not give that chance and will permanently discard that URL.

Now if the above scenario occurs in case of BASIC, the URL (u) will receive a probability (p) of being time intensive. However, that value will be very much low as most of the time the URL was a part of bad profiles. Now this probability will be interpreted as:

if Event E occurs the profile will be labeled as good with probability p

This probability will be associated with the probability of selection for the URL i.e a proportion of the wheel will be assigned to that URL based on the probability. Thus if the value is low it will have a low probability of being selected without being permanently discarded. This will atleast provide a chance, though very small, to that URL of being selected again and establish itself to be time intensive.

4.7.3 Resolving Conflicts

Let us consider that the set of all URLs is denoted as $\mathcal{U}$. Now let us consider two sets of URLs $U_1, U_2 \subset \mathcal{U}$ are chosen to create test scripts. The function ψ defines a rule for a URL with respect to its URL set. If the URL u is present in both the subsets (U_1, U_2) i.e. $u \in (U_1 \cup U_2)$ then the following conflicting situation may arise:

$$\psi(U_1, u) \rightarrow u \text{ is Good} \tag{4.8}$$

$$\psi(U_2, u) \rightarrow u \text{ is Bad} \tag{4.9}$$

This means that if URL u is a part of U_1 then a rule will be generated saying that- if u is kept in a test script then the profile generated by the test script will be labeled as good. However, the opposite rule is generated when u will reside within U_2. This will create a conflicting situation as the system can not decide whether to keep that URL or to discard it.

If the above scenario occurs in case of BASIC that URL will receive a probability. If the function χ assigns a probability to the URL (u), then the following will occur:

$$p_1 = \chi(U_1, u) \rightarrow [0, 1] \tag{4.10}$$

$$p_2 = \chi(U_2, u) \rightarrow [0, 1] \tag{4.11}$$

That is in case of BASIC, for the first scenario URL u will receive a high probability of being time intensive. For the second scenario, the URL will receive a low probability of being time intensive. The area of that URL in the roulette wheel will be readjusted based on these probabilities. This readjustment will also be relative with respect to the URLs used in the current iteration. Thus no conflicting situation will arise and only the possibility of each URL of being selected will only be readjusted.

4.8 Summary

In this section, a novel scheme for guiding system's execution towards its computationally intensive region named as BASIC is illustrated. Multiple aspects of the BASIC scheme that includes the URL selection strategy, clustering execution profiles, calculating prior probabilities and calculating the posterior probability is also described. BASIC has also addressed some issues of rule based system like complexity, permanent URL discarding and creating stalemate situations. It has also been discussed that how BASIC resolves all those issues.

Chapter 5

Experimental Setup and Results

This chapter presents the implementation scheme of BASIC along with its evaluation on different criteria. BASIC was tested with two popular open source E-commerce applications - JPetStore and MVC Music Store. Initially a brief description of these two applications are presented. The implementation scheme of BASIC is provided having an insight of the used technologies. The performance of BASIC is compared to two different approaches- FOREPOST and Random Testing. The details regarding the experimental setup of those two schemes are also provided. Finally, the performance of BASIC is evaluated on the basis of following- complexity, capability of preventing URLs being discarded, conflict resolution and capability to worsen system's response time.

5.1 Experimental Setup

Two common open source E-commerce applications - JPetStore and MVC Music Store have been used to evaluate the performance of BASIC. Initially a description of these two applications is provided with configuration details (web server and database configurations). Along with BASIC, two other schemes- FOREPOST and Random Testing have been implemented for a comparative analysis. This section highlighted all these details.

5.1.1 Test Applications

The applications chosen for conducting experiment- JPetStore and MVC Music Store both possess regions that are performance critical. One of those regions is the catalog browsing option where users can browse for their desired items and a

large number of users can browse in parallel. These parts also requires database communications making it those ideal candidates for performance bottlenecks. A description of these applications and their deployment configuration are provided below:

JPetStore

JPetStore[1] is an online application that provides a virtual pet shop from where users can buy pets. Users can choose animals of their own choice and add those to the virtual shopping cart. Users have to enter their billing details including shipping address and card information. JPetStore 6 is developed based on MyBatis, Spring Framework and Stripes. The URLs associated to the browsing module (users can browse through different categories of pets) are based on HTTP Get requests. These URLs are considered during the experiment to find out which one of them plays a vital role in worsening response time.

JPetStore was deployed in a desktop machine using the Apache Tomcat Web Server version 7. The server was specifically allocated one GB of memory for allowing it to smoothly handle the user requests. An XML based database is attached to the application filled up with mock data. The MyBatis framework is used to get the desired data from the database during the test case execution.

MVC Music Store

MVC Music Store[2] is another online music store application which allows users to buy music albums of several genres. Each user gets a virtual cart for adding multiple items and the user can checkout the cart at the time of his/her departure. The MVC Music Store is based on a Model-View-Controller framework and there

[1] `github.com/mybatis/jpetstore-6`
[2] `mvcmusicstore.codeplex.com`

are mainly two types of GET requests. One is for getting into various genres of music and another is for getting into music albums of a specific genre. In this experiment, the URLs associated to those GET requests were considered for the inclusion in test cases.

The application was hosted in Internet Information Service (IIS) version 7. A separate application pool is created in the IIS and the application was deployed under it. One GB of memory was allocated to this pool so that it can serve a large number of requests simultaneously. Microsoft SQL Server 2008 was used as the database for this application. The database was made filled with mock data provided with the application before executing the test cases.

5.1.2 Random Testing

The Random Testing Technique chooses URLs for adding in the test script using a random approach. A tool was developed using Java Programming language that randomly chooses URLs from a list. URLs are added in the test scripts and executed using Apache Jmeter API [47]. Apache Jmeter is a tool to load test functional behavior and measure performance. It is based on a fully multi-threaded framework that allows to specify the number of concurrent users needed to simulate. This is achieved by mentioning the number of threads in the Thread Group option of Apache Jmeter.

The tasks that need to be assigned to each of the virtual users should be specified after deciding the total number of users. That is, the number of HTTP requests sent by each of the user have to be mentioned. It is also required to mention the number of times the test case will execute. For that, the Loop Controller option provided by the Jmeter was used. The URLs were added in a sampler and this along with the loop controller are added in the Hash tree option. This hash tree was passed to the Jmeter engine to execute the tests.

5.1.3 FOREPOST

In case of Random testing, URLs were chosen for test cases using a random approach. In the next iteration of the adaptive process, the URLs will again be chosen randomly. Thus no extra information is required to be stored for further analysis. However, FOREPOST requires certain information to generate rules and thus two extra modules will be required- the profiler and the analyzer. The profiler stores the information gathered upon executing test cases. This information is termed as the profile. Each profile consists of its execution time and the URLs used in the test case corresponding to this profile. A Java class named as Profile was created to represent the execution profiles. The Profile class will contain a list of URL and a variable to store the execution time. At the end of each iteration a list of Profile was provided to the analyzer.

The analyzer classified each profiles as either good, bad or unassigned. For that the execution time of the profiles are compared. After classifying the profiles, rules were generated using the JRip [48] classifier provided by Weka. Weka requires an ARFF file to generate rules which was created programmatically. An ARFF file has two parts - attributes and instances. In this implementation the URLs were the attributes and profiles were the instances. Each instance represents a profile having the number of times a URL was present in the profile. The label of the profile was used as the class variable of the ARFF file. The rules which were generated at the previous iteration was used to include URLs to a test script in the current iteration.

5.1.4 BASIC

BASIC used the same profiler like FOREPOST. However, it had an analyzer of its own that implemented the Bayes theorem to characterize URLs. The analyzer was

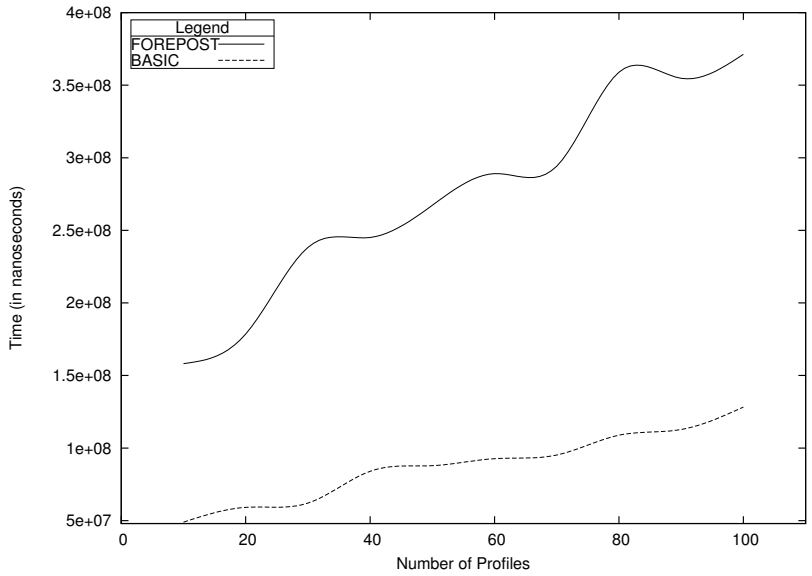

Figure 5.1: Time taken by FOREPOST to generate rules and BASIC to update probability with the increase of profile numbers

developed using Java programming language and the list of profiles was sent to it as an input. A hashed set from the Java Collections framework was used to store the URLs used in the current iteration. The occurrences of each of those URLs were counted. The probabilities were then calculated and used by the roulette wheel selection algorithm.

