On Improving (Non)Functional Testing

Qi Luo

Nanchang, Jiangxi, China

Master of Engineering, Tsinghua University, China, 2011
Bachelor of Engineering, Beihang University, China, 2008

A Dissertation presented to the Graduate Faculty of
The College of William & Mary in Candidacy for the Degree of
Doctor of Philosophy

Department of Computer Science

College of William & Mary
May, 2018

© Copyright by Qi Luo 2018

APPROVAL PAGE

This Dissertation is submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

Qi Luo

Approved by the Committee, December 2017

Committee Chair
Associate Professor Denys Poshyvanyk, Computer Science
College of William & Mary

Associate Professor Peter Kemper, Computer Science
College of William & Mary

Professor Evgenia Smirni, Computer Science
College of William & Mary

Assistant Professor Xu Liu, Computer Science
College of William & Mary

Associate Professor Massimiliano Di Penta, Dept. of Engineering
University of Sannio, Italy

ABSTRACT

Software testing is commonly classified into two categories, nonfunctional testing and functional testing. The goal of nonfunctional testing is to test nonfunctional requirements, such as performance and reliability. Performance testing is one of the most important types of nonfunctional testing, one goal of which is to detect the phenomena that an Application Under Testing (AUT) exhibits unexpectedly worse performance (e.g., lower throughput) with some input data. During performance testing, a critical challenge is to understand the AUT's behaviors with large numbers of combinations of input data and find the particular subset of inputs leading to performance bottlenecks. However, enumerating those particular inputs and identifying those bottlenecks are always laborious and intellectually intensive. In addition, for an evolving software system, some code changes may accidentally degrade performance between two software versions, it is even more challenging to find problematic changes (out of a large number of committed changes) may lead to performance regressions under certain test inputs. This dissertation presents a set of approaches to automatically find specific combinations of input data for exposing performance bottlenecks and further analyze execution traces to identify performance bottlenecks. In addition, this dissertation also provides an approach that automatically estimates the impact of code changes on performance degradation between two released software versions to identify the problematic ones likely leading to performance regressions.

Functional testing is used to test the functional correctness of AUTs. Developers commonly write test suites for AUTs to test different functionalities and locate functional faults. During functional testing, developers rely on some strategies to order test cases to achieve certain objectives, such as exposing faults faster, which is known as Test Case Prioritization (TCP). TCP techniques are commonly classified into two categories, dynamic and static techniques. A set of empirical studies has been conducted to examine and understand different TCP techniques, but there is a clear gap in existing studies. No study has compared static techniques against dynamic techniques and comprehensively examined the impact of test granularity, program size, fault characteristics, and the similarities in terms of fault detection on TCP techniques. Thus, this dissertation presents an empirical study to thoroughly compare static and dynamic TCP techniques in terms of effectiveness, efficiency, and similarity of uncovered faults at different granularities on a large set of real-world programs, and further analyze the potential impact of program size and fault characteristics on TCP evaluation. Moreover, in the prior work, TCP techniques have been typically evaluated against synthetic software defects, called mutants. For this reason, it is currently unclear whether TCP performance on mutants would be representative of the performance achieved on real faults. To answer this fundamental question, this dissertation presents the first empirical study that investigates TCP performance when applied to both real-world faults and mutation faults for understanding the representativeness of mutants.

TABLE OF CONTENTS

Acknowledgements vii

Dedication viii

List of Tables ix

List of Figures xiv

1 Introduction 2

2 FOREPOST: Finding Performance Problems Automatically with Feedback-Directed Learning Software Testing 9

2.1 Background and Motivation 13

2.1.1 State of the Art and Practice 13

2.1.2 A Motivating Example 14

2.2 The FOREPOST Approach 17

2.2.1 An Overview of FOREPOST 17

2.2.1.1 Obtaining Rules 17

2.2.1.2 Identifying Bottlenecks 19

2.2.2 Blind Source Separation 20

2.2.3 Independent Component Analysis 22

2.2.4 FOREPOST and FOREPOST$_{RAND}$ Architecture and Workflow 23

2.2.5 The Algorithm for Identifying Bottlenecks 26

2.3 Evaluation 27

2.3.1 Research Questions 27
2.3.2 Subject AUTs and Experimental Hardware 29
2.3.3 Research Question 1 31
2.3.4 Research Question 2 33
2.3.5 Research Question 3 35
2.4 Results 36
2.4.1 Research Question 1 36
2.4.2 Research Question 2 40
2.4.3 Research Question 3 45
2.5 Threats to Validity 47
2.5.1 Internal Validity 47
2.5.2 External validity 48
2.5.3 Construct Validity 49
2.6 Utilizing FOREPOST in Cloud Computing 49
2.7 Related Work 51
2.8 Conclusion and Discussion 54
2.9 Bibliographical Notes 55

3 Automating Performance Bottleneck Detection using Search-Based Application Profiling 56
3.1 Problem Statement 57
3.1.1 Background on Input-Sensitive Profiling 57
3.1.2 Analyzing Profile Data for Bottlenecks 58
3.1.3 The Problem Statement 60
3.2 Our Approach 61
3.2.1 Overview of GA-Prof 61
3.2.2 Using Genetic Algorithms in GA-Prof 62

3.2.2.1 Background on Genetic Algorithms 63
3.2.2.2 Why We Use Genetic Algorithms in GA-Prof 63
3.2.2.3 Automating Profiling Using GAs 64
3.2.3 Identifying Performance Bottlenecks 66
3.2.4 GA-Prof's Architecture and Workflow 68
3.3 Empirical Evaluation 69
3.3.1 Subject Applications 70
3.3.2 Methodology 70
3.3.3 Variables 72
3.3.4 Threats to Validity 73
3.4 Empirical Results 74
3.4.1 Searching Through Input Combinations 75
3.4.2 Understanding Performance Bottlenecks 77
3.4.3 Comparing GA-Prof to FOREPOST 80
3.5 Related Work 81
3.6 Conclusion and Discussion 83
3.7 Bibliographical Notes 84

4 Mining Performance Regression Inducing Code Changes in Evolving Software 85
4.1 Problem Statement 87
4.1.1 State of the Art and Practice 88
4.1.2 An Example Performance Regression 89
4.1.3 The Problem Statement 90
4.2 Approach 91
4.2.1 An Overview of Our Approach 91
4.2.2 Search-based Input Profiling for Performance Regressions . 92

4.2.3 Identifying Performance Regression Inducing Changes via Mining . . . 94
4.2.4 Workflow of PerfImpact . . . 97
4.3 Evaluation . . . 99
4.3.1 Research Questions . . . 99
4.3.2 Subject AUTs . . . 100
4.3.3 Methodology . . . 100
4.4 Empirical Results . . . 103
4.4.1 Finding Performance Regression Inputs . . . 103
4.4.2 Identifying Code Changes . . . 105
4.5 Threats to Validity . . . 108
4.6 Related Work . . . 110
4.7 Conclusion and Discussion . . . 111
4.8 Bibliographical Notes . . . 113

5 How Do Static and Dynamic Test Case Prioritization Techniques Perform on Modern Software Systems? An Extensive Study on GitHub Projects 114
5.1 Background & Related Work . . . 119
5.1.1 Static TCP Techniques . . . 121
5.1.2 Dynamic TCP Techniques . . . 123
5.1.3 Empirical studies on TCP techniques . . . 125
5.1.4 Mutation Analysis . . . 127
5.1.5 Metrics for TCP techniques . . . 129
5.2 Empirical Study . . . 130
5.2.1 Research Questions (RQs): . . . 130
5.2.2 Subject Programs, Test Suites and Faults . . . 131
5.2.3 Design of the Empirical Study . . . 133

5.2.4 Tools and Experimental Hardware 141
5.3 Results 142
5.3.1 RQ_1 & RQ_2 & RQ_3: Effectiveness of Studied Techniques Measured by APFD and APFDc at Different Granularities 143
5.3.1.1 Results at Test Class Level 143
5.3.1.2 Results at Test Method Level 147
5.3.2 Impact of Subject Program's Size 151
5.3.3 Impact of Software Evolution 152
5.3.4 Impact of Mutant Quantities on TCP Effectiveness 156
5.3.5 Impact of Mutant Types on TCP Effectiveness 158
5.3.6 Similarity between Uncovered Faults for Different TCP techniques 161
5.3.7 Efficiency of Static TCP Techniques 164
5.4 Threats to Validity 165
5.5 Lessons Learned 168
5.6 Conclusion and Discussion 170
5.7 Bibliographical Notes 171
6 Assessing Test Case Prioritization on Mutants and Real Faults 172
6.1 Background & Related Work 176
6.1.1 TCP Problem Formulation 176
6.1.2 Studied TCP Techniques 177
6.1.3 Threats to the Validity of Mutation-Based TCP Performance Evaluations 177
6.1.4 Studies Examining the Relationship Between Mutants and Real Faults 178
6.2 Empirical Study 180

6.2.1 Research Questions (RQs): 180
6.2.2 Study Context 181
6.2.3 Methodology 183
6.2.3.1 RQ_1: TCP Effectiveness on Real Faults 184
6.2.3.2 RQ_2: Representativeness of Mutants 185
6.2.3.3 RQ_3: Effects of Fault Properties 186
6.2.4 Experiment Tools and Hardware 188
6.2.4.1 Mutation Analysis 188
6.2.4.2 Implementation of TCP Techniques 188
6.2.4.3 Hardware 189
6.3 Results 189
6.3.1 RQ_1: TCP Effectiveness on Real Faults 189
6.3.2 RQ_2: Representativeness of Mutants 190
6.3.3 RQ_3: Effects of Fault Properties 192
6.3.3.1 Effects of Coupling Between Mutants and Real Faults 192
6.3.3.2 Effect of Different Mutation Operators 193
6.4 Threats to Validity 195
6.5 Lessons Learned 198
6.6 Conclusion and Discussion 200
6.7 Bibliographical Notes 200

7 Conclusion 201

ACKNOWLEDGEMENTS

I would like to thank my advisor Denys Poshyvanyk, who guided me, supported me, and encouraged me during my whole Ph.D. life. I will never forget the moments when he guided me through the challenges of being a Ph.D.: wrote a research paper, prepared a research presentation, and tackled the various obstacles I faced in my research. He helped me become a qualified independent researcher and encouraged me during the tough times. Thank you, Denys, for everything.

I would like to thank my committee members, Massimiliano Di Penta, Xu Liu, Peter Kemper, and Evgenia Smirni. Thank you for your valuable suggestions, questions, and feedback which helped me improve my work significantly.

I would like to thank my collaborators: Lingming Zhang, Kevin Moran, Mark Grechanik, Du Shen, and also all the SEMERU members. Thank you for all your comments, discussions, and advices to help me make my work stronger.

Finally, my deepest gratitude goes to my parents, Xiaoping Luo and Luanli Jin. Thank you for your accomplishment, support, encouragement, and endless love. I would not be the person I am today without you.

To my parents

LIST OF TABLES

2.1 Characteristics of the insurance application Renters. 29
2.2 Independent variables in sensitivity analysis. 35
2.3 Selected rules that are learned for **R**enters and **J**PetStore. 36
2.4 Precision for FOREPOST when n_u=5 and n_p=10 44
2.5 Recall for FOREPOST when n_u=5 and n_p=10 44
2.6 F-score for FOREPOST when n_u=5 and n_p=10 44
3.1 Comparing GA-Prof and FOREPOST for detecting performance bottlenecks in JPetStore (JP) and DellDVDStore (DS). All numbers are averaged over multiple runs. "# of Methods" indicates the number of injected bottlenecks that are captured by one certain technique. "Final Ranks" indicates the ranks of injected bottlenecks in the final ranked list. 80
4.1 The stats of the subject programs. 100
4.2 The *time difference* between two versions for random inputs (Rd) and PerfImpact selected inputs (PI) in JPetStore (JP) and Agilefant (AF). 105
4.3 Examples of code changes in Agilefant. 107
4.4 Performance regression testing approaches. 112
5.1 The List of Contributions 120

5.2 The stats of the subject programs: Size: #Loc; TM: #test cases at method level; TC: #test cases at class level; All: #all mutation faults; Detected: #faults can be detected by test cases. 127

5.3 Muation Operators Used 132

5.4 Studied TCP Techniques 134

5.5 Results for the ANOVA and Tukey HSD tests on the average APFD and APFDc values at test-class level, which are depicted in Figure 5.1. The last column shows the results for Kendall tau Rank Correlation Coefficient τ_b between the average APFDc and average APFD.141

5.6 Results for the ANOVA, and Tukey HSD tests on the average APFD and APFDc values at test-method level, which are depicted in Figure 5.2. The last column shows the results for Kendall tau Rank Correlation Coefficient τ_b between the average APFDc and average APFD. 141

5.7 The results of Wilcoxon signed rank test on the average APFD values for each pair of TCP techniques. The techniques T1 to T9 refer to TP$_{cg-tot}$, TP$_{cg-add}$, TP$_{str}$, TP$_{topic-r}$, TP$_{topic-m}$, TP$_{total}$, TP$_{add}$, TP$_{art}$, TP$_{search}$ respectively. For each pair of TCP techniques, there are two sub-cells. The first one refers to the p-value at test-class level and the second one refers to the p-value at test-method level. The p-values are classfied into three categories, 1) $p>0.05$, 2) $0.01<p<0.05$, 3) $p<0.01$. The p-values for categories $p>0.05$ and $p<0.01$ are presented as $p>0.05$ and $p<0.01$ respectively. If a p-value is less than 0.05, the corresponding cell is shaded. 146

5.8 The results of Wilcoxon signed rank test on the average APFDc values for each pair of TCP techniques. This table follows exactly the same format as Table 5.7. 146

5.9 Results for the ANOVA and Tukey HSD tests on the average APFD and APFDc values at test-class level across smaller subject programs. The last column shows the results for Kendall tau Rank Correlation Coefficient τ_b between the average APFDc and average APFD. 150
5.10 Results for the ANOVA and Tukey HSD tests on the average APFD and APFDc values at test-class level across larger subject programs. The last column shows the results for Kendall tau Rank Correlation Coefficient τ_b between the average APFDc and average APFD. . . 151
5.11 Results for the ANOVA and Tukey HSD tests on the average APFD and APFDc values at test-method level across smaller subject programs. The last column shows the results for Kendall tau Rank Correlation Coefficient τ_b between the average APFDc and average APFD. 151
5.12 Results for the ANOVA and Tukey HSD tests on the average APFD and APFDc values at test-method level across larger subject programs. The last column shows the results for Kendall tau Rank Correlation Coefficient τ_b between the average APFDc and average APFD. 152
5.13 Results for average APFD values on different sizes of mutation faults. The last column shows the results for Kendall tau Rank Correlation Coefficient τ_b between the average APFD values with different sizes of mutation faults and the average APFD values shown in Tables 5.5 and 5.6. 156
5.14 Results for average APFDc values on different sizes of mutation faults. This table follows the same format as Table 5.13. 157

5.15 Results for average APFD values on different types of mutation faults. The last column shows the results for Kendall tau Rank Correlation Coefficient τ_b between the average APFD values with different types of mutation faults and the average APFD values shown in Tables 5.5 and 5.6. 158
5.16 Results for average APFDc values on different types of mutation faults. The last column shows the results for Kendall tau Rank Correlation Coefficient τ_b between the average APFDc values with different types of mutation faults and the average APFDc values shown in Tables 5.5 and 5.6. 159
5.17 The classification of subjects on different granularities using Jaccard distance. The four values in each cell are the numbers of subject projects, the faults of which detected by two techniques are highly dissimilar, dissimilar, similar and highly similar respectively. The technique enumeration is consistent with Table 5.7. 164
5.18 The classification of subjects on different granularities using Jaccard distance. The four values in each cell are the numbers of subject projects, the faults of which detected by two techniques are highly dissimilar, dissimilar, similar and highly similar respectively. The technique enumeration is consistent with Table 5.7. 165
5.19 Execution costs for the static TCP techniques. The table lists the average, min, max, and sum of costs across all subject programs for both test-class level and test-method level (i.e., cost at test-class level/cost at test-method level). Time is measured in second. . . . 165

6.1 The stats of the subject programs: #Real: #real-world faults; #All: #all mutation faults; #Detected: #mutation faults can be detected by test cases; #Subsuming: subsuming mutants. 181
6.2 Studied TCP Techniques. 183
6.3 Average APFD & APFDc values for all eight TCP techniques, for both real, mutation fault and subsuming mutation fault detection, across all subject programs. Additionally, the grouping results for the Tukey HSD test are shown in capitalized letters (*e.g.,* AB). S.Mutants refers to subsuming mutants. 188
6.4 Results of the ANOVA analysis and the Kendall τ_b Coefficient for the overall APFD(c) values shown in Table 6.3. 190
6.5 Results for the Kendall τ_b Rank Correlation Coefficient between APFD(c) values for TCP techniques on detecting mutation faults and detecting each type of real faults described in Section 6.2.3.3. 190
6.6 Results for the Kendall τ_b Rank Correlation Coefficient between APFD(c) values for TCP techniques on detecting real faults and detecting each type of mutation faults. 194

LIST OF FIGURES

2.1 A speech model of blind source separation. 20

2.2 Schematics of the ICA matrix decomposition. 23

2.3 The architecture and workflow of FOREPOST and FOREPOST$_{RAND}$. FOREPOST does not contain the step 14. 24

2.4 The summary of the results for Empirical Study 1. 37

2.5 Average execution times (in second) for different groups of injected bottlenecks (i.e.,bottlenecks#1 and bottlenecks#2), where n_p = 10 and n_u = 5. 40

2.6 Average execution times (in second) when controlling different independent variables. 42

2.7 Comparison between FOREPOST using uniform or different bottlenecks for JPetStore. The red boxplots refer to bottlenecks#1. The green boxplots refer to bottlenecks#2. 44

2.8 Average execution times (in second) for FOREPOST and FORPOST$_{RAND}$, where n_p = 10 and n_u = 5. 45

2.9 Comparison between FOREPOST and FOREPOST$_{RAND}$ for JPetStore. 47

2.10 Comparison between FOREPOST and FOREPOST$_{RAND}$ for Dell DVD Store. 47

3.1 A pseudocode example of input-sensitive profiling. 58

3.2 The architecture and workflow of GA−Prof. 69

3.3 Execution elapsed time measured in seconds for subject AUTs. We compare average elapsed times of each transaction in first and last generations for each application. The x-axis corresponds to the first and last generations, and y-axis corresponds to systems' average elapsed time. The results for all three subject applications are averaged over 30 runs. Subfigure (a), (b) and (c) corresponds to JPetStore, DellDVDStore and Agilefant, respectively. 76
3.4 The results for elapsed execution time across every generation for each application, measured in seconds. The x-axis corresponds to generations, and y-axis corresponds to average elapsed time. Subfigure (a), (b) and (c) corresponds to JPetStore, DellDVDStore and Agilefant, respectively. 77
3.5 Distribution of the quantity of captured injected bottlenecks. The x-axis corresponds to the number of injected bottlenecks that are captured by one certain GA−Prof run. The y-axis corresponds to the number of GA−Prof runs. Subfigure (a), (b) and (c) corresponds to JPetStore, DellDVDStore and Agilefant, respectively. 78
3.6 Understanding the trend of ranks of injected bottlenecks. The x-axis corresponds to generations, and y-axis corresponds to the rank of bottlenecks. In each subfigure, the rank of the method is shown in black circles. The standard deviation at each generations is shown in black vertical lines and whiskers. The fit straight line is shown is blue dashed lines. 79

4.1 A performance regression example due to possible thread blocking. 89
4.2 Examples of URLs and a chromosome in our GA implementation. Each number in the chromosome refers to a unique URL ID. 92

4.3 The examples of GA operators, crossover and mutation. 93
4.4 Three sample execution traces of an AUT. 96
4.5 The workflow of PerfImpact. 98
4.6 The box-and-whisker plots represent *time differences* between two released versions across generations on JPetStore (JP) and Agilefant (AG). 103
4.7 The box-and-whisker plots represent the ranks of the changes in Table 4.3. The x-axis represents the generations, and the y-axis represents the ranks. Smaller values that appear on y-axis imply higher ranks. 104
4.8 The figures show the average of total execution times of the changes in Table 4.3. This total execution time of one change is the total execution time of all methods in its respective impact set. The blue dots show the average of total execution time in old version of Agilefant ($v_{3.2}$), and the red dots show the average of total execution time in new version of Agilefant ($v_{3.3}$ or $v_{3.5}$). The curves are the fitting curves generated using Polynomial Function model. The inputs were selected in the last generation. The x-axis represents the average of total execution time, and the y-axis represents the number of users. Time is measured in seconds. 105
4.9 Examples of code changes in Agilefant. (a) shows the source code of change (f) in Table 4.3, and (b) shows the source code of change (d) in Table 4.3. 106

5.1 The box-and-whisker plots represent the values of APFD and APFDc for different TCP techniques at test-class level. The x-axis represents the APFD and APFDc values. The y-axis represents the different techniques. The central box of each plot represents the values from the lower to upper quartile (i.e., 25 to 75 percentile). . . . 143
5.2 The box-and-whisker plots represent the values of APFD and APFDc for different TCP techniques at test-method level. The x-axis represents the APFD and APFDc values. The y-axis represents the different techniques. The central box of each plot represents the values from the lower to upper quartile (i.e., 25 to 75 percentile). . . 144
5.3 The box-and-whisker plots represent the values of APFDc for different TCP techniques at different test granularities. The x-axis represents the APFDc values. The y-axis represents the different techniques. The central box of each plot represents the values from the lower to upper quartile (i.e., 25 to 75 percentile). 145
5.4 Test-Class-level test prioritization in evolution 153
5.5 Test-Method-level test prioritization in evolution 154
5.6 Average Jaccard similarity of faults detected between static and dynamic techniques across all subjects at method and class-level granularity. 161
5.7 Counts and percentage for different types of mutation faults across all subjects at cut point 10% for class-level granularity. The types of mutation faults are classified based on the mutation operators shown in Table 5.3. 162

5.8 Counts and percentage for different types of mutation faults across all subjects at cut point 10% for method-level granularity. The types of mutation faults are classified based on the mutation operators shown in Table 5.3. 162

6.1 APFD(c) values for TCP techniques in terms of detecting different types of real faults. 192
6.2 Examples of bug fixing changes. 194
6.3 Average APFD(c) values across different mutation operators referenced as: NC = NegateConditional, RC = RemoveConditional, CC = ConstructorCall, NVM = NonVoidMethodCall, M = Math, MV = MemberVariable, IC = InlineConstant, I = Increments, AP = ArgumentPropagation, CB = ConditionalsBoundary, S = Switch, VMC = VoidMethodCall, IN = InvertNegs, RV = ReturnVals, and RI = RemoveIncrements. 196

On Improving (Non)Functional Testing

Chapter 1

Introduction

Software testing is one of the most important tasks in software evolution and maintenance. Developers commonly perform two types of testing: functional testing [192, 307, 130, 100] and nonfunctional testing [56, 151, 50, 312, 13]. This dissertation introduces a set of testing approaches and empirical studies to understand and improve both of those two types of testing.

The goal of nonfunctional testing is to test non-functional requirements, such as reliability and scalability [9]. There are many types of non-functional testing, like load testing [85, 211], performance testing [179, 213, 98, 135], stress testing [262], etc. In this dissertation, I focus on improving performance testing, the goal of which is to identify the phenomena that an Application Under Testing (AUT) exhibits unexpectedly worse performances (e.g., longer execution time or lower throughput) [196, 197, 114]. In performance testing, software engineers commonly rely on profilers, i.e., tools that insert instructions into the AUT to obtain frequency, memory usage, and elapsed execution time of method calls. When profiling, software engineers perform the following actions: 1) instrument the AUT with a profiler and run the instrumented AUT using some input values and 2) from the collected measurements, determine which methods are responsible for excessive execution time and resource usage. Simply put, all AUT's methods are sorted in a descending order by their execution times and the top N methods on this list are declared bottlenecks

and investigated further by engineers.

A weakness of this process is that its success for detecting bottlenecks depends on the chosen set of input values for the AUT. A critical challenge is to profile nontrivial applications with large numbers of combinations of their input parameter values. Many nontrivial applications have complex logic that programmers express by using different control-flow statements, which are often deeply nested. In addition, these control-flow statements have branch conditions which contain expressions that use different variables whose values are computed using some input parameters. The complete performance analysis could be achieved if an AUT was profiled with all allowed combinations of values for its inputs. Unfortunately, this is often infeasible because of the enormous number of combinations; for example, 20 integer inputs whose values range from zero to nine give us 10^{20} combinations. It is arduous to choose specific values of input parameters to profile the executions of these applications in order to obtain bottlenecks. Moreover, this procedure is always manual, intellectually intensive and laborious. Also, its effectiveness is limited and it increases the cost of application development. To address this problem, input-sensitive profiling was introduced where the sizes of their inputs and the values of the input parameters are varied to uncover performance problems in the AUT [302, 66, 175]. However, little effort has been put into identifying the performance bottlenecks under those specific input values. In addition, during software evolution, the source code of a system frequently changes due to bug fixes or new feature requests, which may accidentally degrade the performance of a newly released software version. It is challenging to find problematic changes (out of a large number of committed changes) that may be responsible for performance regressions under certain test inputs.

When performing functional testing, developers test a method or a function to examine if the outputs are shown as expected [5]. There are different types of functional testing, such as regression testing [296, 70], smoke testing [207, 209], sanity testing [104], and usability testing [304, 97]. This dissertation focuses on Test Case Prioritization (TCP), one of the most important approaches of regression testing [306, 247, 192, 172]. The

goal of TCP is to reorder test cases for maximizing a pre-defined objective function, such as exposing faults earlier or reducing the execution time cost [193, 192]. TCP techniques are broadly classified into two groups: dynamic techniques [247, 247, 145, 184] and static techniques [311, 178, 266]. Typically, dynamic techniques apply a certain test prioritization algorithms (e.g., total and additional greedy algorithms, genetic algorithms) on runtime execution information (e.g., test coverage) to prioritize the test cases. While dynamic techniques can be powerful, they may not be applicable to some software systems due to absent coverage information or time-consuming process. Recently, researchers proposed a set of TCP techniques that rely on static information extracted from source code [311, 178, 266], and conducted a set of empirical studies to further analyze these TCP techniques [247, 92, 82, 243, 266].

However, there is a clear gap in the existing empirical studies: i) static TCP techniques have not been systematically studied against the dynamic techniques; ii) no study has comprehensively shown the impact of different test granularities, program sizes, fault characteristics, efficiency and the similarities in terms of uncovered faults on static techniques; iii) previous studies have not evaluated TCP techniques on a large set of real-world software systems; iv) no study has deeply analyzed the impact of software evolution on TCP domain. In addition, these TCP techniques typically are evaluated in terms of detecting synthetic program faults, called mutants, instead of real faults. The use of mutants in the evaluation of TCP approaches may lead to a potential threat to validity extracted from the mutant analysis. It is unclear whether the mutants are representative for the real faults in TCP evaluation. In essence, the correlation of the performance of TCP techniques in detecting mutants may not imply to the similar correlation in detecting real faults. But none of the prior research studies has investigated how TCP techniques perform on detecting real faults and whether the performance of TCP techniques on mutation fault detection is representative of their performance on real fault detection.

To minimize the existing gaps mentioned above, this dissertation i)proposes novel approaches, namely FOREPOST [196] (Chapter 2), GA-Prof [258] (Chapter 3), and Per-

fImpact [197] (Chapter 4), which utilize Machine Learning (ML) algorithms and Genetic Algorithms (GA) to find specific combinations of input data for exposing performance problems, and then use Independent Component Analysis (ICA) and Change Impact Analysis (CIA) to further analyze execution traces for identifying performance bottlenecks or the problematic code changes leading to performance bottlenecks; and ii) presents two large-scale empirical studies: one compares static and dynamic TCP techniques at different test case granularities on a set of real-world software systems [193, 194] (Chapter 5), and the other one compares the performance of TCP techniques in terms of detecting real faults and mutation faults in order to investigate the representativeness of mutants in TCP evaluation (Chapter 6). This work is supported in part by the NSF CCF-1218129.

In summary, this dissertation makes the following contributions:

- **Three Input-Sensitive Performance Testing Approaches (Nonfunctional Testing).** This dissertation proposes a set of novel approaches, namely FOREPOST (Chapter 2), GA-Prof (Chapter 3), and PerfImpact (Chapter 4) to find specific combinations of input data for exposing performance problems and to further analyze the corresponding execution traces in order to identify problematic code. In particular, FOREPOST and GA-Prof detect performance bottlenecks in single-version scenario, while PerfImpact identifies performance regressions in an evolving system. Moreover, we further utilize FOREPOST to build model for an AUT explored in the cloud and create rules for programmers to improve provisioning strategies that guide the cloud to (de)allocate resources to this AUT. This work was published in the 2015 International Symposium on Software Testing and Analysis (ISSTA) [258], the 13th International Conference on Mining Software Repositories (MSR) [197], the 7th ACM/SPEC on International Conference on Performance Engineering (ICPE) [116], and Empirical Software Engineering (EMSE) 2017 [196].

- **Two Empirical Studies for TCP Techniques (Functional Testing).** To the best of our knowledge, we conduct the first extensive empirical study (Chapter 5) aim-

ing at empirically evaluating four static TCP techniques and comparing them with four state-of-art dynamic TCP techniques at different test-case granularities (e.g., method and class-level) in terms of effectiveness, efficiency, and similarity of faults detected. This study was performed on 58 real-word Java programs encompassing 714 KLoC. It evaluates effectiveness of TCP in terms of two popular metrics (Average Percentage of Faults Detected (APFD) and its cost cognizant APFDc) to investigate the potential impact of different metrics on TCP evaluation. It also investigates the potential impacts of fault characteristics (e.g., fault quantities and types), program characteristics (e.g., program sizes) and software evolution on TCP domain.

In addition, this dissertation presents the first empirical study which compares the performance of TCP techniques in terms of real fault detection to their mutation-based performance to determine whether mutation faults are representative of real faults in TCP domain (Chapter 6). The context of our study includes eight well-studied TCP approaches, 75k+ mutation faults, and 357 real-world faults from five open-source Java programs in the Defects4J dataset. The impacts for both of mutant coupling and mutation operator types on TCP performance are examined in this study. We extract several meaningful learned lessons from these two studies which will help guide future research in TCP domain, especially regarding TCP and mutation analysis. Some of the key learned lessons are summarized as follows:

- Test granularity impacts the effectiveness of TCP techniques. In general, for both of APFD and APFDc values, TCP techniques on method-level granularity are able to detect faults sooner. Thus, using method-level granularity or even exploring finer granularities would help researchers and developers achieve better performance for TCP techniques.
- The performance of TCP techniques is not consistent across different subject programs. One TCP technique may outperform other techniques on some

subjects but perform worse on other subject programs. This finding implies that program characteristics may affect TCP performance and suggests that for a specific subject program it may be necessary to investigate the potential relationship between program characteristics and the performance of TCP techniques in order to choose the best one.

- The mutant characteristics (i.e., mutant quantity and type) do not impact the performance of TCP techniques in terms of APFD(c) values. Moreover, the program characteristics (e.g., program size) and the software evolution have little impact on TCP evaluation. This observation would guide researchers to design valid experimental settings in the future.
- The similarities of the top prioritized test cases from different TCP techniques are quite low. That is, there is only a small number of common faults detected by the top ranked test cases from different TCP techniques. This prompts further research into the potential for combining different types of TCP techniques (or strategies) to achieve better performance.
- In the evaluation of TCP techniques, the mutants may not be representative for real faults, implying that the conclusions of some prior mutant-based studies may not be realistic. We suggest that researchers and developers should revisit their previous mutant-based results and involving real faults for their future research.
- The representativeness of mutants varies across different TCP metrics (i.e., APFD(c)). In general, the correlation between real faults and mutants is quite low in terms of APFD values, but this correlation is median to strong when considering APFDc values. This finding suggests that researchers could use APFDc metrics to evaluate TCP techniques for mutant-based analysis to make the conclusion more realistic in the future.
- The program characteristics may determine the representativeness of mutants

in TCP domain. It is possible that building a fault model based on program characteristics and developing the types of mutation operators based on such fault model in order to improve the representativeness of mutants in TCP or even software testing domain.

The first study was published in the ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE) 2016 [193]. The extended version of this study is accepted at IEEE Transactions on Software Engineering (TSE) 2018 [194].

Chapter 2

FOREPOST: Finding Performance Problems Automatically with Feedback-Directed Learning Software Testing

The goal of performance testing is to find performance bottlenecks, when an *application under test (AUT)* unexpectedly exhibits worsened characteristics for a specific workload [217, 281]. One way to find performance bottlenecks effectively is to identify test cases for finding situations where an AUT suffers from unexpectedly high response time or low throughput [150, 36]. Test engineers construct performance test cases, and these cases include actions (e.g., interacting with GUI objects or invoking methods of exposed interfaces) as well as input test data for the parameters of these methods or GUI objects [138]. It is difficult to construct effective test cases that can find performance bottlenecks in a short period of time, since it requires test engineers to test many combinations of actions and input data for nontrivial applications.

Developers and testers need performance management tools for identifying performance bottlenecks automatically in order to achieve better performance of software while

keeping the cost of software maintenance low. In a survey of 148 enterprises, 92% said that improving application performance was a top priority [297, 252]. In a recent work, Zaman et al. performed a qualitative study that demonstrated that performance bottlenecks are not easy to reproduce and that developers spend more time working on them [300]. Moreover, Nistor et al. found that fixing performance bottlenecks is difficult and better tools for locating and fixing performance bottlenecks are needed by developers [230, 228, 188, 189, 187]. As a result, different companies work on tools to alleviate performance bottlenecks. The application performance management market is over 2.3 billion USD and growing at 12% annually, making it one of the fastest growing segments of the application services market [33, 103]. Existing performance management tools collect and structure information about executions of applications, so that stakeholders can analyze this information to obtain insight into performance. Unfortunately, none of these tools identifies performance bottlenecks automatically. The difficulty of comprehending the source code of large-scale applications and their high complexity lead to performance bottlenecks that result in productivity loss approaching 20% for different domains due to application downtime [117].

Considering that source code may not even be available for some components, engineers concentrate on black-box performance testing of the whole application, rather than focusing on standalone components [14, 140]. Depending on input values, an application can exhibit different behaviors with respect to resource consumption. Some of these behaviors involve intensive computations that are characteristic of performance bottlenecks [313]. Naturally, testers want to summarize the behavior of an AUT concisely in terms of its inputs. In this way, they can select input data that will lead to significantly increased resource consumption, thereby revealing performance bottlenecks. Unfortunately, finding proper rules that collectively describe properties of such input data is a highly creative process that involves deep understanding of input domains [18, page 152].

Descriptive rules for selecting test input data play a significant role in software testing [42], because these rules approximate the functionality of an AUT. For example, a rule for

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

We offer a novel solution for *Feedback-ORiEnted PerfOrmance Software Testing (FOREPOST)* by finding performance bottlenecks automatically through learning and using rules that describe classes of input data that lead to intensive computations. FOREPOST is an adaptive, feedback-directed learning testing system that learns rules from an AUT's execution traces. These rules provide insights into properties of test input data that lead to increased computational loads in applications, and are used to automatically select test input data for performance testing. Moreover, we introduce an alternative version, namely FOREPOST$_{RAND}$, which considers both random input data and the specific inputs based on generated descriptive rules. The intuition here is that we believe involving random input data would be helpful to cover more AUT execution paths, identifying more potential performance bottlenecks.

This chapter makes the following contributions:

- We propose a novel approach, FOREPOST, that collects and utilizes execution traces of the AUT to learn rules that describe the computational intensity of the workload in terms of the properties of the input data. These rules are used by an adaptive automated test script automatically, in a feedback loop, to steer the execution of the AUT by selecting input data based on the newly learned rules.

- We propose another version of the approach, FOREPOST$_{RAND}$, which combines of random input data with the input data selected by using learned rules. We expect that involving random input data would be useful to explore more potential computationally intensive executions.