5.2 Complexity Comparison between FOREPOST and BASIC

As discussed in Chapter 4, the runtime of BASIC should be lower than FOREPOST considering the case of steering application's execution towards time intensiveness. This is because FOREPOST considers all the URLs in the pre-processing stage before rule generation whereas BASIC only considers the URLs of the current iteration. This is shown in Figure 5.1 where the runtime is plotted with the increase of number of profiles.

It can be seen that the runtime of both the approaches increases with the num-

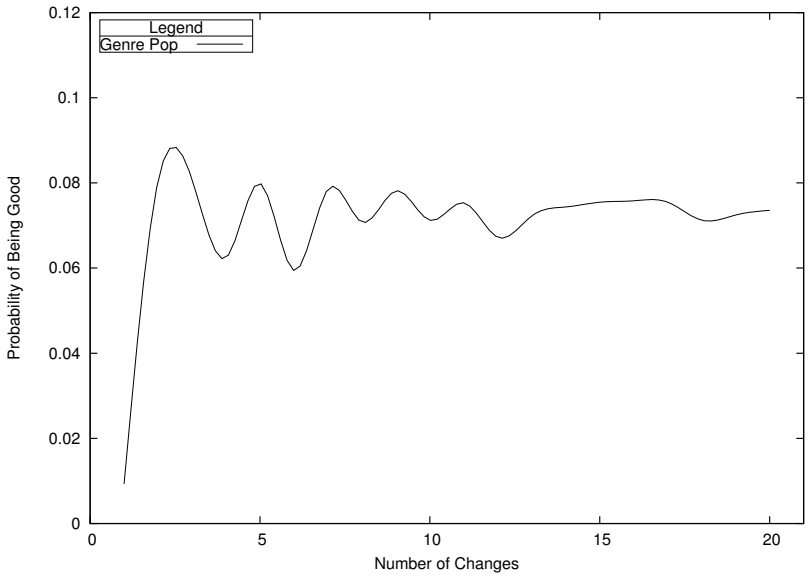

Figure 5.2: Illustration of how a low probability can change in the BASIC approach

ber of profiles. However, the runtime of BASIC is much lower than FOREPOST. The slope of the curve in case of BASIC is less steeper than of FOREPOST indicating that the runtime of FOREPOST will always be high. From Figure 5.1 it is seen that for 100 profiles the time taken by BASIC to complete the calculation is around 130 millisecond whereas in case of BASIC it is around 350 millisecond.

5.3 Preventing URLs being Discarded in BASIC

Permanently discarding a URL from the consideration for being included in a test script is not right. Because this decision is made on relative data and this URL may turn out to be a good one if was selected with another set of URLs. In case of BASIC, the URL will receive a low probability but will still have a possibility of getting selected from the roulette wheel selection algorithm. After getting selected with another set of URLs, it may receive a high probability of being time intensive. One such scenario is illustrated for the URL "/Store/Browse?Genre=Pop" in Figure 5.2.

The URL initially received a low probability indicating that it will not be time

Table 5.1: First two set of URLs used to generate conflicting rules using the FOREPOST approach

Number	Set-I	Set-II
1	/Store/Browse?Genre=Rock	/Store/Details?id=10
2	/Store/Details?id=1	/Store/Details?id=11
3	/Store/Details?id=2	/Store/Details?id=12
4	/Store/Details?id=3	/Store/Details?id=13
5	/Store/Details?id=4	/Store/Details?id=14
6	/Store/Details?id=5	/Store/Details?id=15
7	/Store/Details?id=6	/Store/Details?id=16
8	/Store/Details?id=7	/Store/Details?id=17
9	/Store/Details?id=8	/Store/Details?id=18
10	/Store/Details?id=9	/Store/Details?id=19

intensive. However, being picked with other URLs it received a higher value and gradually its probability got stabilized. In Figure 5.2, the probability of the URL is plotted whenever a change occurred in its value. It can be seen that after one stage the value does not tend to change much and becomes stable. The probabilities after conducting the experiment for all the URLs of JPetStore and MVC Music Store is presented in Table B.1 and B.2 in the Appendix.

5.4 Avoiding Conflict in BASIC

A rule based system can fall into a stalemate situation due to having conflicting rules. Let us consider the two set of URLs of the application music store presented in Table 5.1. It has been seen experimentally that the URL number 1 is more time intensive than the other URLs in Table 5.1. Thus the rule **R1** was generated after generating execution profiles from those URLs. The meaning of 25 in the rule is the URL was considered once in the test case. However, the value is 25 as that URL was hit by 5 concurrent users, each of them for 5 times. Thus the value is shown as 25.

$$\textbf{R1}\ \ (\text{Genre\_Rock} \leq 0) \implies \text{class=bad}$$

Table 5.2: Second two set of URLs used to generate conflicting rules using the FOREPOST approach

Number	Set-I	Set-II
1	/Store/Browse?Genre=Rock	/Store/Details?id=1
2	/Store/Browse?Genre=Classical	/Store/Details?id=2
3	/Store/Browse?Genre=Jazz	/Store/Details?id=3
4	/Store/Browse?Genre=Pop	/Store/Details?id=4
5	/Store/Browse?Genre=Disco	/Store/Details?id=5
6	/Store/Browse?Genre=Latin	/Store/Details?id=6
7	/Store/Browse?Genre=Metal	/Store/Details?id=7
8	/Store/Browse?Genre=Alternative	/Store/Details?id=8
9	/Store/Browse?Genre=Reggae	/Store/Details?id=9
10	/Store/Browse?Genre=Blue	/Store/Details?id=10

R1 means that the URL "/Store/Browse?Genre=Rock" should be included in the test case for getting slower response time. However, an opposite rule (**R2**) is derived upon considering the URLs in Table 5.2 is considered. This is because the URLs in Set-I of Table 5.2 are all good in worsening the system response time. Thus the URL "/Store/Browse?Genre=Rock" may have not performed well with respect to the other URLs and thus **R2** was generated. This means that the URL should not be considered in the next iteration as it is not time intensive.

$$\textbf{R2}\ (\text{Genre\_Rock} \geq 25) \implies \text{class=bad}$$

All the rules that have been generated for both the applications along with their interpretations are provided in Table A.1 and A.2. These rules will give an insight about the role of certain URLs towards time intensiveness in terms of FOREPOST.

5.5 Performance Evaluation in terms of Execution Time

The main goal of BASIC is to guide the execution towards time intensive region that is worsening the system response time. Based on this goal, this section

Table 5.3: Data of Total execution time with varying number of transactions for the application JPetStore

Transactions	Random	FOREPOST	BASIC	BASIC-OR
250	1430723.237	1443085.689	1439661.979	1437007.138
500	2847698.03	2859056.674	2922622.072	2861229.25
750	4273913.192	4330576.845	4426519.443	4278412.608
1000	5730539.775	5974350.569	5928301.04	5694613.037
1250	7174701.929	7500326.852	7434614.218	7119003.22

presents a comparative analysis of BASIC with Random Testing and FOREPOST. The performance of BASIC after incorporating odds ratio (BASIC-OR) is also presented. Initially the total execution time for each of those approaches are presented. Later the variability of the total execution time and the increase in execution time at each iteration is also presented.

5.5.1 Comparing Total Execution Time

From Figure 5.3 and 5.4 it is seen that with the increase of transactions, better results can be obtained using BASIC and FOREPOST scheme. Moreover, BASIC performs similar to FOREPOST making it an alternative for guiding application's execution towards time intensiveness. The data corresponding to these figures are presented in Table 5.3 and 5.4.

In Figure 5.3, it can be seen that BASIC-OR did not performed well even with respect to Random testing. However, in case of the music store application the performance of BASIC-OR increases with increase in number of transactions. This is presented in Figure 5.4 that illustrates that BASIC-OR performs better than both BASIC and FOREPOST.