- We propose a novel algorithm to identify performance bottlenecks, which have significant contributions to the executions with worse performance.

- We applied FOREPOST to two open-source application benchmarks, JPetStore and Dell DVD Store. FOREPOST automatically found rules that steered executions of JPetStore and Dell DVD Store towards input data that increased the average execution time by 78.2 % and 333.3 %, and identified more injected performance bottlenecks, as compared to random testing. We also conduct a controlled experiment to analyze the impact of independent variables on the power of FOREPOST to identify performance bottlenecks.

- We conducted a controlled experiment to compare FOREPOST to FOREPOST$_{RAND}$. The experimental results demonstrate that FOREPOST$_{RAND}$ helps improve the accuracy of identifying bottlenecks at the expense of finding less computationally expensive bottlenecks.

- We introduced a framework, namely PRESTO, for performance testing in cloud. It utilizes FOREPOST to automatically build behavioral models for AUTs during performance testing to understand the relationship between behaviors with cloud provided resources, and recommend provisioning strategies that guide the cloud to (de)allocate resources for AUTs. The experimental results show that PRESTO is able to create rules for provisioning resources to maintain AUT's throughputs at the desired level.

2.1 Background and Motivation

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

2.1.1 State of the Art and Practice

The random testing approach, as its name suggests, involves the random selection of test input data for input parameter values, which was shown remarkably effective and efficient for testing and bug finding [43]. It is widely used in industry, and has been proved to be more effective than systematic testing approaches [121, 120, 236]. Concurrently, another implementation of performance testing involves selecting a small subset of "`good`" test cases with which different testing objectives can be achieved [162]. Specifically, more performance bottlenecks can be found in a shorter period of time. Good test cases are more likely to expose bugs and to produce results that yield additional insight into the behavior of the application under test (i.e., they are more informative and more useful for troubleshooting). Constructing good test cases requires significant insight into an AUT and its features and useful rules for selecting test input data.

Performance testing of enterprise applications is manual, laborious, costly, and not particularly effective. Several approaches were proposed to improve the efficiency of performance testing [174, 37, 152]. For example, operational profile models the occurrence probabilities of functions and the distributions of parameter values, which has been introduced to test most frequently used operations [174]. Rule-based techniques are effective for discovering performance bottlenecks by identifying the problematic patterns from the source code, such as misunderstandings in API calls or problematic call sequences [152]. However, these techniques always work for some specific types of performance bottlenecks, not widely used in industry. In practice, a prevalent method for performance testing is *intuitive testing*, which is a method for testers to exercise the AUT based on their intuition and experience, surmising probable errors [67]. Intuitive testing was first introduced

in 1970s as an approach to use the experience of test engineers to focus on error-prone and relevant system functions without writing time-consuming test specifications. Thus it lowers pre-investment and procedural overhead costs [67]. When running many different test cases and observing application's behavior, testers intuitively sense that there are certain properties of test cases that are likely to reveal performance bottlenecks. However, one of the major risk of intuitive testing is losing key people (i.e., key testers). The knowledge and experience of test engineers are gone when they leave the company. Training new testers is time-consuming and expensive. Thus, it is necessary to distill the properties of test cases that reveal performance bottlenecks automatically to avoid losing money and time. *Distilling these properties automatically into rules that describe how these properties affect performance of the application is a subgoal of our approach.*

In psychology, intuition means a faculty that enables people to acquire knowledge by linking relevant but spatially and temporally distributed facts and by recognizing and discarding irrelevant facts [280]. What makes intuitive acquisition of knowledge difficult is how relevancy of facts is perceived. In software testing, facts describe properties of systems under test, and many properties may be partially relevant to an observed phenomenon. Intuition helps testers to (i) form abstractions by correctly assigning relevancy rankings to different facts, (ii) form hypotheses based on these abstractions, and (iii) test these hypotheses without going through a formal process. With FOREPOST, we partially automate the intuitive process of obtaining performance rules.

2.1.2 A Motivating Example

Consider a renter insurance program, *Renters*, designed and built by a major insurance company. A goal of this program is to compute quotes for insurance premiums for rental condominiums. Renters is written in Java and it contains close to 8,500 methods that are invoked more than three million times over the course of a single end-to-end pass through the application. Its database contains approximately 78 million customer profiles,

which are used as test input data for Renters. Inputs that cause heavy computations are sparse, and random test selection often does not perform a good job of systematically locating these inputs. A fundamental question of performance testing is how to select a manageable subset of the input data for performance test cases with which performance bottlenecks can be found faster and automatically.

Consider an example of how intuitive testing works for Renters. An experienced tester notices at some point that it takes more CPU and hardware resources (fact 1) to compute quotes for residents of the states California and Texas (fact 2). Independently, the database administrator casually mentions to the tester that a bigger number of transactions are executed by the database when this tester runs test cases in the afternoon (fact 3). Trying to find an answer to explain this phenomenon, the tester makes a mental note that test cases with northeastern states are usually completed by noon and new test cases with southwestern states are executed afterwards (fact 4). A few days later the tester sees a bonfire (fact 5) and remembers that someone's property was destroyed in wildfires in Oklahoma (fact 6). All of a sudden the tester experiences an epiphany – it takes more resources for Renters to execute tests for the states California and Texas because these states have the high probability of having wildfires. When test cases are run for wildfire states, more data is retrieved from the database and more computations are performed. The tester then identifies other wildfire states (e.g., Oklahoma) and creates test cases for these states, thereby concentrating on more challenging tests for Renters rather than blindly forcing all tests, which is unfortunately a common practice now [219]. Moreover, even if the tester detects that the test cases for the states take more execution resources, it is also important to pinpoint the reason why test cases that consume more resources. When the tester looks into the execution information of these test cases, he finds that these test cases always execute some specific methods that take an unexpectedly long time to execute. Thereby, the tester identifies the performance bottlenecks and tries to optimize these methods to save time for testing. Furthermore, it is also helpful to detect performance bottlenecks if testers find that test cases for some states perform

against their intuition. For example, some northern states like Minesota never have wildfires, so its insurance quotes relevant with wildfires should be nearing zero. However, some intensive checking for wildfire area for these states may still be performed. Hence, it is possible that these test cases invoke some unnecessary methods consuming more resources than necessary. The testers can look into the corresponding execution traces to pinpoint the potential performance bottlenecks.

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

Rules for selecting test input data that quickly lead to finding performance bottlenecks are notoriously difficult to capture. Since these rules are buried in the source code, they are hard to locate manually. Test engineers must intimately know the functionality of the subject application under test, understand how programmers designed and implemented the application, and hypothesize on how the application behavior matches the requirements. Without having useful rules that summarize these requirements, it is difficult to define objectives that lead to selecting good test cases [162]. Moreover, the performance bottlenecks are also difficult to locate manually, since the tester needs to understand exactly how the AUT executes with the selected input data and analyze each methods to

pinpoint the ones which take more resources.

Currently, the state-of-the-art for finding useful rules is to use the experience and the intuition of the performance test engineers who spent time observing the behavior of AUTs when running manually constructed test cases. There is little automated support for discovering problems with performance testing. A recent work by Jiang et al. is the first that can automatically detect performance bottlenecks in the load testing results by analyzing performance logs [150]. However, the test inputs that cause performance bottlenecks are not located. Experience and intuition are the main tools that performance test engineers use to surmise probable errors [221, 67]. Our goal is to automate the discovery of rules and abstractions that can help stakeholders pinpoint performance bottlenecks and to reduce the dependency on experience and intuition of test engineers.

2.2 The FOREPOST Approach

In this section we give an overview of FOREPOST and FOREPOST$_{RAND}$, explain the detailed algorithm, and describe the architecture and workflow finally.

2.2.1 An Overview of FOREPOST

There are two key ideas behind FOREPOST: 1) extracting rules from execution traces that describe relations between input data and the corresponding workloads of performance tests and 2) identifying bottleneck methods using these rules. Besides learned rules, FOREPOST$_{RAND}$ involves random input data to execute more execution paths.

2.2.1.1 Obtaining Rules

As part of the first key idea, the instrumented AUT is initially running using a small number of randomly selected test input data. Its execution profiles are collected and automatically clustered into different groups that collectively describe different performance results of

the AUT. For example, there can be two groups that are corresponding to good and bad performance test cases, respectively.

The set of values for the AUT inputs for good and bad test cases represent the input to a Machine Learning (ML) classification algorithm. In FOREPOST, we choose the rule learning algorithm, called Repeated Incremental Pruning to Produce Error Reduction (RIPPER) [65], to obtain the rules that guide the selection of test input data in test scripts. RIPPER is a rule learning algorithm, which is modified from the Incremental Reduced Error Pruning (IREP) [102]. It integrates pre-pruning and post-pruning into a learning phrase and follows a separate-and-conquer strategy. Each rule will be pruned right after it is generated, which is similar to the IREP. But the difference is that it chooses an alternative rule-value metric in the pruning phrase, provides a new stopping condition and optimizes the initial rule set, which is obtained by IREP.

This input of ML algorithm is described as implications of the form $V_{I_1}, \ldots, V_{I_k} \rightarrow T$, where V_{I_m} is the value of the input I_m and $T \in \{G, B\}$, with G and B representing good and bad test cases correspondingly. In fact, T is the summarized score of an execution trace that describes summarily whether this execution has evidence of performance bottlenecks. The ML classification algorithm learns the model and outputs rules that have the form $I_1 \odot V_{I_1} \bullet I_2 \odot V_{I_2} \bullet \ldots \bullet I_k \odot V_{I_k} \rightarrow T$, where $\odot$ is one of the relational operators (e.g., $>$ and $=$) and $\bullet$ is one of the logical connectors (i.e., $\wedge$ and $\vee$). These rules are instrumental in guiding the selection of the test input data in test scripts. For example, in a web application, if I_1 refers to a URL request “viewing the page of cat”, and I_2 refers to a URL request “viewing the page of dog”, the rule $(I_1 = V_{I_1}) \wedge (I_2 > V_{I_2}) \rightarrow G$ means that “viewing the page of cat” V_{I_1} times and “viewing the page of dog” more than V_{I_2} times is good to trigger performance bottlenecks. The next test cases should be generated based on this rule. More detailed rules are provided in Table 2.3.

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

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

2.2.1.2 Identifying Bottlenecks

Our goal is to automatically identify bottlenecks as method calls whose execution seriously affects the performance of the whole AUT. For example, consider a method that is periodically executed by a thread which checks if the content of some file is modified. While this method may be one of the bottlenecks, it is invoked in both good and bad test cases. Thus, its contribution to the resource consumption as the necessary part of the application logic does not lead to any insight that may resolve a performance problem. Our second key idea is to consider the most significant methods that occur in good test cases and that are not invoked, or have little to no significance, in bad test cases, where the significance of a method is a function of the resource consumption that its execution triggers. We measure resource consumption as a normalized weighted sum of (i) the number of times that this method is invoked, (ii) the total elapsed time of its invocations minus the elapsed time of all methods that are invoked from this method, and finally, (iii) the number of methods whose invocations are spawned from this method. In FOREPOST, Independent Component Analysis (ICA) is used to identify the performance bottlenecks. The detailed algorithm will be explained in sections 2.2.2 and 2.2.3.

2.2.2 Blind Source Separation

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

We draw an analogy between separating method invocations in execution traces into components that represent high-level features and a well-known problem of separating signals that represent different sources from a signal that is a mixture of these separate signals. This problem is known as *blind source separation (BSS)* [238, pages 13-18].

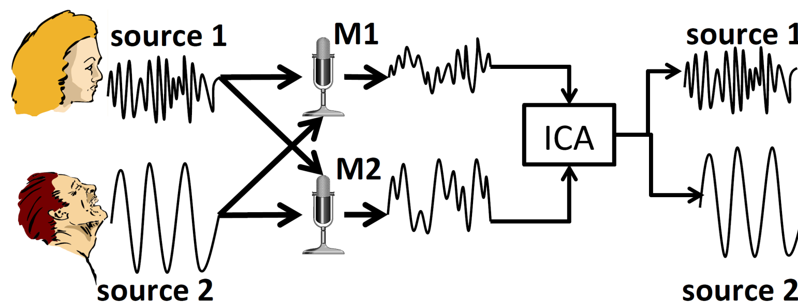


Figure 2.1: A speech model of blind source separation.

The idea of BSS is illustrated using a model where two people speak at the same time in a room with two microphones `M1` and `M2` as it is shown in Figure 2.1. Their speech sig-

nals are designated as `source 1` and `source 2`. Each microphone captures the mixture of the signals `source 1` and `source 2`, which are the corresponding signal mixtures from `M1` and `M2` respectively. The original signals `source 1` and `source 2` are separated from the mixtures using a technique called *independent component analysis (ICA)* [136, 112]. ICA is based on the assumption that different signals from different physical processes are statistically independent. For example, different features are often considered independent since they are implemented in applications as separate concerns [237, 264]. When physical processes are realized (e.g., different people speak at the same time, or stocks are traded, or an application is run and its implementations of different features are executed in methods), these different signals are mixed, and these signal mixtures are recorded by some sensors. Using ICA, independent signals can be extracted from these mixtures with a high degree of precision.

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

2.2.3 Independent Component Analysis

A schematics of the ICA matrix decomposition is shown in Figure 2.2. The equation $\mathbf{x} = \mathbf{A} \cdot \mathbf{s}$ described the process, where **x** is the matrix that contains the observed signal mixtures and **A** is the transformation or mixing matrix that is applied to the signal matrix **s**. In our case, the matrix **x** is shown in Figure 2.2 on the left hand side of the equal sign, and its rows correspond to application execution traces from different input data, and its columns correspond to method invocations that are observed for each trace.

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

Using ICA, the matrix **x** is decomposed into a transformation and a signal matrices that are shown on the right hand side of the equal sign in Figure 2.2. The input to ICA is the matrix **x** and the number of source signals, which in our case is the number of features

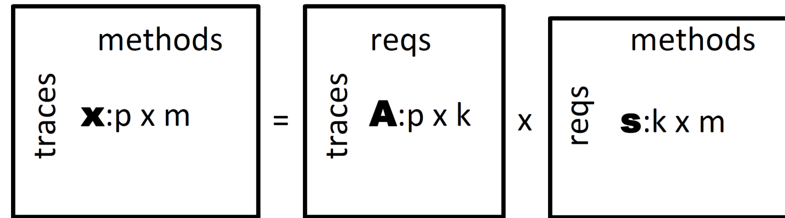


Figure 2.2: Schematics of the ICA matrix decomposition.

(features in the Figure 2.2) implemented in the application. The elements of the matrix **A**, A_p^q, specify the weights that each profile p contributes to executing code that implements the feature q, and the elements of the matrix **s**, s_q^k, specify the weights that each method k contributes to executing code that implements the feature q. Methods that have the highest weights for the given features are considered the most significant and interesting for troubleshooting performance bottlenecks. This is a hypothesis that we evaluate in Sections 4.3 and 4.4.

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

The architecture of FOREPOST and FOREPOST$_{RAND}$ is shown in Figure 2.3. Solid arrows show command and data flows between components, and numbers in circles indicate the sequence of operations in the workflow. The beginning of the workflow is shown with the fat arrow that indicates that the *Test Script* executes the application by simulating users and invoking methods of the AUT interfaces. The *Test Script* is written (1) by the test engineer as part of automating application testing.

Once the test script starts executing the application, its execution traces are collected (2) by the *Profiler*, and these traces are forwarded to the *Execution Trace Analyzer*, which produces (3) the *Trace Statistics*. We implemented the *Profiler* using the TPTP framework[1]. These statistics contain information on each trace, such as the number of invoked methods, the elapsed time it takes to complete the end-to-end application run, the

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

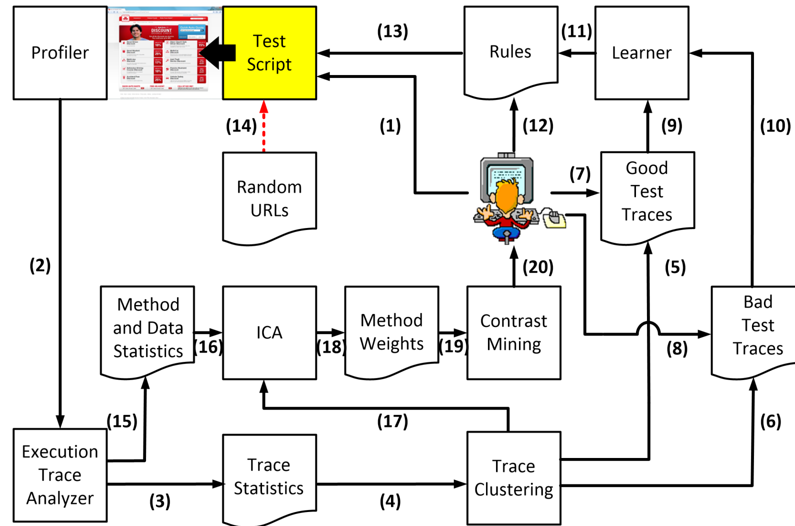


Figure 2.3: The architecture and workflow of FOREPOST and FOREPOST$_{RAND}$. FOREPOST does not contain the step 14.

number of threads, and the number of unique methods that were invoked in this trace. The trace statistics are supplied (4) to the module *Trace Clustering*, which uses the average execution time to perform unsupervised clustering of these traces into two groups that correspond to (5) *Good* and (6) *Bad test traces*. The user can review the results of clustering and (7,8) reassign the clustered traces if needed. These clustered traces are supplied (9,10) to the *Learner*, which uses a ML algorithm, RIPPER [65], to learn the classification model and (11) output rules that were described in Section 3.2.1. The user can review (12) these rules and mark some of them as erroneous if the user has sufficient evidence to do so. Next, the rules are supplied (13) to the *Test Script*. In FOREPOST, once the *Test Script* receives a new set of rules, it partitions the input space into blocks according to these rules and starts forming test inputs by selecting one input from each block. In FOREPOST$_{RAND}$, the *Test Script* is a combination of random input data and several blocks of input space that correspond to different rules. The major difference in the architecture of FOREPOST$_{RAND}$ is that it considers random URLs as input data that is shown in step (14) does not contain this step (14). We expect that adding the random input data could enlarge the test coverage to find more potential performance bottlenecks.

After generating new input data, the *Profiler* collects execution traces of these new test runs. The cycle repeats with new rules that are learned after several passes, and the input space is repartitioned adaptively to accommodate these rules. We implemented the ML part of FOREPOST using JRip[2], which is implemented by Weka [284].

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

Finally, once the input space is partitioned into clusters that lead to good and bad test cases, we want to find methods that are specific to good performance test cases and that are most likely to contribute to bottlenecks. This task is accomplished in parallel to computing rules, and it starts when the *Execution Trace Analyzer* produces `(15)` the method and data statistics of each trace, and then uses this information to construct `(16)` two matrices $\mathbf{x_B}$ and $\mathbf{x_G}$ for bad and good test cases correspondingly, based on the information provided by `(17)`. Constructing these matrices is done as described in Section 2.2.3. Once these matrices are constructed, ICA decomposes them `(18)` into the matrices $\mathbf{s_B}$ and $\mathbf{s_G}$ corresponding to bad and good tests. Recall that our key idea is to consider the most significant methods that occur in good test cases and that are not invoked, or have little to no significance in bad test cases. Cross-referencing the matrices $\mathbf{s_B}$ and $\mathbf{s_G}$, which specify the method weights for different features, the *Contrast Mining* `(19)` compares the

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

method weights in both of good and bad test cases, and determines the top methods that the performance testers should look at (20) to identify and debug possible performance bottlenecks. This step completes the workflow of FOREPOST. The detailed algorithm for identifying bottlenecks, including ICA and *Contrast Mining*, is shown in the section 2.2.5.

2.2.5 The Algorithm for Identifying Bottlenecks

In this section we describe our algorithm for identifying bottlenecks using FOREPOST and FOREPOST$_{RAND}$. This algorithm clusters the traces based on the trace statistics, and then generates the matrices for both good and bad test cases. By using the ICA algorithm, it calculates the weight for each method in both good and bad test cases. If one method is significant in good test cases but not significant in bad test cases, then we conjecture that it is likely to be a bottleneck. This algorithm provides a ranked lists of bottlenecks as its final output. The algorithm `FOREPOST` is shown in Algorithm 1. `FOREPOST` takes as its input the set of captured execution traces, T, and the signal threshold, U, which is used to select methods whose signals indicate their significant contribution in execution traces. The set of methods that are potential bottlenecks, B, is computed and returned in line 15 of the algorithm.

In step 2 the algorithm initializes to the empty set the set of bottlenecks and the set of clusters that contain execution traces that are matched to good and bad test cases. In step 3 the procedure `ClusterTraces` is called that automatically clusters execution traces from the set T into good (C_{good}) and bad (C_{bad}) test case clusters. Next, in steps 4 and 5 the procedure `CreateSignalMixtureMatrix` is called on clusters of traces that correspond to bad and good test cases respectively to construct two matrices $\mathbf{x_b}$ and $\mathbf{x_g}$ corresponding to bad and good test cases, as described in Section 2.2.4. In step 6 and 7, the procedure `ICA` decomposes these matrices into the matrices $\mathbf{s_b}$ and $\mathbf{s_g}$ corresponding to bad and good test cases, as described in step (15) in Section 2.2.4.

Next, the algorithm implements the *Contrast Mining* component in steps 8–15, mining

all the methods for all the feature components in the decomposed matrices. More specifically, for each method whose signal exists in the transformation and signal matrices that correspond to good cases, we compare if this method does not occur in the counterpart matrices for bad test case decompositions. Alternatively, if the same method from the same component occurs, then the distance between these two signals in the good and bad test should be quite large. The distance is calculated as shown in equation 2.1, where $M_g^i = M_b^k \wedge R_g^j = R_b^l$, M = method, g = good, b = bad, R = component, which means the same method from the same component occurs.

$$D_{e_g} = \sum_{i=0}^{N_{M_g}} \sum_{j=0}^{N_{R_g}} \sqrt{\left(SL_g^{ij} - SL_b^{kl}\right)^2} \tag{2.1}$$

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

2.3 Evaluation

In this section, we state our research questions (RQs) and we describe how we evaluated FOREPOST on three applications: the commercial application, Renters, that we described as our motivating example in Section 2.1.2 and two open-source applications, JPetStore and Dell DVD Store, which are frequently used as industry benchmarks.

2.3.1 Research Questions

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

Algorithm 1: The algorithm for identifying bottlenecks.

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

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

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

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

The rationale for these RQs lies in the complexity of the process of detecting performance bottlenecks. Not all methods are bottlenecks whose execution times are large. For example, the method *main* can be described as a bottleneck, since it takes naturally the most time to execute. However, it is unlikely that a solution may exist to reduce its execution time significantly. Thus, the effectiveness of bottleneck detection involves not

only the precision with which performance bottlenecks are identified, but also in how fast they can be found and how different parameters affect this process.

In order to address these RQs, we conducted three empirical studies. In this section, we first describe the subject applications used in the studies, then we cover the methodology and variables for each empirical study. The results are presented in Section 4.4.

Table 2.1: Characteristics of the insurance application Renters.

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

2.3.2 Subject AUTs and Experimental Hardware

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

The Renters is a J2EE application that calculates the insurance premiums for rental condominium. Its software metrics are shown in Table 2.1, where Size = lines of code (LOC), NOC = number of classes, NOM = number of methods, NOA = number of attributes, MCC = Average McCabe cyclomatic Complexity, NOP = number of packages. The backend database is DB2 running on the IBM Mainframe, its schema contains over 700 tables including close to 15,000 attributes that contain data on over 78 million customers, which are used as the input to FOREPOST. The application accepts input values

using 89 GUI objects. The total number of combinations of input data is approximately 10^{65}, making it infeasible to comprehensively test Renters. We used Renters in our motivating example in Section 2.1.2.

JPetStore is a Java implementation of the PetStore benchmark, where users can browse and purchase pets, and rate their purchases. This sample application is typical in using the capabilities of the underlying component infrastructures that enable robust, scalable, portable, and maintainable e-business commercial applications. It comes with full source code and documentation, therefore, we used it in the evaluation of FOREPOST and demonstrated that we can build scalable security mechanisms into enterprise solutions. We used iBatis JPetStore 4.0.5[3]. JPetStore has 2,139 lines of code, 386 methods, 36 classes in 8 packages, with the average cyclomatic complexity of 1.224; it is deployed using the web server Tomcat 6.0.35 and it uses Derby as its backend database.

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

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

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

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

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

2.3.3 Research Question 1

Our goal is to determine which approach is better for finding good performance test cases faster. Given the complexity of the subject applications, it is not clear with what input data the performance can be worsened significantly for these applications. In addition, given the large space of the input data, it is not feasible to run these applications on all the inputs to obtain the worst performing execution profiles. These limitations dictate the methodology of our experimental design, specifically for choosing the competitive approaches to FOREPOST. We selected the random testing as the main competitive approach to FOREPOST, since it is widely used in industry and it has been proved to consistently outperform different systematic testing approaches [236, 121]. To support our claims in this paper, our goal is to show, with strong statistical significance, under what conditions FOREPOST outperforms random testing.

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

To obtain URL requests that exercise different components of JPetStore, we used the spider tool in JMeter to traverse the web interface of JPetStore, and recorded the URLs that were sent to the backend during this process. In random testing, multiple URLs were randomly selected to form a transaction. In FOREPOST, the URL selection process was guided by the learned rules. We limited the number of URLs in each transaction to 100. This number was chosen experimentally based on our observations of JPetStore users

who explored approximately 100 URLs before switching to other activities. Increasing the number of certain URL requests in transactions at expense of not including other URL requests may lead to increased workloads, and the goal of our experimental evaluation is to show that FOREPOST eventually selects test input data (i.e., customer profiles for Renters or combinations of URLs for JPetStore and Dell DVD Store) that lead to increased workloads when compared to the competitive approaches.

When testing JPetStore and Dell DVD Store, URLs in a transaction are issued to the backend consecutively to simulate a single user. Multiple transactions are randomly selected and issued in parallel to simulate concurrent users using the system. During the testing we used different numbers of concurrent transactions, and measured the average time required by AUT backend to execute a transaction. A goal of this performance testing was to find combinations of different URLs in transactions for different concurrent users that lead to significant increase in average time per transaction, which is often correlated with the presence of performance bottlenecks. Since our experiments involved random selection of input data, it was necessary to conduct the experiments multiple times and pick the average to avoid skewed results. We ran each experiment 5 times on each subject to consider the collected data as a good representative sample

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

2.3.4 Research Question 2

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

We added two different groups of delays into JPetStore as bottlenecks. The first group contains bottlenecks with exactly the same delay, whereas the second group contains bottlenecks with different length of delays. The bottlenecks#1 contain methods with the same delay of 0.1s in each bottleneck, and the bottlenecks#2 contain methods with different delays (e.g., 0.05s, 0.1s and 0.15s). The artificial bottlenecks were injected into nine methods from the 386 probed methods. On the other hand, we injected one group of bottlenecks with the same delay into Dell DVD Store. Since Dell DVD Store only contains 32 native methods, we decided to inject the artificial bottlenecks into both the Dell DVD Store source code and the standard libraries (two in source code and three in library). To make sure that the injected bottlenecks are going to be representative, before injecting these bottlenecks, we ran FOREPOST on JPetStore to find the original bottlenecks (i.e., methods from the original code that were ranked on the top when no artificial bottlenecks were injected), as well as the original positions of the injected bottlenecks. The injected bottlenecks are ranked on low positions, that implies that our injected bottlenecks are not really original bottlenecks and we choose them randomly. After injecting bottlenecks, some of the artificial bottlenecks are ranked in the top ten results, but the original bottlenecks are ranked on lower positions , implying that the lengths of delays which we injected

are significant enough to be detected.

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

The values of four independent variables are shown in Table 2.2, where Profiles n_p = number of profiles for learning rules, iterations n_i = times of learning rules, bottlenecks n_b = number of artificial bottlenecks, users n_u = number of users. The first independent variable is the *number of profiles* (i.e., n_p) that needs to be collected in order to enable learning rules. Intuitively, the number of collected profiles can affect the resulting rules. For example, the rules extracted from only ten profiles should contain different information from execution traces or profiles, as compared to the configuration containing 15 profiles. In our sensitivity analysis, the numbers of profiles are set to 10, 15, and 20. Our goal is to empirically investigate whether the number of profiles has substantial impact on the accuracy of FOREPOST.

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

increases. Our goal is to analyze the performance of FOREPOST after different numbers of iterations.

The third independent variable is the *number of bottlenecks* (i.e., n_b), which is the number of artificial performance bottlenecks injected into subject applications. Artificial delays were injected randomly into methods for simulating the realistic performance bottlenecks. The numbers of bottlenecks are set to 6, 9, and 12, and all bottlenecks have the same delay. Our goal is to empirically investigate the performance of FOREPOST on detecting different numbers of performance bottlenecks. All the artificial bottlenecks are shown in Appendix of our paper [196].

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

Table 2.2: Independent variables in sensitivity analysis.

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

2.3.5 Research Question 3

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

in both of FOREPOST and FOREPOST$_{RAND}$ to compare these two approaches side by side.

Table 2.3: Selected rules that are learned for **R**enters and **J**PetStore.

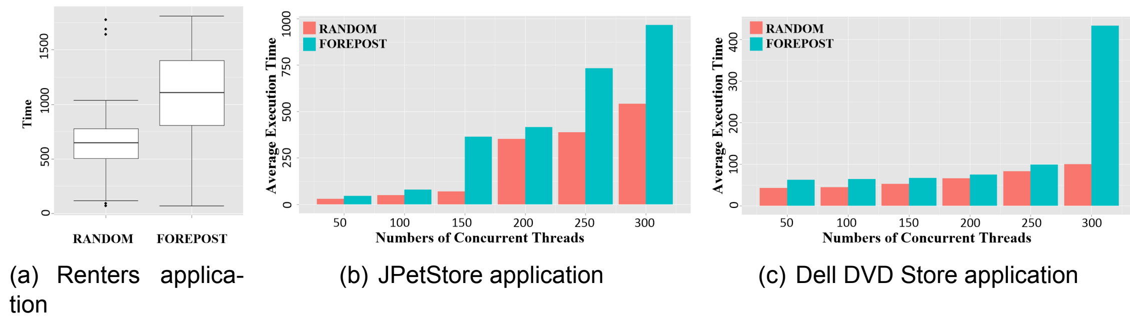
Rule	Antecedent	Cons
R–1	$(\texttt{customer.numberOfResidents} \leq 2) \wedge$ $(\texttt{coverages.limitPerOccurrence} \geq 400000) \wedge$ $(\texttt{preEligibility.numberOfWildAnimals} \leq 1)$	Good
R–2	$(\texttt{adjustments.homeAutoDiscount} = 2) \wedge$ $(\texttt{adjustments.fireOrSmokeAlarm} = \texttt{LOCAL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

2.4 Results

In this section, we describe and analyze the results obtained from our experiments with Renters, JPetStore and Dell DVD Store. We provide only parts of the results in this paper. The complete results of all our experiments are shown in my online appendix [11].

2.4.1 Research Question 1

Finding Test Inputs for Increased Workloads. The results for Renters are shown in the box-and-whisker plots in Figure 2.4 (a) that summarize the execution times for end-to-end single application runs with different test input data, where the time for end-to-end runs is measured in seconds. The central box represents the values from the lower to upper



(a) Renters application (b) JPetStore application (c) Dell DVD Store application

Figure 2.4: The summary of the results for Empirical Study 1.

quartile (i.e., 25 to 75 percentile). The middle line represents the median. The thicker vertical line extends from the minimum to the maximum value. We first calculated the effect size to compare the FOREPOST with RANDOM using Cohen's d [64]. The result is 1.2. According to Cohen's definition, the value of the effect size is large ($\geq$ 0.8) implying that there is a difference between the execution times for RANDOM and FOREPOST. To further test the NULL hypotheses that there is no significant difference between the execution time for the random and FOREPOST approaches, we performed statistical tests for two paired sample means. Before applying paired significance test, we first applied the Shapiro-Wilk Normality Test [257] to check the normality distribution assumption. The results show that the sample data does not follow normal distribution even at the 0.01 significance level. Therefore, we chose to use the Wilcoxon Signed-Rank Test [282] to compare the two sample sets, because it is suitable for the case that the sample data may not be normally distributed [191]. The results of the statistical test allow us to reject the NULL hypotheses and accept the alternative hypotheses with strong statistical significance ($p < 0.0001$), which states that **FOREPOST is more effective at finding test input data that steers applications towards more computationally intensive executions than random testing**, thus addressing RQ$_1$.

This conclusion is confirmed by the results for JPetStore and Dell DVD Store that are shown in Figure 2.4 (b) and (c), where the bars represent average times per transaction in seconds for the Random and FOREPOST approaches for different numbers of concurrent

transactions ranging from 50 to 300. In JPetStore, while in performing random testing, it takes on average 542.1 seconds to execute 300 transactions. With FOREPOST, executing 300 transactions takes on average 965.8 seconds, which shows 78.2% increase. In Dell DVD Store, while performing random testing, it takes on average 100.0 seconds to execute 300 transactions. With FOREPOST, executing 300 transactions takes on average 433.3 seconds, which shows 333.3% increase. This implies that FOREPOST outperforms random testing by more than one order of magnitude. Random testing is evaluated on the instrumented JPetStore and Dell DVD Store, so that the cost of instrumentation is evenly factored into the experimental results. FOREPOST has a large overhead, close to 80% of the baseline execution time, however, once rules are learned and stabilized, they can be used to partition the input space without using instrumentation.

Identifying Bottlenecks and Learned Rules. When applying the algorithm for identifying bottlenecks (see Section 2.2.5) on Renters, we obtained a list of top 30 methods that the algorithm identified as potential performance bottlenecks out of approximately 8,500 methods. To evaluate how effective this algorithm is, we asked the insurance company to allocate the most experienced developer and tester for Renters to review this list and provide feedback on it. According to the management of the insurance company, it was the first time when a developer and a tester were in the same room together to review results of testing.

The reviewing process started with the top bottleneck method, `checkWildFire- Area`. The developer immediately said that FOREPOST did not work since this method could not be a bottleneck for a simple reason – this method computes insurance quotes only for U.S. states that have wildfires, and FOREPOST selected test input data for northern states like Minnesota that never have wildfires. We explained that FOREPOST automatically selected the method `checkWildFireArea` as important because its weight was significant in execution traces for good test cases, and it was absent in traces for bad test cases. It meant that this method was invoked many times for the state of Minnesota and other northern states, even though its contribution in computing insurance quotes was zero

for these states. Invoking this method consumes more resources and time in addition to significantly increasing the number of interactions with the backend databases. After hearing our arguments, the developer and the tester told us that they would review the architecture documents and the source code and get back to us.

A day later they got back with a message that this and few other methods that FOREPOST identified as bottlenecks were true bottlenecks. It turned out that the implementation of the `Visitor` pattern in Renters had a bug, which resulted in incorrect invocations of the method `checkWildFireArea`. Even though it did not contribute anything to computing the insurance quote, it consumed significant resources. After implementing a fix based on the feedback from FOREPOST, the performance of Renters increased by approximately seven percent, thus addressing RQ_1 that **FOREPOST is effective at identifying bottlenecks**. More experiments of identifying bottlenecks in FOREPOST presented in the section 2.4.2 and 2.4.3 also support this conclusion.

Examples of learned rules are shown in Table 2.3, where the first letters of the names of the AUTs are used in the names of rules to designate to which AUTs these rules belong. The last column (Cons) designates the consequent of the rule that corresponds to good and bad test cases that these rules describe. When professionals from the insurance company looked at these and other rules in more depth, they identified certain patterns that indicated that these rules were logical and matches some features. For example, the rules `R-1` and `R-3` point out to strange and inconsistent insurance quote inputs, where low deductible goes together with very high coverage limit, and it is combined with the owner of the condo taking prescribed medications, and with the condo having fewer than two residents. All these inputs point to situations that are considered higher risk insurance policies. These classes of input values trigger more computations that lead to significantly higher workloads.