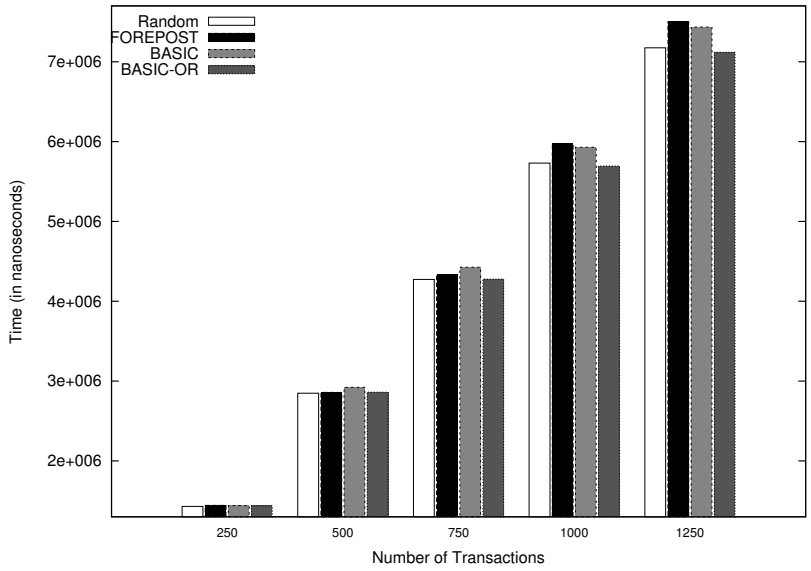

Figure 5.3: Comparison of total execution time with increasing number of transactions for the application JPetStore

Table 5.4: Data of Total execution time with varying number of transactions for the application MVC Music Store

Transactions	Random	FOREPOST	BASIC	BASIC-OR
1250	2.19377E+12	2.51611E+12	2.62552E+12	2.74623E+12
2500	4.409E+12	5.22405E+12	5.29189E+12	5.51264E+12
3750	6.64888E+12	8.07611E+12	7.92406E+12	8.31928E+12
5000	8.8902E+12	1.0733E+13	1.05798E+13	1.10879E+13
6250	1.1142E+13	1.33448E+13	1.32732E+13	1.38382E+13

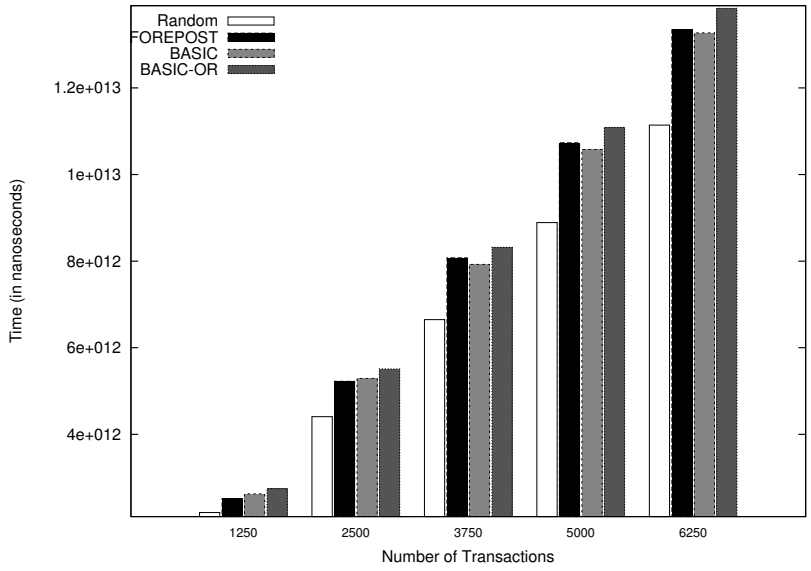

Figure 5.4: Comparison of total execution time with increasing number of transactions for the application MVC Music Store

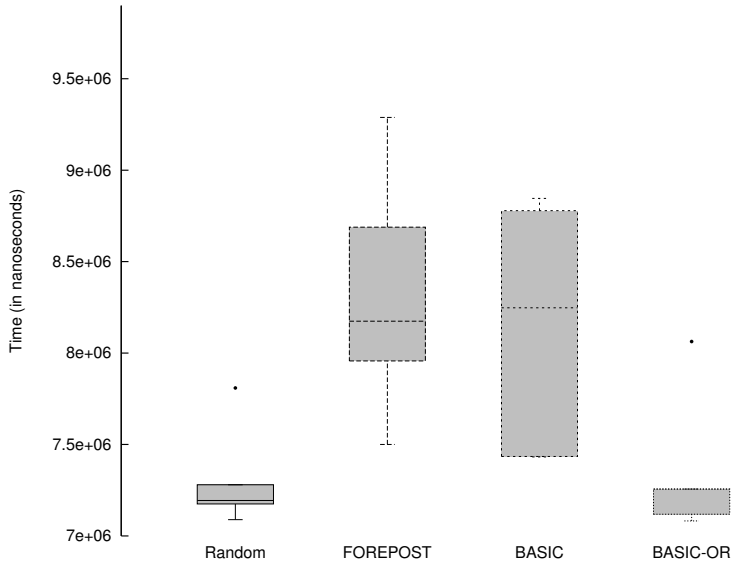

Figure 5.5: Box whisker plot showing the variability in execution profile's time for the application JPetStore

5.5.2 Variability in Execution Time

Figure 5.5 and 5.6 shows the variability in the execution time in terms of box whisker plot. For the pet store application, it can be seen from Figure 5.5 that though Random testing and BASIC-OR have low variability (indicated by the squeezed boxes), their execution time is much less than BASIC and FOREPOST. Although the variability of BASIC and FOREPOST is much higher which indicating that sometimes they could also have lower execution time.

From Figure 5.6 it can be seen that the variability of all the four approaches are comparatively less in case of music store application. It is clearly visible that the performance of FOREPOST, BASIC and BASIC-OR is much higher than the performance of Random testing. The variability of the BASIC approach is the minimum however, BASIC-OR has the minimum lower value as well as the highest value. Although the FOREPOST approach has a higher value similar to BASIC and BASIC-OR, the median of its total execution time is much lower.

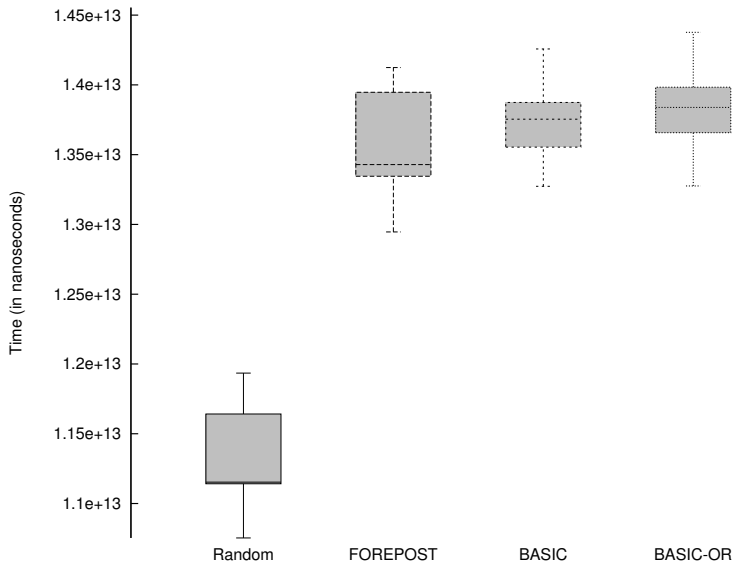

Figure 5.6: Box whisker plot showing the variability in execution profile's time for the application MVC Music Store

5.5.3 Time Growth in Each Iteration

It is essential to see that how the execution time is increasing on each iteration for each of the four approaches. This will provide an insight of which approach is moving towards the highest point quickly. Figure 5.7 shows the time growth on each iteration for the pet store application. It is observed that the growth of FOREPOST is much better than the other three. BASIC outperforms the other two however the performance of BASIC-OR is the worst in this case.

In case of the music store application, as shown in Figure 5.8, it is clearly visible that the Random testing is outperformed by the other three approaches. The performance of the BASIC-OR approach is better compared to the other two approaches. That is the time growth in case of BASIC-OR is much faster and quickly reaches the pick. However after sometime, the time growth for all the approaches get reduced and tends to stabilize.