For JPetSore, rules `J-1` and `J-2` describe inputs as the number of occurrences of URLs in transactions, where URLs are shown using descriptive names (e.g., “Checkout” for the URL that enables customers to check out their shopping carts). It is important that

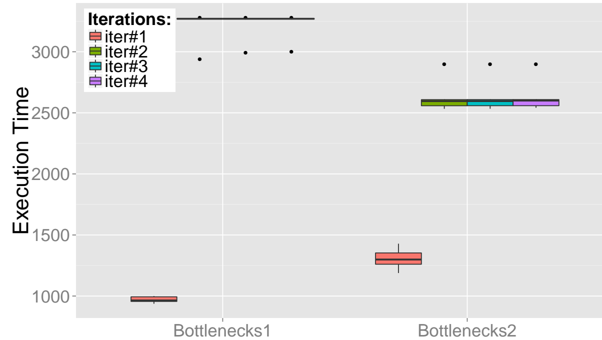


Figure 2.5: Average execution times (in second) for different groups of injected bottlenecks (i.e.,bottlenecks#1 and bottlenecks#2), where n_p = 10 and n_u = 5.

rules for both applications are input-specific. While we do not expect that rules learned for one system would apply to a completely different system, training a new set of rules using the same algorithm should deliver similar benefits.

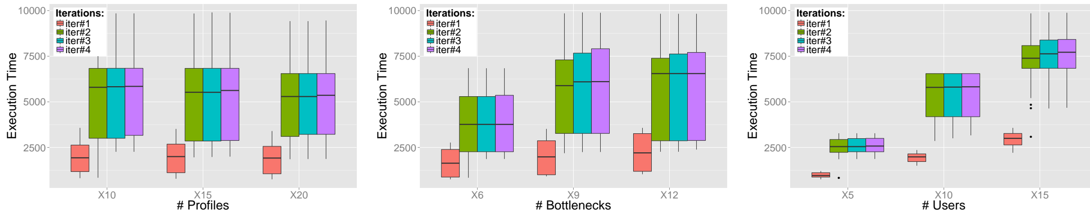
2.4.2 Research Question 2

Can FOREPOST identify injected bottlenecks? Recall that we injected two groups of artificial bottlenecks to investigate how FOREPSOT identifies different injected bottlenecks. The bottlenecks#1 refer to the methods that have same artificial delay, and the bottlenecks#2 refer to the methods that have different artificial delays. The results are shown in Fig. 2.5, where different colors represent different iterations. The central box represents the values from the lower to upper quartile (i.e., 25 to 75 percentile). The middle line represents the median. As the results show, the execution times for bottlenecks#1 are generally larger than the execution times for bottlenecks#2. The reason is that different delays in these two groups of bottlenecks lead to different AUT behaviors. Some injected bottlenecks in bottlenecks#1 have longer delays, leading to more computationally intensive executions as compared to the bottlenecks#2.

We further check the effectiveness of FOREPOST on identifying these two groups of bottlencks, and the results are shown in in Fig. 2.7. Due to the space limitation, we

only show detailed results when the number of iterations is equal to four. More results can be found in our paper [196]. The precision is measured as the percentage of the artificial bottlenecks returned in the top methods. The recall is measured as the ratio of the artificial bottlenecks within the top methods to all the injected bottlenecks. The F-score is calculated based on the precision and recall (i.e., $F - score = \frac{2 * precision * recall}{precision + recall}$). We set different cut points (i.e., 2 - 20) to calculate those metrics. For example, if we set the cut point as ten, it means we only consider the ranks of methods which are in top ten and the methods outside the top ten are ignored. The results show that FOREPOST can find more bottlenecks in bottlenecks#1, implying that FOREPOST is able to find more serious bottlenecks (length of delay in bottlenecks#1 is longer than bottlenecks#2). Moreover, we observe that the results of bottlenecks#2 vary greatly as compared to the results of bottlenecks#1. One possible reason is that the injected bottlenecks with different delays (i.e., bottlenecks#2) make the AUT performance vary, thus FOREPOST may converge to the different executions for uncovering different performance bottlenecks, leading to unstable results. On the contrary, FOREPOST always converge to some stable states when injecting the same performance bottlenecks (i.e., bottlenecks#1).

How do different parameters of FOREPOST affect its performance? To investigate the impacts of various independent variables on the execution time, we control the value of each independent variable, and present the corresponding results in Fig. 2.6, where Different colors represent different iterations. The central box represents the values from the lower to upper quartile (i.e., 25 to 75 percentile). The middle line represents the median. In this figure, each sub-figure represents the execution time information when we control the value of each independent variable (e.g., number of profiles, bottlenecks, and users). In each sub-figure, the x-axis presents the values for the controlled independent variable, the y-axis presents the execution time, and boxplots in different color represent different iteration. Note that the boxplots present the median (line in the box), upper/lower quartile, and 90th/10th percentile values. From the figure, we can infer the following observations:



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

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

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

Figure 2.6: Average execution times (in second) when controlling different independent variables.

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

When controlling the independent variable of the *number of profiles* (see Fig. 2.6 (a)), we find that the execution time does not change much across different number of profiles. We also observe an interesting finding that when the number of profiles per iteration increases, the execution time for various other settings tends to be more stable. For example, the boxplot for using 20 profiles is more stable than that for ten profiles. The reason is that after collecting more profiles, the machine learning results can be more accurate in guiding meaningful rule generation. Therefore, the empirical results when controlling the number of profiles demonstrate that FOREPOST becomes more stable when using more profiles.

When controlling the independent variable of the *number of bottlenecks* (see Fig. 2.6 (b)), we find that the execution time increases gradually and the performance has wider range when injecting more bottlenecks. One possible reason is that more injected bottlenecks may incur more bottlenecks to be actually executed, leading to longer execution time. Moreover, the interaction of different bottlenecks can make the AUT's performance rather unstable, which enlarges the possible range for the application execution time.

When controlling the independent variable of the *number of users* (see Fig. 2.6 (c)), we find that the execution time increases linearly when simulating more users. For example, when using five users, the average execution time for the inputs selected by FOREPOST (e.g., the 4th iteration) is around 2500 seconds, while it is approximately 5500 and 7500 seconds when the number of users increases to ten and 15 respectively. Obviously, when the number of users increases, it would take more time to execute the URL requests sending by the increased users. We can also find that the execution time becomes more unstable when simulating ten users as compared to five users. The possible reason is that when increasing the number of users, the problems of multi-threading, synchronization, etc., may arise, causing the studied application to have performance bottlenecks of wider range.

We further check the impact of *number of bottlenecks* in identifying bottlenecks when n_p = 10, and n_u = 5. The precision, recall and F-score are shown in Tables 2.4, 2.5, and 2.6. More results can be found in our paper [196]. In each of the three tables, Column 1 lists the different cut points used, Columns 2-5/6-9/10-13 list the corresponding metric results for FOREPOST when injecting 6/9/12 bottlenecks. According to the three tables, we have the following observations:

First, the best cut point depends on the number of potential bottlenecks in the application under test as well as the number of iterations used in FOREPOST. For example, as shown in Table 2.6 (n_u=5 and n_p=10), when controlling the number of bottlenecks to be six, the cut point with the highest F-score value is six after 1st iteration, and four after 2nd iterations. Similarly, when controlling the number of iterations to be four, the cut point with

Table 2.4: Precision for FOREPOST when n_u=5 and n_p=10

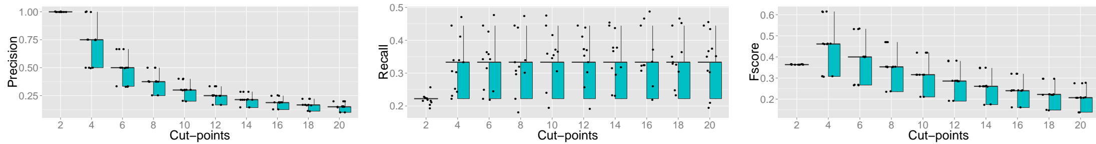
Cut points	6 Bottlenecks				9 Bottlenecks				12 Bottlenecks			
	iter #1	iter #2	iter #3	iter #4	iter #1	iter #2	iter #3	iter #4	iter #1	iter #2	iter #3	iter #4
2	100.00	100.00	90.00	90.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	90.00
4	95.00	65.00	55.00	60.00	100.00	65.00	70.00	75.00	100.00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

Table 2.5: Recall for FOREPOST when n_u=5 and n_p=10

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

Table 2.6: F-score for FOREPOST when n_u=5 and n_p=10

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



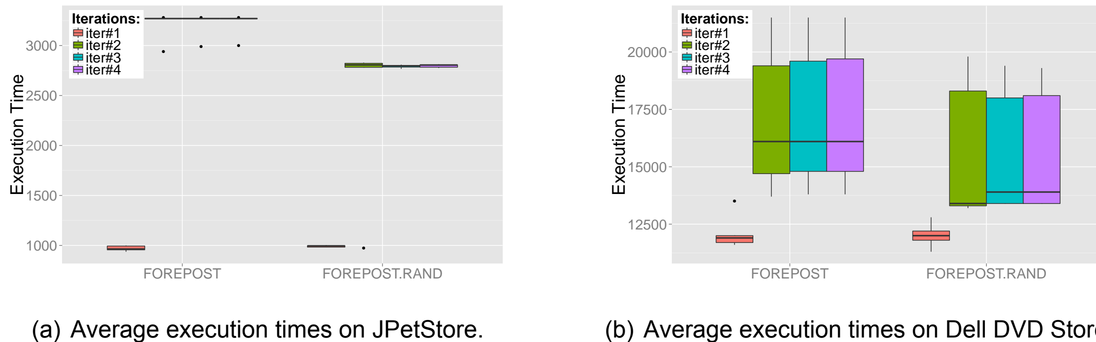
(a) Precision for 4th iteration (b) Recall for 4th iteration (c) F-score for 4th iteration

Figure 2.7: Comparison between FOREPOST using uniform or different bottlenecks for JPetStore. The red boxplots refer to bottlenecks#1. The green boxplots refer to bottlenecks#2.

the highest F-score value is four when injecting six bottlenecks, and six when injecting twelve bottlenecks.

Second, as the number of iterations increases, FOREPOST tends to miss some injected performance bottlenecks. For example, when using six as the cut point value where n_b = 9, the random inputs (i.e., the 1st iteration) have a F-score of 80.00, but the selected inputs (e.g., the 2nd iteration) only has a F-score of 40.00. A possible explanation is that FOREPOST runs a much more manageable and focused subset of input data after gen-

erating rules, which means the size of this subset is much smaller and easier to test, but it can lead to intense computations faster than other subsets. However, since the input data is limited by the rules, the domain would become smaller if rules are only associated with a small subset of methods. Although some methods have a high probability to be associated with the bottlenecks, FOREPOST still does not list them since they are not invoked in the execution traces. For example, before learning rules, there are 386 methods invoked in JPetStore. But only 208 methods are invoked after learning rules, which means the learned rules focused on a subset of input data, therefor less methods are invoked during the execution. Meanwhile, as observed from Figure 2.6, the average execution times of selected inputs are quite higher than the average execution times of random inputs (i.e., inputs in the 1st iteration), which implies that FOREPOST finds the subset of input data that lead to intense computations in a short time. So FOREPOST identifies computationally more expensive execution paths as compared to random performance testing at the expense of lower precision.



(a) Average execution times on JPetStore.

(b) Average execution times on Dell DVD Store.

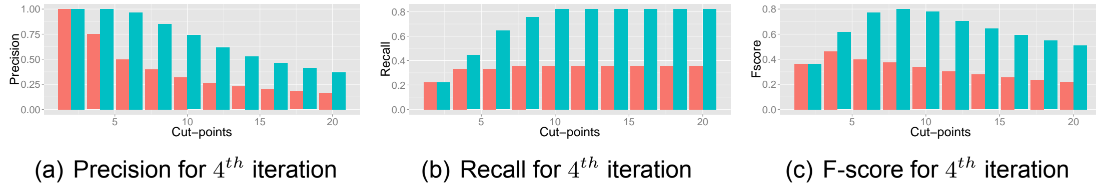
Figure 2.8: Average execution times (in second) for FOREPOST and FORPOST$_{RAND}$, where n_p = 10 and n_u = 5.

2.4.3 Research Question 3

Comparing FOREPOST and FOREPOST$_{RAND}$ in Finding Test Input for Increased Workloads. The results for comparing FOREPOST and FOREPOST$_{RAND}$ are shown in Fig. 2.8, where different colors represent different iterations. The central box represents

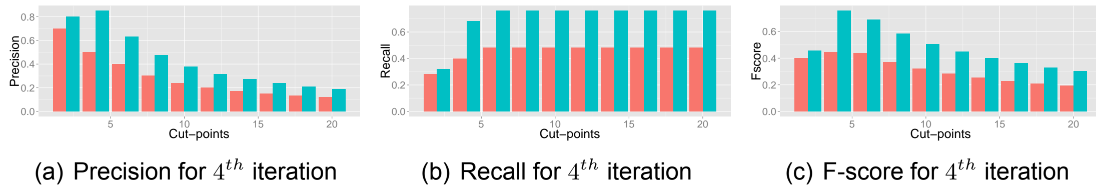
the values from the lower to upper quartile (i.e., 25 to 75 percentile). The middle line represents the median. On JPetStore, the inputs selected by FOREPOST take around 3200 seconds after the 2nd iteration, while inputs selected by FOREPOST$_{RAND}$ take around 2800 seconds. Although there is no significant difference after the 2nd iteration, the average execution times of FOREPOST are always larger than those of FOREPOST$_{RAND}$. The results demonstrate that FOREPOST is more effective in finding test input data that steer applications towards more computationally intensive executions compared with FOREPOST$_{RAND}$. The conclusion is confirmed by the results for Dell DVD Store shown in Fig. 2.8 (b). Morever, the increase between random inputs and selected inputs in execution time on Dell DVD Store is smaller as compared to JPetStore since Dell DVD Store has relatively smaller number of combinations of inputs. Thus, even randomly selected inputs can cover significant part of the computationally intensive executions. All in all, FOREPOST is more effective in finding inputs to cover computationally intensive executions, thus addressing RQ$_3$.

Comparing FOREPOST and FOREPOST$_{RAND}$ in Identifying Bottlenecks. Since the precision, recall and F-score are quite similar after the 1st iteration since inputs in both FOREPOST and FOREPOST$_{RAND}$, we only show the results of the fourth iteration in this paper (Fig. 2.9 and 2.10, where each bar represents the average precision/recall/f-score across five runs of the same setting. The red bars refer to FOREPOST. The green bars refer to FOREPOST$_{RAND}$.). More results can be found in our paper [196]. The results show that FOREPOST$_{RAND}$ have all clearly larger values as compared to FOREPOST, implying that FOREPOST$_{RAND}$ outperforms FOREPOST in terms of accuracy. As we mentioned, FOREPOST may miss to identify some bottlenecks since the input data is generated only based on rules which focus on traces that correspond to computationally intensive executions, while FOREPOST$_{RAND}$ involves random input data in addition to the specific input data based on the rules, covering other traces without losing accuracy. As our results demonstrated, software testers can choose either FOREPOST or FOREPOST$_{RAND}$ based on their goals: either identifying extreme bottlenecks by focus-



(a) Precision for 4th iteration (b) Recall for 4th iteration (c) F-score for 4th iteration

Figure 2.9: Comparison between FOREPOST and FOREPOST$_{RAND}$ for JPetStore.



(a) Precision for 4th iteration (b) Recall for 4th iteration (c) F-score for 4th iteration

Figure 2.10: Comparison between FOREPOST and FOREPOST$_{RAND}$ for Dell DVD Store.

ing on the more intensive executions or identifying as many bottlenecks as possible at a time but less intensive executions.

2.5 Threats to Validity

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

2.5.1 Internal Validity

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

In our implementation of profiling system, we used Probekit to inject probes into binary code for collecting execution traces, which affects the AUT performance behaviors. However, we only logged for some simple events like current system time for method entry

and exit, and the overhead of Probekit was rather negligible. Thus, we believe that the overhead did not affect the results and current conclusions in our paper.

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

2.5.2 External validity

The main external threat to our experimental design is that we experimented only with three subject AUTs. The results may vary for AUTs that have different logic or different architectures. Furthermore, we only have the authority to access the data and source code of Renters to conduct the experiments in empirical study 1, since it is a closed-source application that belongs to an insurance company. Thus, we did not perform the experiments on Renters in empirical studies 2 and 3. Moreover, due to time consuming, we only perform experiments of sensitivity analysis on JPetStore (requiring more than two months). This threat makes it difficult to generalize the obtained results. There are many other different kinds of systems and different types of performance bottlenecks that can be tested in our experiments. However, since all the applications used are highly representative of enterprise-level applications and frequently used as benchmarks [150, 99] in performance testing research in software engineering, we suggest that our results are generalizable, at least in part, to a larger population of applications from these domains.

To evaluate the effectiveness of FOREPOST, we only compared FOREPOST with random testing and FOREPOST$_{RAND}$. This constitutes a threat, in that if we compare FOREPOST to other performance testing approaches, our results may compare differ-

ently. Thus, it may be difficult to derive general conclusions based solely on the comparisons made. However, the goal of FOREPOST is specific to find input data that leads to intensive computations which identify bottlenecks; and controlled experiments related to different performance testing approaches are difficult to compare. Comparing FOREPOST with FOREPOST$_{RAND}$ made the controlled experiments feasible and also reliable. We suggest that it minimized this threat effectively.

2.5.3 Construct Validity

In this chapter, we used the execution time to measure the AUT performance and cluster execution traces, since the execution time is a representative performance metric and is widely used in performance testing area. The threat is that we did not consider other performance metrics. For example, memory leaks may lead to performance bottlenecks that arise over time, but memory usage is not taken into account in our current version. The methods that automatically scale or reconfigure themselves may also affect the AUT performance, introducing performance bottlenecks. However, our approach is not limited to use only the execution time as performance metric, and it can be extended using other types of performance metrics, like involving different metrics in matrix x (Section 2.2.3). We leave this extension as future work.