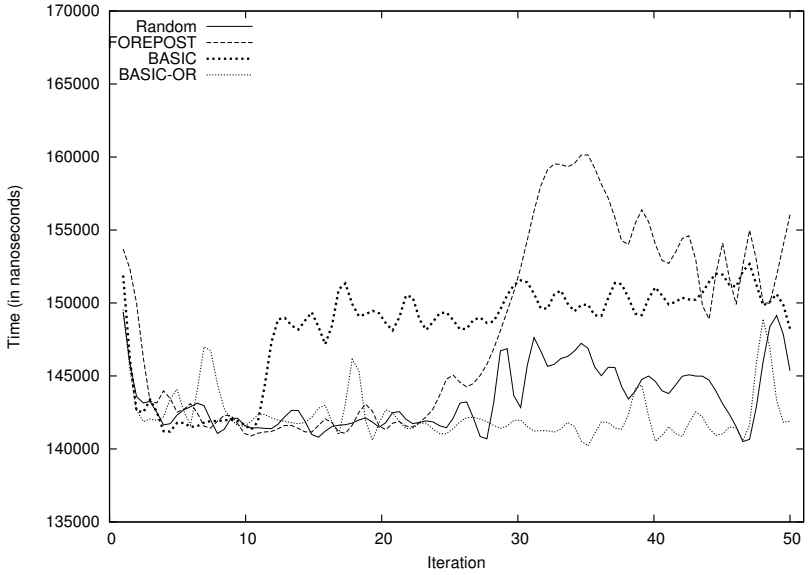

Figure 5.7: The growth of time with the increase of iteration number in four different approaches for the application JPetStore

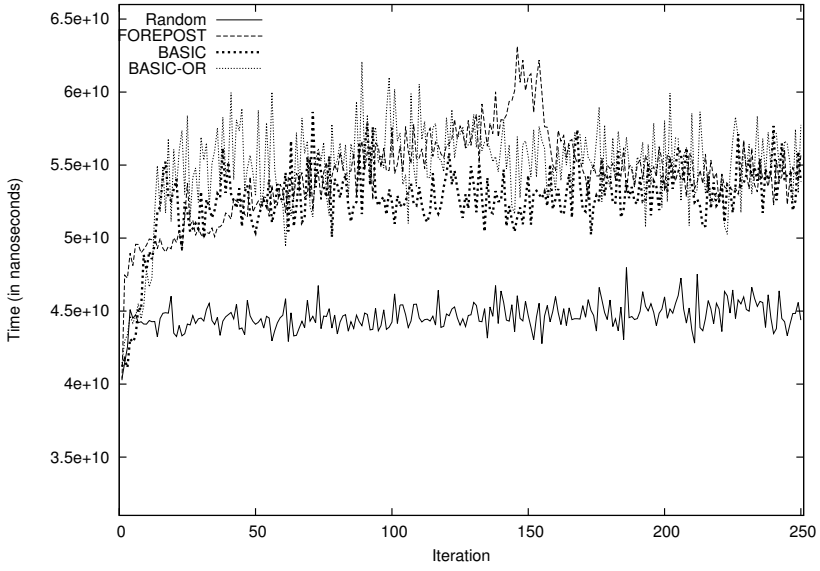

Figure 5.8: The growth of time with the increase of iteration number in four different approaches for the application MVC Music Store

5.6 Summary

This chapter presents the performance analysis of BASIC along with its implementation scheme. The configuration details of the test application is also provided so that others can simulate similar test environments. The results have been compared to Random testing and FOREPOST approaches in terms of complexity, conflict resolution, URL discarding and response time. It has been shown that BASIC performs similar in terms of worsening response time. Moreover, it has a lower runtime and does not create a conflicting situation. The problem of permanently discarding URLs, that persist in FOREPOST, is also removed in BASIC.

Chapter 6

Conclusion

This chapter initially presents a discussion about the key benefits of BASIC and certain aspects of the experimental results. The concluding remarks are then provided to highlight the overall achievements of this work. Finally, the chapter is concluded by giving an insight of future research directions regarding adaptive software performance testing and steering system execution.

6.1 Discussion

BASIC can assist any adaptive performance testing scheme to identify performance bottlenecks, with more precision, by offering some benefits of its own. One of the key benefits of BASIC is that it will not discard any URL by labeling it as bad like FOREPOST. Instead, the proposed approach assigns a very small probability to that URL giving it a minor chance to get selected from the roulette wheel selection algorithm. This is important as this URL may turn out to be a good one in any other combination of URL selection.

It has been shown both theoretically and experimentally that BASIC has a lower complexity than FOREPOST. From the experiment it is seen that for 100 profiles, the runtime (algorithm execution time without the response time of the URLs) of FOREPOST is around 350 milliseconds and BASIC is around 130 milliseconds. That is BASIC has almost 65% lower value in terms of runtime. This is the value of one iteration and if there are 250 iterations then the values for FOREPOST and BASIC could be around 90 and 27 seconds. Although the value of BASIC is much more lower, considering the overall runtime (including response time) this lower value may not be much significant (usually the response time for 250 iterations is within the range of 10800 to 14400 seconds).

The growth in execution time for each iteration is shown in Chapter 5. In case of the JPetStore application the performance of BASIC-OR approach was even worse than Random Testing. One of the reason behind that is a small portion of the application was considered having only 49 URLs. Thus a smaller number of iteration (50) was allowed in the adaptive process. As BASIC-OR incorporates the probability of being "bad" with the probability of being "good", it takes time for the URLs to receive a score of higher value regarding its chance of being time intensive. As the number of iteration was small it did not get that chance. However, in case of MVC Music Store BASIC-OR performed better than BASIC and FOREPOST as in this setup 250 iterations were considered.

In case of the probabilistic system the time growth is supposed to be slow in the initial phases. This is because there will be always a possibility of URLs having lower probability getting selected. In rule based system, rules are generated saying which URLs should be considered and which should not. Thus bad URLs are usually discarded however, as said earlier, marking a URL as bad may not always be the right decision. The probabilistic approach may not be fast in gaining slower response time but gradually it starts to get higher values as the probabilities become stabilized.

6.2 Concluding Remarks

This thesis proposes BASIC - a Bayesian technique for steering large-scale web based systems towards its time intensive region. At the beginning, an insight of software testing was provided by an taxonomy. The interrelationships of performance and exploratory testing was also discussed. It has been shown that how a combination of those two testing schemes can be achieved through adaptive performance testing. Study of the related work showed that, adaptive software performance testing is one of the most recent research trends. The existing scheme

is based on the assumption that steering system execution towards time intensiveness will expose performance bugs. However, the scheme guides the execution by utilizing rules which have issues like rigidity and creating conflicts. To resolve such issues a new steering scheme named as BASIC has been proposed.

The scheme is based on Bayesian learning that selects URLs for test cases based on their capability of worsening system response time. The technique was implemented and tested against two E-commerce applications - JPetStore and MVC Music Store. The performance of BASIC was compared to FOREPOST - a rule based system for guiding system's execution. Experimental results show that BASIC helps in achieving similar slow response time like FOREPOST with the added benefit of being less complex, no conflict situations and no URLs being permanently discarded.

6.3 Future Work

Adaptive software performance testing have many sub-fields such as steering scheme, log analysis, etc. where researchers should focus. This research has focused on one such issue that involves steering the system execution to expose performance bugs. However, there are plenty more research issues regarding execution steering such as test coverage, parameters estimation, etc. that need to be addressed by the research community. These issues are related to both adaptive performance test and the proposed steering scheme BASIC.

One of the major issue in adaptive performance testing is to decide the stopping criteria of the adaptive process. Currently the process terminates after completing a specific number of iterations for a certain number of times. However, determining an optimal point for which no further execution paths are required, can be investigated in the context of current application to expose performance bugs. The adaptive scheme will then terminate after finding such optimal point.

It has been shown that BASIC does not permanently discard a URL. This provides a chance to re-test that URL again for its capability regarding time intensiveness. Thus BASIC improves the test coverage of the adaptive software testing process by providing the chance to test a URL with other set of URLs. However, in this research less emphasis is provided on test coverage. One future work could be providing a mathematical model of the test coverage scheme of BASIC. Techniques should also be provided so that one can calculate the increase in test coverage with respect to the application under test by using BASIC.

BASIC requires to parameters to be estimated - the number of URLs to be added in a test script and the number of profiles that should be created before starting the analysis. These parameters could be chosen on trial and error basis. However, a better way could be if one could establish a relation to these parameters with the size of the application that will be tested. During the testing phase, one could easily estimate those parameters based on its relationship with the application volume.

Bibliography

[1] E. J. Weyuker and F. I. Vokolos, "Experience with performance testing of software systems: issues, an approach, and case study," *IEEE transactions on Software Engineering*, vol. 26, no. 12, pp. 1147–1156, 2000.

[2] D. Menascé, "Load testing of web sites," *IEEE Internet Computing*, vol. 6, no. 4, pp. 70–74, 2002.

[3] J. Bach, "Exploratory testing explained," *Online: http://www. satisfice. com/articles/et-article. pdf*, 2003.

[4] A. Aleti and I. Meedeniya, "Component deployment optimisation with bayesian learning," in *Proceedings of International Symposium on Component Based Software Engineering*, pp. 11–20, ACM, 2011.

[5] M. Grechanik, C. Fu, and Q. Xie, "Automatically Finding Performance Problems with Feedback-Directed Learning Software Testing," in *Proceedings of International Conference on Software Engineering*, pp. 156–166, IEEE, 2012.

[6] G. J. Myers, C. Sandler, and T. Badgett, *The art of software testing*. John Wiley & Sons, 2011.

[7] E. Kit and S. Finzi, *Software testing in the real world: improving the process*. ACM Press/Addison-Wesley Publishing Co., 1995.

[8] B. Beizer, *Black-box testing: techniques for functional testing of software and systems*. John Wiley & Sons, Inc., 1995.

[9] S. H. Edwards, "A framework for practical, automated black-box testing of component-based software," *Software Testing, Verification and Reliability*, vol. 11, no. 2, pp. 97–111, 2001.

[10] P. Tonella and F. Ricca, "A 2-layer model for the white-box testing of web applications," in *Proceedings of Annual International Telecommunications Energy Conference*, pp. 11–19, IEEE, 2004.

[11] J. Zhang, C. Xu, and S. Cheung, "Automatic generation of database instances for white-box testing," in *Proceedings of Annual International Computer Software and Applications Conference*, pp. 161–165, IEEE, 2001.

[12] W. Linzhang, Y. Jiesong, Y. Xiaofeng, H. Jun, L. Xuandong, and Z. Guoliang, "Generating test cases from uml activity diagram based on gray-box method," in *Proceedings of Asia-Pacific Software Engineering Conference*, pp. 284–291, IEEE, 2004.

[13] N. Kicillof, W. Grieskamp, N. Tillmann, and V. Braberman, "Achieving both model and code coverage with automated gray-box testing," in *Proceedings of International Workshop on Advances in Model-based Testing*, pp. 1–11, ACM, 2007.

[14] G. Denaro, A. Polini, and W. Emmerich, "Early performance testing of distributed software applications," in *ACM SIGSOFT Software Engineering Notes*, vol. 29, pp. 94–103, ACM, 2004.

[15] T. Y. Chen, R. G. Merkel, G. Eddy, and P. Wong, "Adaptive random testing through dynamic partitioning," in *Proceedings of International Conference on Quality Software*, pp. 79–86, 2004.

[16] R. Huang, X. Xie, T. Y. Chen, and Y. Lu, "Adaptive random test case generation for combinatorial testing," in *Proceedings of Annual Computer Software and Applications Conference*, pp. 52–61, IEEE, 2012.

[17] K. P. Chan, T. Y. Chen, and D. Towey, "Restricted random testing," in *Software Quality–ECSQ 2002*, pp. 321–330, Springer, 2002.

[18] Z. Q. Zhou, A. Sinaga, and W. Susilo, "On the fault-detection capabilities of adaptive random test case prioritization: Case studies with large test suites," in *Proceedings of Hawaii International Conference on System Science*, pp. 5584–5593, IEEE, 2012.

[19] R. Toldo and A. Fusiello, "Robust multiple structures estimation with j-linkage," in *Computer Vision–ECCV 2008*, pp. 537–547, Springer, 2008.

[20] Z. Q. Zhou, "Using coverage information to guide test case selection in adaptive random testing," in *Proceedings of Annual Computer Software and Applications Conference Workshops*, pp. 208–213, IEEE, 2010.

[21] T. Chen, F. Kuo, H. Liu, and E. Wong, "Code coverage of adaptive random testing," *IEEE Transactions on Reliability*, vol. 62, no. 1, pp. 226–237, 2013.

[22] A. U. Gias, R. Rahman, A. Imran, and K. Sakib, "TFPaaS : Test-first Performance as a Service to Cloud for Software Testing Environment," *International Journal of Web Applications*, vol. 5, no. 4, pp. 153–167, 2013.

[23] W. Dickinson, D. Leon, and A. Podgurski, "Finding failures by cluster analysis of execution profiles," in *Proceedings of International Conference on Software Engineering*, pp. 339–348, IEEE Computer Society, 2001.

[24] D. Leon, A. Podgurski, and L. J. White, "Multivariate visualization in observation-based testing," in *Proceedings of International Conference on Software Engineering*, pp. 116–125, ACM, 2000.

[25] A. Podgurski, W. Masri, Y. McCleese, F. G. Wolff, and C. Yang, "Estimation of software reliability by stratified sampling," *ACM Transactions on Software Engineering and Methodology*, vol. 8, no. 3, pp. 263–283, 1999.

[26] W. H. Day and H. Edelsbrunner, "Efficient algorithms for agglomerative hierarchical clustering methods," *Journal of classification*, vol. 1, no. 1, pp. 7–24, 1984.

[27] A. Wasylkowski, A. Zeller, and C. Lindig, "Detecting object usage anomalies," in *Proceedings of European Software Engineering Conference and Symposium on Foundations of Software Engineering*, pp. 35–44, ACM, 2007.

[28] G. Sevitsky, W. De Pauw, and R. Konuru, "An Information Exploration Tool for Performance Analysis of Java Programs," in *Proceedings of International Conference on Technology of Object-Oriented Languages and Systems*, pp. 85–101, IEEE, 2001.

[29] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen, "Performance Debugging for Distributed Systems of Black Boxes," in *ACM SIGOPS Operating Systems Review*, vol. 37, pp. 74–89, ACM, 2003.

[30] K. C. Foo, Z. M. Jiang, B. Adams, A. E. Hassan, Y. Zou, and P. Flora, "Mining Performance Regression Testing Repositories for Automated Performance Analysis," in *Proceedings of International Conference on Quality Software*, pp. 32–41, IEEE, 2010.

[31] L. Bulej, T. Kalibera, and P. Tuma, "Regression benchmarking with simple middleware benchmarks," in *Proceedings of International Conference on Performance, Computing, and Communications*, pp. 771–776, IEEE, 2004.

[32] I. Cohen, J. S. Chase, M. Goldszmidt, T. Kelly, and J. Symons, "Correlating Instrumentation Data to System States: A Building Block for Automated Diagnosis and Control," in *Proceedings of Symposium on Operating Systems Design and Implementation*, pp. 231–244, 2004.

[33] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora, "Automatic Identification of Load Testing Problems," in *Proceedings of International Conference on Software Maintenance*, pp. 307–316, IEEE, 2008.

[34] D. Cotroneo, R. Pietrantuono, L. Mariani, and F. Pastore, "Investigation of failure causes in workload-driven reliability testing," in *Proceedings of International Workshop on Software Quality Assurance*, pp. 78–85, ACM, 2007.

[35] S. Hangal and M. S. Lam, "Tracking down software bugs using automatic anomaly detection," in *Proceedings of International Conference on Software Engineering*, pp. 291–301, ACM, 2002.

[36] C. Liu, X. Yan, and J. Han, "Mining control flow abnormality for logic error isolation.," in *Proceedings of SIAM International Conference on Data Mining*, SIAM, 2006.

[37] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora, "An automated approach for abstracting execution logs to execution events," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 20, no. 4, pp. 249–267, 2008.

[38] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox, "Capturing, Indexing, Clustering, and Retrieving System History," in *ACM SIGOPS Operating Systems Review*, vol. 39, pp. 105–118, ACM, 2005.

[39] G. Ammons, J.-D. Choi, M. Gupta, and N. Swamy, "Finding and Removing Performance Bottlenecks in Large Systems," in *Proceedings of European Conference on Object-Oriented Programming*, pp. 172–196, Springer, 2004.

[40] R. Bakalova, A. Chow, C. Fricano, P. Jain, N. Kodali, D. Poirier, S. Sankaran, and D. Shupp, "Websphere dynamic cache: improving j2ee application performance," *IBM Systems Journal*, vol. 43, no. 2, pp. 351–370, 2004.

[41] M. Woodside, T. Zheng, and M. Litoiu, "The Use of Optimal Filters to Track Parameters of Performance Models," in *Proceedings of International Conference on the Quantitative Evaluation of Systems*, pp. 74–83, IEEE, 2005.

[42] G. Welch and G. Bishop, "An introduction to the kalman filter," 1995.

[43] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora, "Automated Performance Analysis of Load Tests," in *Proceedings of International Conference on Software Maintenance*, pp. 125–134, IEEE, 2009.

[44] F. Mosteller, "Association and estimation in contingency tables," *Journal of the American Statistical Association*, pp. 1–28, 1968.