2.6 Utilizing FOREPOST in Cloud Computing

In *cloud computing*, stakeholders deploy their software applications on a sophisticated infrastructure that is owned and managed by third-party providers (e.g., public clouds such as Amazon AWS) or in-house installations. Two fundamental properties of cloud computing include provisioning resources to applications on demand and charging their owners for pay-as-you-go resource usage [30]. The elasticity of cloud refers to its capacity to scale resources based on a real workload. Many cloud providers claim that their cloud infrastructures are *elastic*, i.e., they automatically (de/re)allocate resources, both to *scale*

out and *up* – adding resources as demand increases, and to *scale in* and *down* – releasing resources as demand decreases. Using elastic clouds, stakeholders pay only for what they use, when they use it, rather than paying up-front and continuing costs to own and maintain their hardware/software and supporting technical staff [30, 44, 212].

In practice, even the most elastic clouds are not perfectly elastic [142, 30]. Understanding when and how to reallocate resources is a hard problem, since it is generally impossible to quickly and accurately match resources to applications' needs. A recent article underscores this point as it describes its state-of-the-art supervisory system that monitors various black box metrics and then directs the cloud to initiate scaling operations based on that data [110]. As a result, some elasticity-related problems for cloud computing include *under-provisioning* applications so they lack the resources to provide appropriate quality of service, or *over-provisioning* applications so stakeholders end up holding and paying for more resources than they need. Specifically, although elasticity is a fundamental enabler of cost-effective cloud computing, existing *provisioning strategies* (i.e., rules used to (de)allocate resources to applications) are typically obtained in ad-hoc fashion by programmers who study the behavior of the application in the cloud. It is a manual, imprecise, intellectually intensive and laborious effort.

FOREPOST has been shown effective and efficient in building performance behavioral models for AUTs. Thus, we propose a framework, namely *Provisioning Resources with Experimental SofTware mOdeling (PRESTO)*, to enhance cloud elasticity by learning and refining models of software applications (via FOREPOST) through performance testing in the cloud and by using these automatically learned models to help programmers to craft application-specific resource provisioning strategies. That is, PRESTO bridges a pure black-box cloud resource provisioning to software engineering, where behavioral models of the application are re-engineered automatically as part of performance testing, and programmers use these models to create rules for provisioning of resources in the cloud. We evaluated PRESTO on two open-source web-based applications. The results suggest that PRESTO is effective and efficient in achieving precise cloud elasticity by

using software artifacts for guiding resource provisioning in the cloud. All detailed information of PRESTO and experimental results are available in our paper [116] and online appendix [11].

2.7 Related Work

There are many approaches that aid in generating test cases for testing. Avritzer *et al.* proposed an approach that automatically generates test cases and extended it by applying it into a "performability model", which is used to track the resource failures [35]. Partition testing is a set of strategies that divides the program's input domain into subdomains (subsets) from which test cases can be derived to cover each subset at least once [18]. Closely related is the work by Dickinson *et al.* [72], which uses clustering analysis execution profiles to find failures among the executions induced by a set of potential test cases. Although their work and ours used clustering techniques, our work differs in that we cluster the execution profiles based on the length of the execution time and number of methods that have been invoked, and we target performance bottlenecks instead of functional errors.

Load testing is used to determine the AUT performance behaviors under specific workloads. Bayan *et al.* proposed an approach that uses a PID controller to automatically drive the test cases for achieving a pre-specified level of stress/load for a specific resource, such as response time [41]. Another related work by Barna et al. proposed an autonomic framework to explore workload space and identify the points that cause the worst case behavior [40]. It contains a feedback loop that generates workloads, monitors the software system, analyzes the effects of each workload and plans the new workloads. However, this work focuses on the effects of workloads (i.e., number of requests). Thus, it does not consider the effects of different types of requests (e.g., browse, buy, pay) in web applications. The Menasce's work discusses three important activities, load testing, benchmarking, and application performance management, on web-based applications,

and provides a performance models that illustrates the relationship between workload and throughput/response time for improving load testing [210]. Briand *et al.* proposed an approach that uses genetic algorithms to find combinations of inputs that ensure that completion times of a specific task's executions are as close as possible to their deadlines [47]. However, all these approaches do not point out the potential performance bottlenecks in the AUT. In contrast, FOREPOST explores input space and uses machine learning algorithms to identify the combinations of inputs (i.e., requests in web application) for finding performance bottlenecks.

Operational profile is commonly used in performance load testing, where a system can be tested more efficiently because the operations most frequently used are tested the most [220]. It is a quantitative characterization of how the software will be used, which indicates the occurrence probabilities of function calls and the distributions of parameter values. Avritzer *et al.* proposed an approach that uses operational profiles to improve performance testing, where an application-independent performance workload is designed for comparing the existing production with the proposed architecture [37]. In this approach, operational data are collected in the current production environment, and a synthetic workload is fabricated which has a profile close to the average profile compiled by the application in production for the selected operations. However, this work is not aimed at pinpointing specific methods leading to the different performance behaviors of the application.

Learning rules helps stakeholders to reconfigure distributed systems online to optimize for dynamically changing workloads [283]. This work is similar to FOREPOST in using the learning methodology to learn rules, from only low-level system statistics, which of a set of possible hardware configurations will lead to better performance under the current unknown workload. In contrast, FOREPOST uses feedback-directed adaptive performance test scripts to locate most computationally intensive execution profiles and bottlenecks.

There is a recent work that studied 109 real-world performance bugs and found the guidance to detect performance bugs [152]. The study demonstrated the root causes

of performance bugs, thus the efficiency rules should exist and could be collected from patches. Then, the extracted rules from real-world performance-bug patches are used to check performance bottlenecks. These rules are extracted manually, and they are used to analyze software binary code [152], while FOREPOST extracts rules by using machine learning algorithms. Furthermore, the study by Zaman et al. [299] compared performance bugs and the security bugs, and found that performance bugs fixes impacted more files and took more time, while security bugs were fixed and triaged faster, but reopened and tossed frequently, required more developers and were more complex overall.

Another technique related to FOREPOST automatically classifies execution data, collected in the field, which comes from either passing or failing program runs [125]. This technique attempts to learn a classification model to predict if an application run failed using execution data. Jovic *et al.* presented an approach, called *Lag Hunting*, that collects runtime information such as the stack samples, and analyzes this information to detect the latency bugs automatically [155]. Malik *et al.* developed an automated approach that ranked the subsystems that likely involved performance deviations by using the performance signatures [200]. Subsequently they proposed and compared one supervised and three unsupervised approaches for detecting performance deviations automatically for the loading testing in large scale systems, with a smaller and manageable subset of performance counters [201]. Moreover, Syer *et al.* recently combined performance counters and execution logs to detect memory-related issues automatically [263]. On the other hand, FOREPOST learns rules that it uses to select test input data that steer applications towards computationally intensive runs to expose performance bottlenecks.

In the recent work, Zhang, Elbaum, and Dwyer generate performance test cases using dynamic symbolic execution [313]. Similar to FOREPOST, they used heuristics that guided the generation of test cases by determining paths of executions that can introduce higher workloads. Zaparanuks and Hauswirth presented algorithmic profiler that identifies the ingredients of algorithms and their inputs, presented the execution cost by using the repetition tree, and provided the cost function that illustrates the relationship between

the cost and the input size, which can be used to identify algorithms with higher algorithmic complexity [303]. Unlike FOREPOST, white-box testing approach are used, thus requiring access to source code, while FOREPOST is a black-box approach. It is also unclear how the approach [313] will scale to industrial applications with over 100KLOC. We view these approaches as complementary, where a hybrid technique may combine the benefits of both approaches in a gray-box performance testing. This is left for the future work.

2.8 Conclusion and Discussion

In this chapter, we offer a novel solution for automatically finding performance bottlenecks in applications using black-box software testing. Our solution, FOREPOST, is an adaptive, feedback-directed learning testing system that learns rules from execution traces of applications. These rules are then used to automatically select test input data for performance testing. Moreover, we also propose FOREPOST$_{RAND}$, which combine the selected input data with random input data to cover more potential computationally intensive executions. We have applied our approaches to a nontrivial closed-source application at a major insurance company and to two open-source applications in a controlled experiment. The results demonstrate that performance bottlenecks were found automatically in all applications and were confirmed by experienced testers and developers. We compared FOREPOST with FOREPOST$_{RAND}$ on two open-source applications and confirmed that while FOREPOST$_{RAND}$ can identify bottlenecks with higher precision, FOREPOST was able to find the scenarios that lead to substantially more intense computations, which could potentially lead to more serious performance bottlenecks in certain situations. Our results recommend that testers can use FOREPOST$_{RAND}$ for initial performance testing to outline possible roots of performance bottlenecks and use FOREPOST for more focused search of scenarios that result in substantial delays in system execution.

2.9 Bibliographical Notes

The work summarized in this chapter was done in collaboration with Grechanik *et. al* from the University of Illinois at Chicago, which is published in the following papers [196, 198, 116]:

- **Qi Luo**, Aswathy Nair, Mark Grechanik, and Denys Poshyvanyk. "Forepost: Finding performance problems automatically with feedback-directed learning software testing." *Empirical Software Engineering* (2016): 1-51. This paper is based on the following work: Mark Grechanik, Chen Fu, and Qing Xie. "Automatically finding performance problems with feedback-directed learning software testing." In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pp. 156-166. IEEE, 2012.

- **Qi Luo**, Denys Poshyvanyk, Aswathy Nair, and Mark Grechanik. "FOREPOST: a tool for detecting performance problems with feedback-driven learning software testing." In *Proceedings of the 38th International Conference on Software Engineering Companion*, pp. 593-596. ACM, 2016.

- Mark Grechanik, **Qi Luo**, Denys Poshyvanyk, and Adam Porter. "Enhancing rules for cloud resource provisioning via learned software performance models." In *the 7th ACM/SPEC on International Conference on Performance Engineering*, pp. 209-214. ACM, 2016.

Chapter 3

Automating Performance Bottleneck Detection using Search-Based Application Profiling

Although FOREPOST is powerful in detecting performance bottlenecks, it may miss some bottlenecks since it only selects input data based on the learned rules, narrowing down the executions to the specific paths. In addition, it is difficult to learn a precise model from a limited set of execution traces as currently done in FOREPOST (see details in Chapter 2). Inspired by effectiveness of GAs in selecting the optimal solutions as a whole in testing domain [277, 279], we propose a novel approach for automating performance bottleneck detection using search-based application profiling. Our key idea is to use a *genetic algorithm (GA)* as a search heuristic for obtaining combinations of input parameter values that maximizes a *fitness function* that guides the search process [129]. We implemented our approach, coined as *Genetic Algorithm-driven Profiler* (`GA−Prof`) that combines a search-based heuristic with contrast data mining [84] from execution traces to automatically and accurately determine bottlenecks.

This chapter makes the following noteworthy contributions:

- To the best of our knowledge, `GA−Prof` is the first fully automatic input-sensitive pro-

filing approach that explores the input parameter space for detecting performance bottlenecks automatically.

- We evaluated GA−Prof on three popular open-source nontrivial web applications. Our results show that GA−Prof effectively explores a large space of possible combinations of inputs while accurately detecting performance bottlenecks.
- GA−Prof and experimental results are publicly available at [11].

3.1 Problem Statement

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

3.1.1 Background on Input-Sensitive Profiling

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

Input-sensitive profiling departs from the standard profiling metho- dology by inferring the size or the type of the input that can pinpoint performance problems in a software application. Consider an example of the pseudocode that is shown in Figure 3.1. Line

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

Figure 3.1: A pseudocode example of input-sensitive profiling.

1 specifies that input variables x, y, z and u are initialized with some values. In line 2, the value of the variable v is assigned the result of the execution of the method m of A that takes two parameters: x and y and returns their product. In line 3, if the value of v is greater than the value of z, components C and B interact by invoking the method m of B and passing its return value as the parameter to the method h of C. Otherwise, components B and D interact by invoking the method m of B and passing its return value as the parameter to the method h of D. A conclusion that can be inferred from profiling this code depends on specific inputs.

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

3.1.2 Analyzing Profile Data for Bottlenecks

Our illustrative example shown in Figure 3.1 demonstrates two ideas. First, it is not enough to collect performance measurements for some selected input values during profiling – they can be misleading in determining bottlenecks. Consider a situation where a

method is invoked many times in different AUT runs for some combinations of input values. In each separate execution trace the total elapsed execution time of the method may not put it on the top of the list of bottlenecks, however, when analyzed across different traces, these methods may be viewed as bottlenecks based on their overall contribution to the total elapsed execution time.

Second, it is important to distinguish bottlenecks based on their *generality* versus their *specificity* for different input values when using input-sensitive profiling. Some methods are computationally intensive, they implement some important requirements and they are invoked for most of the combinations of input data. The method `main` in Java applications is an example of a generally invoked method. In our illustrative example that is shown in Figure 3.1, these are the method `m` of `A` and the method `m` of `B`. Even though profilers easily put these methods on top of the list of bottlenecks, there is often little that software engineers can do to fix these bottlenecks, since these methods are general-purpose. Another example of such general-purpose bottleneck is a logging facility that records execution events on a persistent media. While some improvements can be performed to make a logging facility more efficient, it is often a necessary overhead. Throughout this paper we call these bottlenecks *natural* as opposed to artificially injected or those that result from incorrect implementation of some requirements. The former bottlenecks are rarely fixed while the latter ones are often considered performance related bugs.

On the other hand, specific bottlenecks are methods that are invoked in response to certain combinations of input values. These bottlenecks are most difficult to find, since they involve an exploration of the enormous space of combinations of the input values that collectively are a small ratio of the total input values space. As it often happens, these bottlenecks remain undetected until the application performance worsens significantly when deployed in the field and used by customers. An important goal of input-sensitive profiling is to increase the specificity of determined bottlenecks by finding a small number of combinations of input values that lead to exposing worsened performance in certain methods of the AUT.

3.1.3 The Problem Statement

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

It is equally important to ensure that the exploration of the input parameter space is not done indiscriminately, since many generated input values may not be contributing anything to measuring the effectiveness of the bottleneck detection algorithm. Consider our motivating example in Figure 3.1, where the input variable `u` may have many values thereby magnifying the input space. Clearly, this parameter does not affect the methods in lines `2-3` and profiling this application with different values for the input variable `u` reduces the efficiency of detecting bottlenecks. Thus, not only is it ineffective to explore the input parameter value space indiscriminately, but it is also highly inefficient (if feasible at all) to profile applications on all combinations of input values. The core problem is how to guide the search process for input values, so that profilers keep extracting useful information for determining and converging on bottlenecks eventually.

Related to the problem of effectiveness and efficiency of input-sensitive profiling is a problem of detecting specific bottlenecks, *i.e.*, those bottleneck methods that become visible only for a small number of combinations of input values. Automatically detect-

ing highly specific bottlenecks is undecidable and very expensive in general. However, multiple evidence show that performance engineers use contrast analysis on collected execution traces, where they analyze correlations among various performance counters with respect to different load profiles [149]. We partially address the problem of determining highly specific bottlenecks in this paper.

3.2 Our Approach

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

3.2.1 Overview of GA-Prof

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

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

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

Identifying performance bottlenecks. Potential performance bottlenecks are detected by using information extracted from multiple traces. Our approach focuses on specific performance problems (not general performance bottlenecks appearing in every application run), which affect AUT's performance significantly. Since the traces are clustered into *good traces* that consume more resources (*e.g.*, execution time) and the *bad traces* that consume less resources, GA−Prof marks a method as a performance bottleneck if it has significant contribution to good traces but less significant contribution to bad traces (see Section 3.2.2.3). A conjecture is that an AUT's specific bottleneck will manifest itself only in a few computationally expensive executions for specific inputs. By extracting these specific performance bottlenecks from collected traces automatically, we make GA−Prof favor the highly specific rather than general bottleneck methods.

3.2.2 Using Genetic Algorithms in GA-Prof

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

3.2.2.1 Background on Genetic Algorithms

GAs are based on the mechanism of natural selection [134] and they use stochastic search techniques to generate solutions to optimization problems. GAs have been widely used in applications where optimization is required but a solution cannot be easily found. The advantage of GA is in having multiple individuals evolve in parallel to explore a large search space of possible solutions. An individual/solution is represented by *chromosome*, *i.e.* a sequence of *genes*.

There are different variations of GAs, but the core idea is that new individuals (*i.e.*, *offspring*) are generated using fitter existing individuals (*i.e.*, *parents*). A pre-defined *fitness function* [134] is used to evaluate the fitness of each individual based on some *fitness value*. Fitter individuals have a better chance to survive. In order to create a new generation, new individuals are created by applying several operators to existing individuals. These operators include (i) a selection operator, (ii) a crossover operator and (iii) a mutation operator. The selection operator selects *parents* based on fitness values. The crossover operator recombines a pair of selected individuals and generates two new individuals. The mutation operator produces a mutant of one individual by randomly altering its *gene*.

3.2.2.2 Why We Use Genetic Algorithms in GA-Prof

GAs are based on heuristic and optimization-based search over solution spaces. An alternative to GAs is to use pattern recognition, such as *machine learning (ML)* algorithms. Specifically, our previous work on FOREPOST showed that it is possible to obtain performance bottlenecks for nontrivial applications with a high degree of precision using feedback-directed learning system [115]. With FOREPOST, execution traces for the AUT are collected, they are assigned to different performance classes (*i.e.*, Good and Bad), and then ML algorithms are used to learn the model of the AUT that maps classes of inputs to different performance behaviors of the AUT (*e.g.*, Good and Bad). Our hypothesis

is that `GA−Prof` is more effective than FOREPOST because determining what combinations of input values reveal performance bottleneck is inherently a search and optimization problem for which GA algorithms are suited the best. Given the complexity of a nontrivial application, it is difficult to learn a precise model from a limited set of execution traces. We confirm this hypothesis with our experimental results in Section 3.4.3. In future work, we will explore a combination of GA and ML approaches to the problem of input-sensitive profiling.