[45] J. Furnkranz and G. Widmer, "Incremental reduced error pruning," in *Proceedings of International Conference on Machine Learning*, pp. 70–77, 1994.

[46] W. W. Cohen, "Fast effective rule induction," in *Proceedings of International Conference on Machine Learning*, (Lake Tahoe, California), 1995.

[47] E. H. Halili, *Apache JMeter: A practical beginner's guide to automated testing and performance measurement for your websites*. Packt Publishing Ltd, 2008.

[48] A. Rajput, R. P. Aharwal, M. Dubey, S. Saxena, and M. Raghuvanshi, "J48 and jrip rules for e-governance data," *International Journal of Computer Science and Security*, vol. 5, no. 2, p. 201, 2011.

Appendix A

Rules Generated by FOREPOST

Table A.1: Rules Generated by FOREPOST for the application JPetStore

Rule	**Interpretation**
(itemId_EST-20 $\geq$ 25) $\implies$ class=good	This URL must be kept in the test case
(itemId_EST-3 $\geq$ 25) $\implies$ class=bad	This URL must not be kept in the test case
(categoryId_REPTILES $\geq$ 25) $\implies$ class=bad	This URL must not be kept in the test case
(categoryId_BIRDS $\geq$ 25) $\implies$ class=bad	This URL must not be kept in the test case
(productId_AV-CB-01 $\leq$ 25) $\implies$ class=bad	This URL must not kept in the test case
(itemId_EST-28 $\geq$ 25) $\implies$ class=bad	This URL must not be kept in the test case
(productId_FI-SW-02 $\leq$ 25) $\implies$ class=good	This URL must not be kept in the test case

Table A.2: Rules Generated by FOREPOST for the application MVC Music Store

Rule	**Interpretation**
(id_179 $\geq$ 25) $\implies$ class=good	This URL must be kept in the test case
(Genre_Rock $\leq$ 25) $\implies$ class=good	This URL must not be kept in the test case
(id_63 $\geq$ 25) $\implies$ class=good	This URL must be kept in the test case
(id_101 $\leq$ 25) $\implies$ class=bad	This URL must be kept in the test case
(id_128 $\geq$ 25) $\implies$ class=bad	This URL must not be kept in the test case
(id_96 $\geq$ 25) $\implies$ class=good	This URL must be kept in the test case
(Genre_Metal $\geq$ 25) $\implies$ class=good	This URL must be kept in the test case
(id_86 $\leq$ 25) $\implies$ class=bad	This URL must be kept in the test case
(id_224 $\leq$ 25) $\implies$ class=bad	This URL must be kept in the test case
(id_162 $\geq$ 25) $\implies$ class=good	This URL must be kept in the test case
(id_33 $\geq$ 25) $\implies$ class=bad	This URL must not be kept in the test case
(id_31 $\geq$ 25) $\implies$ class=good	This URL must be kept in the test case
(id_105 $\geq$ 25) $\implies$ class=good	This URL must be kept in the test case
(id_181 $\leq$ 25) $\implies$ class=bad	This URL must be kept in the test case
(id_9 $\leq$ 25) $\implies$ class=good	This URL must not be kept in the test case
(id_51 $\leq$ 25) $\implies$ class=good	This URL must not be kept in the test case
(id_191 $\geq$ 25) $\implies$ class=good	This URL must be kept in the test case
(id_175 $\leq$ 25) $\implies$ class=good	This URL must not be kept in the test case
(Genre_Disco $\geq$ 25) $\implies$ class=good	This URL must be kept in the test case
(id_95 $\geq$ 25) $\implies$ class=bad	This URL must not be kept in the test case
(id_199 $\geq$ 25) $\implies$ class=bad	This URL must not be kept in the test case
(id_121 $\geq$ 25) $\implies$ class=good	This URL must be kept in the test case
(id_204 $\leq$ 25) $\implies$ class=good	This URL must not be kept in the test case
(id_207 $\geq$ 25) $\implies$ class=bad	This URL must not be kept in the test case
(id_197 $\geq$ 25) $\implies$ class=bad	This URL must not be kept in the test case
(id_219 $\leq$ 25) $\implies$ class=bad	This URL must be kept in the test case
(id_179 $\leq$ 25) $\implies$ class=bad	This URL must be kept in the test case

Appendix B

Probabilities of being Good Calculated by BASIC

Table B.1: Probabilities assigned to the URLs of the application JPetStore by BASIC

URL(without domain name)	Probabilities
/actions/Catalog.action?viewCategory=&categoryId=FISH	0.0035851422
/actions/Catalog.action?viewCategory=&categoryId=DOGS	0.0035851422
/actions/Catalog.action?viewCategory=&categoryId=REPTILES	0.0030313855
/actions/Catalog.action?viewCategory=&categoryId=CATS	0.0335391883
/actions/Catalog.action?viewCategory=&categoryId=BIRDS	0.0085997919
/actions/Catalog.action?viewProduct=&productId=FI-SW-01	0.0058498864
/actions/Catalog.action?viewProduct=&productId=FI-SW-02	0.0080641044
/actions/Catalog.action?viewProduct=&productId=FI-FW-01	0.0085997919
/actions/Catalog.action?viewProduct=&productId=FI-FW-02	0.0501654526
/actions/Catalog.action?viewProduct=&productId=K9-BD-01	0.0442275011
/actions/Catalog.action?viewProduct=&productId=K9-PO-02	0.0035851422
/actions/Catalog.action?viewProduct=&productId=K9-DL-01	0.0085997919
/actions/Catalog.action?viewProduct=&productId=K9-RT-01	0.0085997919
/actions/Catalog.action?viewProduct=&productId=K9-RT-02	0.0917311133
/actions/Catalog.action?viewProduct=&productId=K9-CW-01	0.0030313855
/actions/Catalog.action?viewProduct=&productId=RP-SN-01	0.0035851422
/actions/Catalog.action?viewProduct=&productId=RP-LI-02	0.2579937562
/actions/Catalog.action?viewProduct=&productId=FL-DSH-01	0.0085997919
/actions/Catalog.action?viewProduct=&productId=FL-DLH-02	0.0085997919
/actions/Catalog.action?viewProduct=&productId=AV-CB-01	0.0085997919
/actions/Catalog.action?viewProduct=&productId=AV-SB-02	0.0335391883
/actions/Catalog.action?viewItem=&itemId=EST-1	0.0085997919
/actions/Catalog.action?viewItem=&itemId=EST-2	0.0030313855
/actions/Catalog.action?viewItem=&itemId=EST-3	0.1332967741
/actions/Catalog.action?viewItem=&itemId=EST-4	0.0035851422
/actions/Catalog.action?viewItem=&itemId=EST-5	0.0035851422
/actions/Catalog.action?viewItem=&itemId=EST-6	0.0085997919
/actions/Catalog.action?viewItem=&itemId=EST-7	0.0035851422
/actions/Catalog.action?viewItem=&itemId=EST-8	0.0085997919
/actions/Catalog.action?viewItem=&itemId=EST-9	0.0085997919
/actions/Catalog.action?viewItem=&itemId=EST-10	0.0085997919
/actions/Catalog.action?viewItem=&itemId=EST-11	0.0085997919
/actions/Catalog.action?viewItem=&itemId=EST-12	0.0085997919
/actions/Catalog.action?viewItem=&itemId=EST-13	0.0030313855
/actions/Catalog.action?viewItem=&itemId=EST-14	0.0035851422
/actions/Catalog.action?viewItem=&itemId=EST-15	0.0085997919
/actions/Catalog.action?viewItem=&itemId=EST-16	0.0030313855
/actions/Catalog.action?viewItem=&itemId=EST-17	0.0030313855
/actions/Catalog.action?viewItem=&itemId=EST-18	0.0085997919
/actions/Catalog.action?viewItem=&itemId=EST-19	0.0035851422
/actions/Catalog.action?viewItem=&itemId=EST-20	0.0035851422

/actions/Catalog.action?viewItem=&itemId=EST-21	0.0085997919
/actions/Catalog.action?viewItem=&itemId=EST-22	0.0442275011
/actions/Catalog.action?viewItem=&itemId=EST-23	0.0085997919
/actions/Catalog.action?viewItem=&itemId=EST-24	0.0030313855
/actions/Catalog.action?viewItem=&itemId=EST-25	0.0335391883
/actions/Catalog.action?viewItem=&itemId=EST-26	0.0035851422
/actions/Catalog.action?viewItem=&itemId=EST-27	0.0085997919
/actions/Catalog.action?viewItem=&itemId=EST-28	0.0397740374

Table B.2: Probabilities assigned to the URLs of the application MVC Music Store by BASIC

url	**Probabilities**
http://localhost:80/music/Store/Browse?Genre=Rock	0.0187777136
http://localhost:80/music/Store/Browse?Genre=Classical	6.36E-004
http://localhost:80/music/Store/Browse?Genre=Jazz	0.0184336089
http://localhost:80/music/Store/Browse?Genre=Pop	4.97E-004
http://localhost:80/music/Store/Browse?Genre=Disco	0.0160936969
http://localhost:80/music/Store/Browse?Genre=Latin	8.84E-004
http://localhost:80/music/Store/Browse?Genre=Metal	3.16E-004
http://localhost:80/music/Store/Browse?Genre=Alternative	0.0135587923
http://localhost:80/music/Store/Browse?Genre=Reggae	1.93E-005
http://localhost:80/music/Store/Browse?Genre=Blue	6.36E-004
http://localhost:80/music/Store/Details?id=1	8.84E-004
http://localhost:80/music/Store/Details?id=2	8.84E-004
http://localhost:80/music/Store/Details?id=3	1.66E-005
http://localhost:80/music/Store/Details?id=4	3.16E-004
http://localhost:80/music/Store/Details?id=5	1.31E-005
http://localhost:80/music/Store/Details?id=6	0.0063162076
http://localhost:80/music/Store/Details?id=7	6.36E-004
http://localhost:80/music/Store/Details?id=8	4.65E-005
http://localhost:80/music/Store/Details?id=9	8.84E-004
http://localhost:80/music/Store/Details?id=10	1.51E-005
http://localhost:80/music/Store/Details?id=11	3.16E-004
http://localhost:80/music/Store/Details?id=12	4.97E-004
http://localhost:80/music/Store/Details?id=13	7.53E-005
http://localhost:80/music/Store/Details?id=14	6.36E-004
http://localhost:80/music/Store/Details?id=15	7.05E-004
http://localhost:80/music/Store/Details?id=16	3.16E-004
http://localhost:80/music/Store/Details?id=17	4.97E-004
http://localhost:80/music/Store/Details?id=18	4.65E-005
http://localhost:80/music/Store/Details?id=19	0.0093339512
http://localhost:80/music/Store/Details?id=20	0.0135587923
http://localhost:80/music/Store/Details?id=21	1.90E-005
http://localhost:80/music/Store/Details?id=22	2.59E-005
http://localhost:80/music/Store/Details?id=23	4.89E-005
http://localhost:80/music/Store/Details?id=24	6.36E-004
http://localhost:80/music/Store/Details?id=25	6.51E-005
http://localhost:80/music/Store/Details?id=26	2.31E-005
http://localhost:80/music/Store/Details?id=27	0.0389078387
http://localhost:80/music/Store/Details?id=28	3.16E-004
http://localhost:80/music/Store/Details?id=29	1.37E-005

http://localhost:80/music/Store/Details?id=30	0.0226120232
http://localhost:80/music/Store/Details?id=31	6.36E-004
http://localhost:80/music/Store/Details?id=32	0.0098309913
http://localhost:80/music/Store/Details?id=33	3.16E-004
http://localhost:80/music/Store/Details?id=34	3.16E-004
http://localhost:80/music/Store/Details?id=35	8.83E-007
http://localhost:80/music/Store/Details?id=36	8.83E-007
http://localhost:80/music/Store/Details?id=37	2.31E-005
http://localhost:80/music/Store/Details?id=38	0.0313031248
http://localhost:80/music/Store/Details?id=39	2.71E-005
http://localhost:80/music/Store/Details?id=40	3.08E-005
http://localhost:80/music/Store/Details?id=41	8.84E-004
http://localhost:80/music/Store/Details?id=42	6.36E-004
http://localhost:80/music/Store/Details?id=43	4.43E-005
http://localhost:80/music/Store/Details?id=44	0.0117481461
http://localhost:80/music/Store/Details?id=45	1.75E-006
http://localhost:80/music/Store/Details?id=46	3.16E-004
http://localhost:80/music/Store/Details?id=47	6.36E-004
http://localhost:80/music/Store/Details?id=48	2.28E-005
http://localhost:80/music/Store/Details?id=49	1.75E-006
http://localhost:80/music/Store/Details?id=50	1.75E-006
http://localhost:80/music/Store/Details?id=51	4.97E-004
http://localhost:80/music/Store/Details?id=52	6.36E-004
http://localhost:80/music/Store/Details?id=53	8.72E-005
http://localhost:80/music/Store/Details?id=54	6.36E-004
http://localhost:80/music/Store/Details?id=55	3.15E-005
http://localhost:80/music/Store/Details?id=56	2.36E-005
http://localhost:80/music/Store/Details?id=57	0.0117481461
http://localhost:80/music/Store/Details?id=58	0.0262333155
http://localhost:80/music/Store/Details?id=59	4.59E-005
http://localhost:80/music/Store/Details?id=60	8.84E-004
http://localhost:80/music/Store/Details?id=61	1.75E-006
http://localhost:80/music/Store/Details?id=62	8.84E-004
http://localhost:80/music/Store/Details?id=63	2.61E-006
http://localhost:80/music/Store/Details?id=64	8.84E-004
http://localhost:80/music/Store/Details?id=65	8.84E-004
http://localhost:80/music/Store/Details?id=66	8.84E-004
http://localhost:80/music/Store/Details?id=67	0.0147110217
http://localhost:80/music/Store/Details?id=68	4.59E-005
http://localhost:80/music/Store/Details?id=69	0.0135587923
http://localhost:80/music/Store/Details?id=70	8.84E-004
http://localhost:80/music/Store/Details?id=71	8.83E-007
http://localhost:80/music/Store/Details?id=72	7.39E-005
http://localhost:80/music/Store/Details?id=73	1.52E-005
http://localhost:80/music/Store/Details?id=74	0.0117481461
http://localhost:80/music/Store/Details?id=75	8.84E-004
http://localhost:80/music/Store/Details?id=76	8.84E-004
http://localhost:80/music/Store/Details?id=77	1.37E-005
http://localhost:80/music/Store/Details?id=78	4.97E-004
http://localhost:80/music/Store/Details?id=79	4.41E-005
http://localhost:80/music/Store/Details?id=80	2.88E-005
http://localhost:80/music/Store/Details?id=81	8.84E-004
http://localhost:80/music/Store/Details?id=82	1.39E-005

http://localhost:80/music/Store/Details?id=83	8.84E-004
http://localhost:80/music/Store/Details?id=84	8.83E-007
http://localhost:80/music/Store/Details?id=85	4.59E-005
http://localhost:80/music/Store/Details?id=86	4.97E-004
http://localhost:80/music/Store/Details?id=87	6.36E-004
http://localhost:80/music/Store/Details?id=88	6.36E-004
http://localhost:80/music/Store/Details?id=89	1.37E-005
http://localhost:80/music/Store/Details?id=90	6.36E-004
http://localhost:80/music/Store/Details?id=91	1.51E-005
http://localhost:80/music/Store/Details?id=92	8.84E-004
http://localhost:80/music/Store/Details?id=93	8.83E-007
http://localhost:80/music/Store/Details?id=94	8.84E-004
http://localhost:80/music/Store/Details?id=95	2.28E-005
http://localhost:80/music/Store/Details?id=96	6.36E-004
http://localhost:80/music/Store/Details?id=97	6.36E-004
http://localhost:80/music/Store/Details?id=98	3.90E-005
http://localhost:80/music/Store/Details?id=99	4.54E-005
http://localhost:80/music/Store/Details?id=100	8.84E-004
http://localhost:80/music/Store/Details?id=101	1.75E-006
http://localhost:80/music/Store/Details?id=102	8.84E-004
http://localhost:80/music/Store/Details?id=103	2.92E-005
http://localhost:80/music/Store/Details?id=104	2.59E-005
http://localhost:80/music/Store/Details?id=105	2.59E-005
http://localhost:80/music/Store/Details?id=106	3.90E-005
http://localhost:80/music/Store/Details?id=107	3.76E-005
http://localhost:80/music/Store/Details?id=108	2.97E-005
http://localhost:80/music/Store/Details?id=109	6.36E-004
http://localhost:80/music/Store/Details?id=110	6.36E-004
http://localhost:80/music/Store/Details?id=111	2.04E-005
http://localhost:80/music/Store/Details?id=112	8.84E-004
http://localhost:80/music/Store/Details?id=113	0.0117481461
http://localhost:80/music/Store/Details?id=114	0.0153694385
http://localhost:80/music/Store/Details?id=115	1.51E-005
http://localhost:80/music/Store/Details?id=116	0.0236984109
http://localhost:80/music/Store/Details?id=117	5.28E-005
http://localhost:80/music/Store/Details?id=118	7.05E-004
http://localhost:80/music/Store/Details?id=119	8.84E-004
http://localhost:80/music/Store/Details?id=120	0.0769314084
http://localhost:80/music/Store/Details?id=121	8.84E-004
http://localhost:80/music/Store/Details?id=122	0.0103901615
http://localhost:80/music/Store/Details?id=123	3.16E-004
http://localhost:80/music/Store/Details?id=124	3.16E-004
http://localhost:80/music/Store/Details?id=125	4.43E-005
http://localhost:80/music/Store/Details?id=126	8.83E-007
http://localhost:80/music/Store/Details?id=127	0.0072215307
http://localhost:80/music/Store/Details?id=128	0.0077976454
http://localhost:80/music/Store/Details?id=129	6.36E-004
http://localhost:80/music/Store/Details?id=130	2.28E-005
http://localhost:80/music/Store/Details?id=131	0.0769314084
http://localhost:80/music/Store/Details?id=132	3.16E-004
http://localhost:80/music/Store/Details?id=133	0.0198960539
http://localhost:80/music/Store/Details?id=134	2.07E-005
http://localhost:80/music/Store/Details?id=135	3.16E-004