3.2.2.3 Automating Profiling Using GAs

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

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

A *termination criterion* determines when to stop evolution. Usually, there is a maximum limit for the number of generations, meaning that evolution will be terminated when maximum allowed number of generation is reached, which we choose experimentally. Also,

Algorithm 2: `GA−Prof`'s algorithm for automating application profiling

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

in order to improve the efficiency of the GA, the evolution process can also be terminated when the results converge, *i.e.*, their changes among generations become infinitesimal. In `GA−Prof` we monitor the average fitness value of every individual in one generation and we terminate the evolution when results converge.

Our GA implementation includes the following steps: (i) randomly generate an initial set of AUT's inputs, (ii) use them to execute AUT and collect execution traces, (iii) calculate the fitness value of to evaluate the quality of each execution trace, and (iv) use fitness values to guide the evolution and choose new sets of input values. `GA−Prof` takes in the complete set of input ranges for the subject application and the GA configurations, including crossover rate, mutation rate, fitness function and termination criterion. Then, the algorithm generates an initial population by randomly sampling the gene pool of complete input set. Here is when the evolution begins. The crossover operator takes in a pair of *parent* chromosomes, randomly selects a crossover (cutting) point and exchanges the remaining gene sequence, thus creating two *offsprings* for a new generation. The total number of parent pairs is dependent on crossover rate. After that, the mutation operator takes in an offspring chromosome and changes the value of genes with another value within the specified range, thus generating a mutant of the offspring chromosome. The probability of genes being changed is so-called mutation rate. All newly generated individuals are considered a temporary pool and need to evaluated by the pre-defined fitness

function. Each one is assigned with a fitness value and fitter individuals are selected to form a new generation. The selection is based on tournament selection. To select one individual, a tournament is run among a random subset of temporary individuals and the winner is selected, while other individuals are put back to the temporary pool. Multiple tournaments are needed until the new generation meets required population. Thus, a new generation is created. This cycle repeats until termination criterion is satisfied and the final population is returned.

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

3.2.3 Identifying Performance Bottlenecks

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

In order to contrast methods in good/bad execution profiles we rely on the *Independent Component Analysis (ICA)* algorithm that can be used to break large execution traces into sets of orthogonal sets of methods relating to different features of an AUT [137, 115, 113]. ICA algorithm is a computational method that is used to extract components from mixed signals if these components are independent and satisfy the non-Gaussian distribution. ICA has been previously used to address concept location [113] and performance testing problems [115].

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

$$D_{e_g} = \sqrt{\sum_{i=0}^{N_{M_g}} \sum_{j=0}^{N_{R_g}} (S_{Good}^{ij} - S_{Bad}^{kl})^2} \tag{3.1}$$

Since execution traces are clustered into good and bad categories, matrix $\|\mathbf{s}\|$ are generated for both of these two clusters, *i.e.* $\|\mathbf{s}_{Good}\|$ and $\|\mathbf{s}_{Bad}\|$. Based on these two matrices, we rely on the Equation 3.1 to compute specificity weight for each method, where D_{e_g} is the distance for each method, $N_{M_{Good}}$ is the number of good methods, $N_{R_{Good}}$ is the number of features. Since we consider the distance as the weight for each method, we favor potential performance bottlenecks that are significant in good execution traces but not invoked or not significant in bad execution traces. As a result, $\texttt{GA}-\texttt{Prof}$ generates a ranked list of methods based on their weights. Higher ranked methods are identified as bottlenecks with a higher degree of specificity.

3.2.4 GA-Prof's Architecture and Workflow

The architecture of `GA−Prof` is shown in Figure 3.2. Solid arrows indicate command and data flows between components and the numbers in parentheses indicate the sequence of operations in the workflow.

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

When the termination criterion is satisfied, potential bottlenecks are identified using the last generation of individuals (input combinations). However, it should be noticed that the bottlenecks can be also produced `GA−Prof` for any given generation. *Traces Statistics* are passed (7) to *Trace Clustering*, and all traces are divided into two groups: *good* (8) and *bad* (9) execution traces. Clustering is done based on computing the median value of the elapsed execution time. Combining this with *Method and Data Statistics* produced (10) by *Execution Trace Analyzer*, ICA algorithm computes (11) *Method Weights* for each method using Equation 3.1. The higher the method's weight in good execution traces the higher the possibility that a method is a AUT's bottleneck. A ranked list of potential bottleneck methods is generated (12) using their weights and is given to the engineer for further evaluation.

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

Figure 3.2: The architecture and workflow of `GA−Prof`.

3.3 Empirical Evaluation

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

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

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

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

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

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

H_A: There is statistically significant difference in the mean values of elapsed execution times triggered by input combinations generated randomly and by GA−Prof, for subject applications.

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

3.3.1 Subject Applications

We evaluated GA−Prof on three subject applications: JPetStore [156], DellDVDStore [4] and Agilefant [1]. These three applications are all web-based open-source database-centric applications. In these systems, users rely on a web-based *Graphical User Interface (GUI)* front-end to communicate with back-end that accepts URLs as inputs. We deploy JPetStore and DellDVDStore on Apache Tomcat [12] server 6.0.35 and Agilefant on 7.0.47. JPetStore is a Java implementation of the benchmark, PetStore. In our empirical study, we used iBatis JPetStore 4.0.5. The system consists of 2,139 lines of code, 384 methods, 36 classes in 8 packages. JPetStore uses Apache Derby [2] as its back-end database and contains 125 URLs. DellDVDStore is an open-source simulation of an online e-commerce site, which has been used in a number of industrial performance-related studies similarly to JPetStore [147, 149, 52, 61, 256]. DellDVDStore uses MySQL [8] as its back-end database and contains 117 URLs. Agilefant is an enterprise-level backlog product and project management system. It also uses MySQL as its back-end database and contains 124 URLs. We used Agilefant 3.5.1 in our experiments. It consists of 10,848 lines of code, 2,528 methods and 254 classes in 21 packages.

3.3.2 Methodology

Since we use web-based subject applications, the inputs for these applications are URL requests. For instance, JPetStore has a web-based client-server architecture. Its GUI

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

We define a *transaction* as a set of URLs that are submitted by a single user. To answer RQ_1, we issued multiple transactions in parallel collecting profiling traces and computing the total elapsed execution time for the back-end to execute the transactions. Our goal is to evaluate if GA−Prof can automatically find combinations of URLs that cause increase in elapsed execution time. In our experiments, we set the number of concurrent users to five and the number of URLs in one transaction to 50. To answer RQ_2, we randomly selected nine methods in each subject application and injected time delays into them to test whether GA−Prof can correctly identify them. In order to answer RQ_3, we chose FOREPOST [115] as competitive approach (see Section 3.2.2.2). We conducted comparison experiments on subject applications, with artificial delays injected, and compared the effectiveness of both approaches identifying them.

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

Since GA−Prof relies on GAs, which are based on randomized algorithms, we had to conduct our experiments multiple times to ensure statistical significance of the results. We followed the guidelines for statistical tests for assessing randomized algorithms [28, 29] when designing the methodology for our empirical study. We repeated the experiments

for each subject application for 30 times.

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

The GA is implemented using the JGAP library, which provides a collection of methods for a wide range of GA purposes[2]. We used the following GA settings for `GA-Prof`: a crossover rate of 0.3, a mutation rate of 0.1, a population of 30 individuals and a tournament selection of size five. We used the total elapsed time as our fitness function, as described in Section 3.2.2.3. The evolution is terminated if the results do not improve for ten generations. The maximum number of generations is set to 30 – we chose this value experimentally based on the duration of AUTs' runs and the limits of our experimental platform.

3.3.3 Variables

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

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

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

3.3.4 Threats to Validity

A threat to validity for our empirical study is that our experiments were performed on only three open-source web-based applications, which makes it difficult to generalize the results to other types of applications that may have different logic, structure, or input types. However, JPetStore and DellDVDStore were used in other empirical studies on performance testing [147, 149, 256, 61, 52] and Agilefant is representative of enterprise-level applications, we expect our results to be generalizable to at least this type of web-based software applications.

Our current implementation of `GA−Prof` deals with only one type of inputs - URLs, whereas other programs may have different input types. While this is a potential threat, in our opinion, this is not a major one, since `GA−Prof` can be easily adapted to encode inputs of other types. There is no theoretical limitation that prevents `GA−Prof` from profiling other types of applications. In order to apply `GA−Prof` to other applications, one only needs to modify gene representation approach so that `GA−Prof` recognizes other types of input, such as numbers, strings and booleans. However, `GA−Prof` currently does not support complex input types, such as inputs with varying lengths. Additionally, it is possi-

ble that GA−Prof generates invalid URL sequences through the GA operators. This can be solved by extracting special constraints of inputs for each AUT to ensure generated URL sequences are valid, however, it is currently out of the scope of this paper. Moreover, there may be cases where some methods are naturally computationally intensive, yet they are not performance problems. Our current implementation cannot distinguish these cases with the real performance problems, since we only used elapsed execution time to measure method performance. We are planning on addressing these limitations in the future work.

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

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

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

3.4 Empirical Results

This section describes and analyzes the results of our experiments on three software systems in order to answer the research questions stated in Section 3.3.

3.4.1 Searching Through Input Combinations

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

This conclusion is confirmed by the results for DellDVDStore shown in Figure 3.4.1. The average elapsed execution time is ≈ 8.1 seconds in the first generation and ≈ 9.3 seconds in the last generation. We can observe the increase in average elapsed time of approximately 14.8%. This increase is smaller as compared to JPetStore, because DellDVDStore has a relatively smaller and simpler structure, which means that even with randomly generated individuals, significant part of the bottleneck methods are triggered in the first generation, leaving relatively small part of the search space for $\texttt{GA-Prof}$ to explore. However, for those applications with a large input set (*i.e.*, large search space), we expect to see a significant increase in elapsed time.

This conjecture is confirmed by the results of Agilefant, shown in Figure 3.4.1. For the first generation, the mean value of elapsed execution time is ≈ 4.13 seconds, and for the last generation, the average time is ≈ 58.22 seconds. The increase in mean value of elapsed execution time is significant because Agilefant is a much larger system as compared to JPetStore and DellDVDStore, and has a much larger input space. Thus,

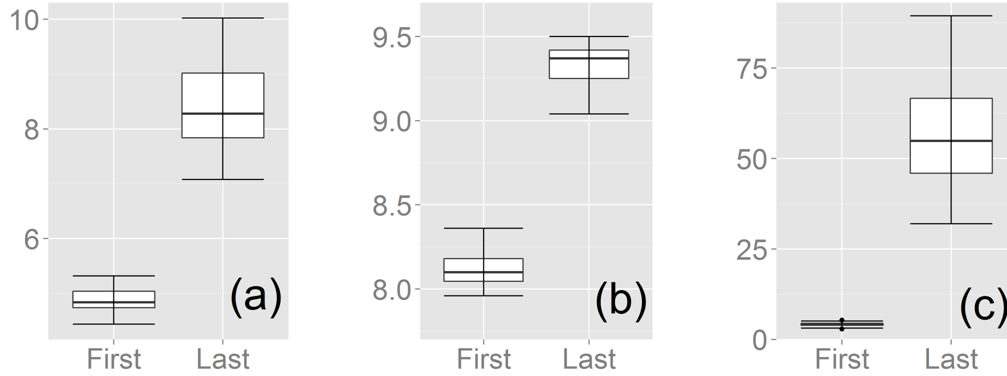


Figure 3.3: Execution elapsed time measured in seconds for subject AUTs. We compare average elapsed times of each transaction in first and last generations for each application. The x-axis corresponds to the first and last generations, and y-axis corresponds to systems' average elapsed time. The results for all three subject applications are averaged over 30 runs. Subfigure (a), (b) and (c) corresponds to JPetStore, DellDVDStore and Agilefant, respectively.

it is more likely that randomly generated combinations of inputs in the first generations may not necessarily be able to focus on the hot spots. Also, the average elapsed times for DellDVDStore and Agilefant to execute one transaction across every generation is shown in Figure 3.4.1 and 3.4.1. As the populations evolve, `GA−Prof` was consistently able to find combinations of inputs that steer applications toward more computationally intensive executions.

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

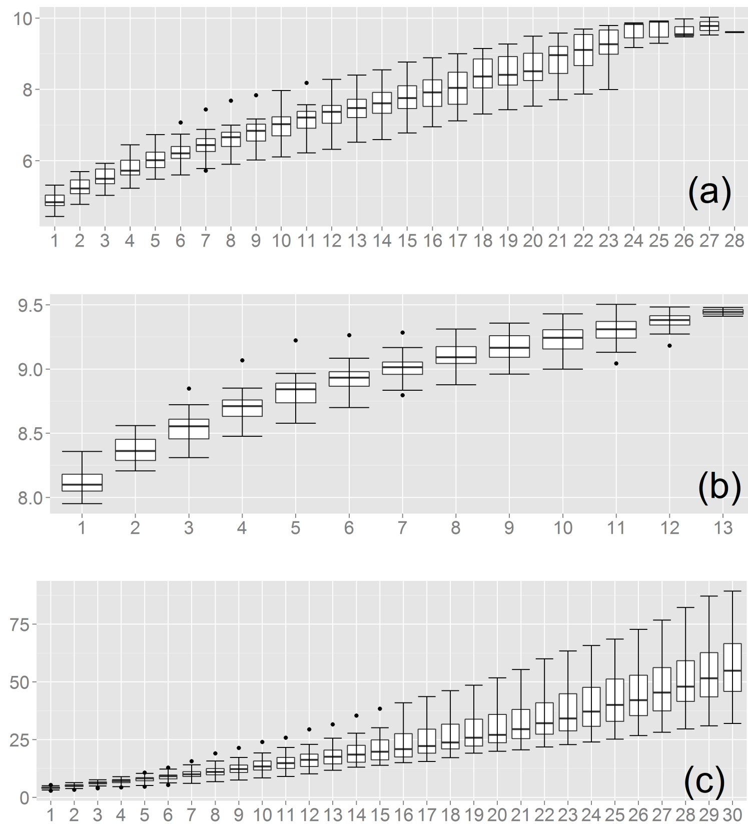


Figure 3.4: The results for elapsed execution time across every generation for each application, measured in seconds. The x-axis corresponds to generations, and y-axis corresponds to average elapsed time. Subfigure (a), (b) and (c) corresponds to JPetStore, DellDVDStore and Agilefant, respectively.

3.4.2 Understanding Performance Bottlenecks

As stated in Section 3.2, `GA−Prof` ranks methods in a descending order and generates a list of potential bottlenecks. Higher ranking indicates the higher probability of being a performance bottleneck. Since we inserted artificial delays into selected methods, we expect these methods (injected bottlenecks) to be ranked higher on the list. We tracked the ranks of each injected bottleneck across generations and we performed linear fitting analysis in order to understand variation and trends in rankings of known bottlenecks.

The *standard deviation* indicates the variation of rankings across generations. For a given injected bottleneck, we take as input the sequence of its ranks. We calculate the standard deviation at each generation using the segment of successive five generations, consisting of the ranks at previous two generations, the current generation and next two

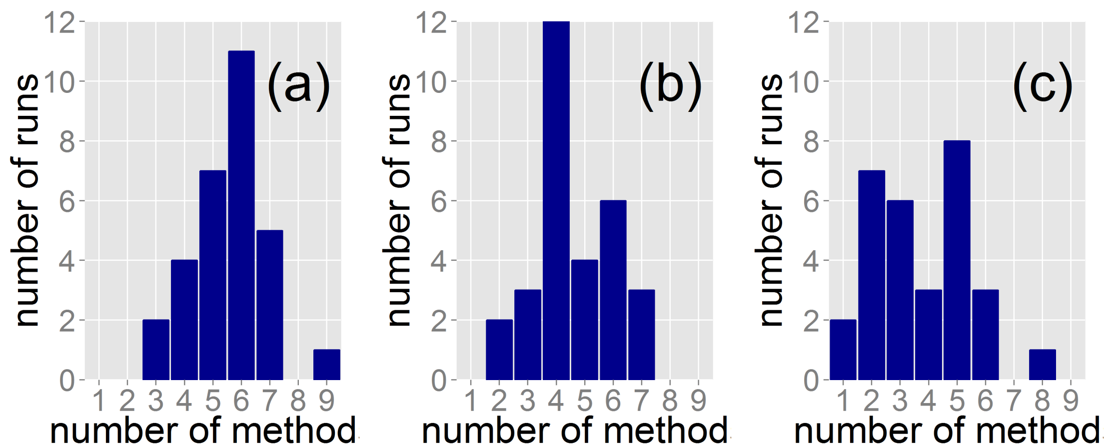


Figure 3.5: Distribution of the quantity of captured injected bottlenecks. The x-axis corresponds to the number of injected bottlenecks that are captured by one certain GA−Prof run. The y-axis corresponds to the number of GA−Prof runs. Subfigure (a), (b) and (c) corresponds to JPetStore, DellDVDStore and Agilefant, respectively.

generations. However, for the first two generations and the last two generations, the value of the standard deviation is assigned to zero because we do not have respective data for generations before and after respectively.

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

If GA−Prof yields a negative slope for the fit straight line for one injected bottleneck, GA−Prof is considered to "capture" this method. If the slope is positive, GA−Prof is considered to "miss" this method. We run GA−Prof multiple times for each subject application, and every GA−Prof run can capture injected bottlenecks. Figure 3.5 shows the distribution of the quantity of captured injected bottlenecks. In experiments with JPetStore (see Figure 5(a)), for most of the time, GA−Prof can capture five or six bottlenecks. The probability of capturing five or more bottlenecks is 80%. The similar distribution pattern can be observed for DellDVDStore and Agilefant, shown in Figure 5(b) and 5(c). To sum up, the average number (expectation) of injected bottlenecks that GA−Prof can capture is 5.6, 4.6, and 3.7 for JPetStore, DellDVDStore and Agilefant, respectively.