http://localhost:80/music/Store/Details?id=136	0.0093339512
http://localhost:80/music/Store/Details?id=137	0.0110238876
http://localhost:80/music/Store/Details?id=138	0.0135587923
http://localhost:80/music/Store/Details?id=139	0.0769314084
http://localhost:80/music/Store/Details?id=140	6.36E-004
http://localhost:80/music/Store/Details?id=141	8.84E-004
http://localhost:80/music/Store/Details?id=142	6.36E-004
http://localhost:80/music/Store/Details?id=143	1.92E-005
http://localhost:80/music/Store/Details?id=144	2.31E-005
http://localhost:80/music/Store/Details?id=145	5.80E-005
http://localhost:80/music/Store/Details?id=146	8.84E-004
http://localhost:80/music/Store/Details?id=147	3.16E-004
http://localhost:80/music/Store/Details?id=148	7.39E-005
http://localhost:80/music/Store/Details?id=149	3.15E-005
http://localhost:80/music/Store/Details?id=150	6.36E-004
http://localhost:80/music/Store/Details?id=151	3.16E-004
http://localhost:80/music/Store/Details?id=152	6.36E-004
http://localhost:80/music/Store/Details?id=153	8.83E-007
http://localhost:80/music/Store/Details?id=154	1.32E-006
http://localhost:80/music/Store/Details?id=155	1.93E-005
http://localhost:80/music/Store/Details?id=156	4.97E-004
http://localhost:80/music/Store/Details?id=157	1.34E-004
http://localhost:80/music/Store/Details?id=158	8.83E-007
http://localhost:80/music/Store/Details?id=159	6.36E-004
http://localhost:80/music/Store/Details?id=160	8.84E-004
http://localhost:80/music/Store/Details?id=161	1.68E-005
http://localhost:80/music/Store/Details?id=162	6.36E-004
http://localhost:80/music/Store/Details?id=163	6.36E-004
http://localhost:80/music/Store/Details?id=164	4.97E-004
http://localhost:80/music/Store/Details?id=165	0.0098309913
http://localhost:80/music/Store/Details?id=166	1.75E-006
http://localhost:80/music/Store/Details?id=167	8.83E-007
http://localhost:80/music/Store/Details?id=168	0.0098309913
http://localhost:80/music/Store/Details?id=169	1.92E-005
http://localhost:80/music/Store/Details?id=170	6.36E-004
http://localhost:80/music/Store/Details?id=171	2.55E-005
http://localhost:80/music/Store/Details?id=172	0.0160936969
http://localhost:80/music/Store/Details?id=173	0.012583829
http://localhost:80/music/Store/Details?id=174	3.15E-005
http://localhost:80/music/Store/Details?id=175	0.0389078387
http://localhost:80/music/Store/Details?id=176	8.83E-007
http://localhost:80/music/Store/Details?id=177	1.48E-005
http://localhost:80/music/Store/Details?id=178	6.96E-005
http://localhost:80/music/Store/Details?id=179	8.84E-004
http://localhost:80/music/Store/Details?id=180	0.0077976454
http://localhost:80/music/Store/Details?id=181	8.83E-007
http://localhost:80/music/Store/Details?id=182	3.16E-004
http://localhost:80/music/Store/Details?id=183	8.83E-007
http://localhost:80/music/Store/Details?id=184	2.61E-006
http://localhost:80/music/Store/Details?id=185	8.84E-004
http://localhost:80/music/Store/Details?id=186	8.83E-007
http://localhost:80/music/Store/Details?id=187	3.08E-005
http://localhost:80/music/Store/Details?id=188	1.75E-006

http://localhost:80/music/Store/Details?id=189	8.83E-007
http://localhost:80/music/Store/Details?id=190	8.83E-007
http://localhost:80/music/Store/Details?id=191	8.84E-004
http://localhost:80/music/Store/Details?id=192	0.01229134
http://localhost:80/music/Store/Details?id=193	0.0769314084
http://localhost:80/music/Store/Details?id=194	0.0160936969
http://localhost:80/music/Store/Details?id=195	4.97E-004
http://localhost:80/music/Store/Details?id=196	4.89E-005
http://localhost:80/music/Store/Details?id=197	0.0093339512
http://localhost:80/music/Store/Details?id=198	3.76E-005
http://localhost:80/music/Store/Details?id=199	8.84E-004
http://localhost:80/music/Store/Details?id=200	1.37E-005
http://localhost:80/music/Store/Details?id=201	8.84E-004
http://localhost:80/music/Store/Details?id=202	8.83E-007
http://localhost:80/music/Store/Details?id=203	4.65E-005
http://localhost:80/music/Store/Details?id=204	8.84E-004
http://localhost:80/music/Store/Details?id=205	1.31E-005
http://localhost:80/music/Store/Details?id=206	4.97E-004
http://localhost:80/music/Store/Details?id=207	4.54E-005
http://localhost:80/music/Store/Details?id=208	6.36E-004
http://localhost:80/music/Store/Details?id=209	6.36E-004
http://localhost:80/music/Store/Details?id=210	0.0093339512
http://localhost:80/music/Store/Details?id=211	4.71E-005
http://localhost:80/music/Store/Details?id=212	4.30E-005
http://localhost:80/music/Store/Details?id=213	6.36E-004
http://localhost:80/music/Store/Details?id=214	6.36E-004
http://localhost:80/music/Store/Details?id=215	2.40E-005
http://localhost:80/music/Store/Details?id=216	8.84E-004
http://localhost:80/music/Store/Details?id=217	6.36E-004
http://localhost:80/music/Store/Details?id=218	4.80E-005
http://localhost:80/music/Store/Details?id=219	8.84E-004
http://localhost:80/music/Store/Details?id=220	3.16E-004
http://localhost:80/music/Store/Details?id=221	6.36E-004
http://localhost:80/music/Store/Details?id=222	5.55E-005
http://localhost:80/music/Store/Details?id=223	0.008488983
http://localhost:80/music/Store/Details?id=224	8.84E-004
http://localhost:80/music/Store/Details?id=225	8.83E-007
http://localhost:80/music/Store/Details?id=226	6.36E-004
http://localhost:80/music/Store/Details?id=227	3.25E-005
http://localhost:80/music/Store/Details?id=228	8.84E-004
http://localhost:80/music/Store/Details?id=229	6.36E-004
http://localhost:80/music/Store/Details?id=230	6.36E-004
http://localhost:80/music/Store/Details?id=231	8.84E-004
http://localhost:80/music/Store/Details?id=232	3.46E-005
http://localhost:80/music/Store/Details?id=233	0.0135587923
http://localhost:80/music/Store/Details?id=234	0.0063162076
http://localhost:80/music/Store/Details?id=235	8.84E-004
http://localhost:80/music/Store/Details?id=236	2.31E-005
http://localhost:80/music/Store/Details?id=237	1.75E-006
http://localhost:80/music/Store/Details?id=238	8.69E-005
http://localhost:80/music/Store/Details?id=239	8.84E-004
http://localhost:80/music/Store/Details?id=240	0.006734049
http://localhost:80/music/Store/Details?id=241	6.36E-004

http://localhost:80/music/Store/Details?id=242	0.0177836334
http://localhost:80/music/Store/Details?id=243	8.83E-007
http://localhost:80/music/Store/Details?id=244	1.93E-005
http://localhost:80/music/Store/Details?id=245	8.84E-004
http://localhost:80/music/Store/Details?id=246	0.0072215307