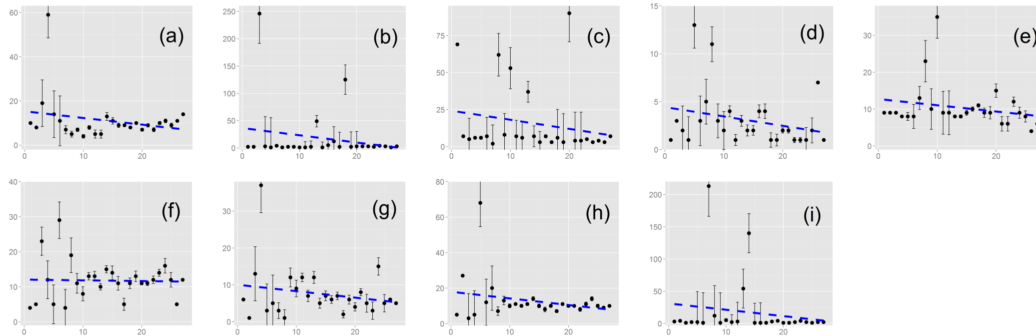


Figure 3.6: Understanding the trend of ranks of injected bottlenecks. The x-axis corresponds to generations, and y-axis corresponds to the rank of bottlenecks. In each subfigure, the rank of the method is shown in black circles. The standard deviation at each generations is shown in black vertical lines and whiskers. The fit straight line is shown is blue dashed lines.

One example of `GA−Prof` run on JPetStore is shown in Figure 3.6. We can see that at most times, injected bottlenecks ranked within top 20 of the descending list, which means that `GA−Prof` 's output is stable and reliable. However, there are some cases where the rank of a bottleneck method is ranked as low as taking the position on the list below 200 and then comes back to the top of the list, for example, Figure 3.4.2. This phenomenon is expected, since our approach is search-based and it can choose input values for some generations that are not optimal. `GA−Prof` approaches to the target (the bottlenecks) by continuous self-correction. It is expected that sometimes `GA−Prof` experiences some "over-correction", which is when we observe a very low ranking of a method. This is inevitable, however, it is not a concern. The method will come back later on top of the list in future generations, as proved by the figures. As a result, `GA−Prof` will eventually yield a reliable list of methods where injected bottlenecks are ranked on top. This can be demonstrated by the fit linear line (blue dashed lines in the figures). In the example in Figure 3.6, we observe a negative slope for all nine methods, which means that the ranking of all nine injected bottlenecks are converging to the top of the list as the `GA−Prof` evolves. However, we do not expect that `GA−Prof` would always be able to capture every single injected bottleneck. A positive slope does not always mean that the method is missed. Sometimes a method is ranked on top of the list at every generation, leaving no

Table 3.1: Comparing GA-Prof and FOREPOST for detecting performance bottlenecks in JPetStore (JP) and DellDVDStore (DS). All numbers are averaged over multiple runs. "# of Methods" indicates the number of injected bottlenecks that are captured by one certain technique. "Final Ranks" indicates the ranks of injected bottlenecks in the final ranked list.

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

space for improvement, thus, the slope can not be negative. Sometimes a method may give way to another method but still stay within top positions of the list. These two cases do not impair the reliability of the ranked list at all. In summary, results demonstrate that GA−Prof is effective in identifying injected bottlenecks, thus, **positively addressing *RQ*$_2$**.

3.4.3 Comparing GA-Prof to FOREPOST

Recall from Section 3.2.2.2 that FOREPOST is the closest competitive approach to GA−Prof that uses machine learning to obtain models that map classes of inputs to performance behaviors of the AUT [115]. Like GA−Prof, FOREPOST outputs a descending list of potential bottlenecks.

In our comparison experiments, we used two configurations for FOREPOST. In `config1`, we used four iterations of learning rules and ten execution traces in between. In `config2`, we used four iterations and 15 execution traces. Since FOREPOST experiments are very time-consuming, we repeated FOREPOST experiments five times for only two subject applications: JPetStore and DellDVDStore. The results are shown in Table 3.1, where we compared the following: 1) how many injected bottlenecks are captured (titled as"# of Method"), and 2) final ranks of injected bottlenecks (titled as"Final Ranks"). Capturing a bottleneck is defined in Section 3.4.2. By "final ranks", we mean the average of all injected bottlenecks rankings in last generation (GA−Prof) or last iteration (FOREPOST).

Table 3.1 shows that GA−Prof was able to capture, on average, 5.6 injected bottlenecks in JPetStore, while FOREPOST captured only 1.8 and 2.2 bottlenecks in two re-

spective configurations. Similarly, for DellDVDStore, `GA−Prof` also captured more bottlenecks. Final ranks are injected bottlenecks' rankings over multiple runs. Smaller numbers represent higher positions in the list, indicating higher probability of being performance problems. For JPetStore, the injected bottlenecks have an average rankings of 13.78 in the list by `GA−Prof`, and 241.67 and 145.98 by FOREPOST. For DellDVDStore, injected bottlenecks are also ranked higher by `GA−Prof`. In summary, `GA−Prof` finds more bottlenecks than FOREPOST, confirming our initial conjecture, and, thus, **positively addressing *RQ*$_3$** that `GA−Prof` is more effective than FOREPOST in identifying performance bottlenecks.

3.5 Related Work

Profiling, a form of dynamic program analysis, is widely used in software testing, such as test generation [88, 268, 173], functional fault detection [31, 242, 314, 32, 63, 154], and non-functional fault detection [287, 66, 222, 71, 190, 123, 291]. Korel provided an approach that generates test cases based on actual executions of AUT to search for the values of input variables, which influence undesirable execution flow, by using function minimization methods [173]. Artzi *et al.* used the Tarantula algorithm to localize source codes which lead to failures in web application by combining the concrete and symbolic execution information [32]. An approach provided by Jiang *et al.* utilizes execution profilers that possibly contain faults to simplify the program and scale down its complexity for in-house testing [146]. But these works only focused on functional faults. Coppa *et al.* provided an approach to measure how the performance scales with increasing size of input, and used it to find out performance faults by analyzing the profiles [66, 94]. Liu *et al.* designed an innovative system, AutoAnalyzer, to identify existence of performance bottlenecks using clustering algorithms and to locate performance bottlenecks by searching algorithm [190]. However, these two papers only paid attention to some specific problems, whereas `GA−Prof` is aimed at exploring and detecting all possible performance

bottlenecks.

Genetic Algorithms (GAs) is widely used in many areas of software engineering [126], such as software maintenance [185, 215, 231], textual analysis [233], cloud computing [101, 128] and testing [16, 15, 127, 208, 53, 38, 109, 289, 203, 204, 205]. Test generation is a key point in software testing. Alshahwan *et al.* used dynamically mined value seeding into search space to target branches and generate the test data automatically [16]. To achieve higher branch coverage, McMinn *et al.* used a hybrid global-local search algorithm, which extended the Genetic Algorithm with a Memetic algorithm, to generate the test cases [130, 100]. Ali *et al.* provided a systematic review for the search-based test case generation, which built a framework to evaluate the empirical search-based test generation techniques by measuring cost and effectiveness [15]. Briand *et al.* applied GAs to stress testing. They developed a method for automatically deriving test cases to maximize the probability of critical deadline misses [48]. In Wegener *et al.*'s work [276, 278, 277], GAs were shown to find unknown execution times, which also used GAs for selecting test input data and exposing performance problems. However, they looked for the longest as well as the shortest execution times. Moreover, they did not repeat their experiments to account for the randomness of GAs. Also, their decision about when to stop evolution was rather arbitrary. Finally, `GA-Prof` uses contrast mining to detect specific bottlenecks across different sets of inputs and profiles.

Performance Testing. Finding and fixing performance problems was shown to be even more challenging than identifying functional problems [301]. Thus, one critical goal in performance testing is to automatically generate test cases which may invoke performance problems. Burnim *et al.* provided a complexity testing algorithm for the symbolic test generation tool, to construct the inputs that lead to the worst-case computational complexity of the program [50]. Jin *et al.* extracted efficiency-related rules from 109 real-world performance bugs, and used them to detect performance bugs [151]. Xiao *et al.* propose an approach that predicts workload-dependent performance bottlenecks by using complexity models [285]. Zhang *et al.* proposed an approach for exposing perfor-

mance bottlenecks using test cases generated by a symbolic-execution based approach [312]. However, unlike GA−Prof, they did not utilize execution information to identify performance problems.

3.6 Conclusion and Discussion

In this chapter, we propose a novel approach for automating performance bottleneck detection using search-based application profiling. Our key idea is to use a genetic algorithm as a search heuristic for obtaining combinations of input parameter values that maximizes a fitness function that represents the elapsed execution time of the application with these input values. We implemented our approach, coined as *Genetic Algorithm-driven Profiler (GA−Prof)* that combines a search-based heuristic with contrast data mining from execution traces to accurately determine performance bottlenecks. We evaluated GA-Prof in the empirical study to determine how effectively and efficiently it detects injected performance bottlenecks into three popular open source web applications: two popular performance benchmarks and one enterprise-level application. Our results demonstrate that GA-Prof effectively explores a large space of the combinations of the input values while automatically and accurately detecting performance bottlenecks. Moreover, we compare GA−Prof to FOREPOST, and the experimental results show that GA−Prof is more effective than FOREPOST because determining what combinations of input values reveal performance bottlenecks is an inherently search and optimization problem for which GAs are best suited for.

Recall that chapter 2 presents FOREPOST which extracts rules from execution traces to generate test data for finding performance problems and identifying bottlenecks. Both FOREPOST and GA−Prof approaches are aiming at finding specific combinations of input sets that steer application execution to hot paths. However, GA−Prof uses genetic algorithms for exploring a large space of input combinations in the context of automating application profiling. Moreover, our experimental results confirm that GA−Prof demon-

strate superior results as compared to those by FOREPOST, which is rooted in our original conjecture - it is difficult to learn a precise model from a limited set of execution traces as currently done in FOREPOST.

3.7 Bibliographical Notes

The work summarized in this chapter was done in collaboration with Mark Grechanik from the University of Illinois at Chicago and Du Shen (the lead author) from the College of William and Mary. It is published in the following paper [258]:

- Du Shen, **Qi Luo**, Denys Poshyvanyk, and Mark Grechanik. “Automating performance bottleneck detection using search-based application profiling.” In *the 2015 International Symposium on Software Testing and Analysis*, pp. 270-281. ACM, 2015.

Chapter 4

Mining Performance Regression Inducing Code Changes in Evolving Software

During software evolution, a number of code changes are committed, and some of them may be responsible for performance regressions. A performance regression is a situation in which an *application under test* (AUT) exhibits unexpectedly worsened performance in a new release as compared to the previous version for the same input values and for a given *workload* (i.e., the number of users, their requests and frequencies of interactions). Stakeholders are interested in understanding code changes behind these regressions.

Performance regression testing is challenging due to at least the following reasons. Firstly, modern software systems evolve rapidly. Many of them follow agile-driven cycles and release new versions in short iterations [54]. With a large number of commits submitted, the cost of detecting performance regressions and linking code changes to performance behaviors increases drastically. Therefore, performance regression testing is usually performed continuously during software maintenance [45, 131]. Secondly, detecting performance regressions and locating the associated code changes for specific inputs in AUTs with large spaces of input combinations are non-trivial and time-consuming

tasks [226].

Let's consider a simplified scenario for detecting performance regressions. Assume there are two versions of an AUT, a newly released version (v_{i+1}) and a previous version (v_i). Programmers commit a number of changes between these two versions. Given the same test inputs, v_i and v_{i+1} the application may exhibit different performance behaviors with respect to its execution time. The test inputs that lead to worsened performance (e.g., longer execution time) in v_{i+1} but not in v_i are the desired inputs that may expose new performance regressions. Their corresponding execution traces are helpful for troubleshooting [131]. In order to find such inputs, stakeholders need to iterate through a large number of input combinations while mining the execution traces for both of v_i and v_{i+1} with the same inputs to monitor changes in performance for each input set. It is challenging for stakeholders to mine a large body of execution traces for identifying the ones can expose potential performance regressions and linking the inputs to these traces. Once such inputs are found (manually or automatically), the corresponding execution traces need to be further examined to detect changes responsible for observed performance regressions. Unfortunately, this process is domain and knowledge dependent, oftentimes manual and expensive.

We propose a novel recommendation system, PerfImpact, to automatically recommend inputs and code changes for programmers that may be closely related to performance regressions using a combination of search-based input profiling [258] and change impact analysis [176]. The search-based input profiling has been extended to execute two different releases of AUT (v_i and v_{i+1}) independently with the same input values, mine execution traces to link inputs with AUT's behaviors, and use a genetic algorithm as a search heuristic for exploring the input value combinations for finding the ones likely exposing performance regressions.

After the inputs are selected, PerfImpact mines the execution traces generated with these inputs, and uses change impact analysis to rank each code change based on its contribution to the AUT's performance regression(s). The code changes having significant

impact on AUT's performance degradation in v_{i+1} are marked as problematic for follow-up code reviews. The goal of PerfImpact is to improve effectiveness of performance regression testing via identifying input combinations than worsen performance behaviors (i.e., longer execution time) in v_{i+1}, and mining the corresponding execution traces to prioritize code changes likely responsible for these regressions. It is possible that some code changes with longer execution time implement new features or fix bugs, not necessarily leading to performance regressions. Our approach may not precisely locate root cases behind performance regressions, but provide a ranked list of code changes potentially leading to regressions that can be used as a starting point for programmers in regression testing. This chapter makes the following contributions:

- We propose a novel recommendation system, PerfImpact, that relies on search-based input profiling to expose performance regressions manifested in newer software versions, mines the corresponding traces, and uses change impact analysis to prioritize the code changes likely responsible for these performance regressions;

- We empirically evaluated PerfImpact on different releases of two open-source web applications, Agilefant ($v_{3.2}$, $v_{3.3}$, and $v_{3.5}$) and JPetStore ($v_{3.0.0}$ and $v_{4.0.5}$) containing numerous real changes. The results demonstrate that PerfImpact is able to effectively explore the combinations of input values and identify performance regressions between different releases. The results also demonstrate that PerfImpact can effectively recommend the changes (both real and injected) likely responsible for the identified regressions;
- We have made the experimental results publicly available in my online appendix [11].

4.1 Problem Statement

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

4.1.1 State of the Art and Practice

Many recent approaches aim at detecting performance regressions by comparing the values of different performance metrics (e.g., performance counters) in two system versions [227, 226, 179]. Typically, they execute the same test cases in each version and use control charts to check if the performance of a target test in v_{i+1} is similar to the performance of a baseline test in v_i. Other approaches use statistical methods, such as ANOVA, to detect performance differences between v_{i+1} and v_i [131]. All these approaches require running a complete set of test cases for detecting regressions. However, since performance testing is usually time-consuming [227], it is imperative to identify a subset of effective inputs or test cases more likely to exhibit performance regressions. While techniques for selecting regression tests have been proposed and evaluated in the context of functional testing [91, 168, 185, 288, 290], generating and selecting performance regression tests still remains a significant challenge.

Understanding which code changes are responsible for particular performance regressions poses to be even more challenging problem. Precisely pinpointing changes (out of thousands of commits) that may be responsible for performance regressions (for certain inputs) is a fairly involved task, requiring deep knowledge of the AUT's source code, behavioral semantics, and even change history. The closest approach to address this problem is the one by Huang *et al.* who proposed a model for estimating the risk of each commit and tagging commits likely leading to performance regressions [135]. This solution relies on static analysis and focuses on specific types of performance regressions, such as dramatic cost difference in intra-procedural paths and loop termination conditions affected by code changes (it does not identify changes responsible for input-specific bottlenecks).

```
                 Vi                              Vi+1
public synchronized         1    public synchronized         1
       void calculate();    2           void calculate();    2

input a, b                  3    input a, b                  3
A item;                     4    A item;                     4
                            5    if (a > b)                  5
item = new A();             6           item = new A();      6
                            7    else item = A.getItem();    7
item.calculate();           8    item.calculate();           8
```

Figure 4.1: A performance regression example due to possible thread blocking.

4.1.2 An Example Performance Regression

Let's consider the example shown in Fig. 4.1. This example illustrates that understanding AUT's behaviors and their relationships to input values (and combinations of inputs) is critical for detecting performance regressions. The example shows code snippets in two versions of a system, v_i and v_{i+1}. In both versions, lines 1-2 declare method $calculate()$ as a synchronized method. Line 3 presents input variables a and b, and line 4 the object $item$ of the type A is instantiated. In v_i, lines 5-7 assign a new instance to $item$; while, in v_{i+1}, lines 5-7 assign a new instance to $item$ or invoke method $getItem()$ to assign an existing instance to $item$, depending on the result of the branch condition in line 5. In both v_i and v_{i+1}, $item$ calls method $calculate()$ in line 8. Note that $calculate()$ is a synchronized method, so if it is called with the same instance in multiple threads simultaneously, the threads will be blocked. However, in v_{i+1}, $item$ is assigned an existing instance if the branch condition in line 5 is not satisfied. Thus, when multiple threads are executing concurrently and sharing the same instance of an $item$, method $calculate()$ may be blocked, which can lead to a performance regression for certain inputs of a and b in v_{i+1}, but not in v_i. Moreover, even if the input values leading to this performance regression are identified, it may be difficult to locate code changes responsible for this performance regression. If we simply rely on total execution time to evaluate performance, we would be able to observe performance degradation, for certain inputs, for method $calculate()$. Yet, in this case, the actual changes responsible for the performance regression are those in line 5 and line 7 in v_{i+1}.

4.1.3 The Problem Statement

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

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

4.2 Approach

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

4.2.1 An Overview of Our Approach

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

Finding Inputs That Lead to Performance Regressions. The first key idea of PerfImpact is to rely on search-based input profiling [258] to mine execution traces for understanding AUT's performance behaviors, and use genetic algorithms (GAs) to explore different combinations of input values for finding the ones that take unexpectedly longer time to execute in v_{i+1} but not in v_i. Our hypothesis is that the input value combinations with larger execution time difference among two studied versions are more likely to trigger performance regressions. While search-based input profiling has been recently used for detecting performance bottlenecks in a given software version [258], PerfImpact instruments and runs two versions of the AUT with the same inputs independently. PerfImpact also defines a new fitness function aimed at mining execution traces to obtain the ones using more time to complete in v_{i+1} than in v_i and selecting input combinations associated with these executions. This fitness function is designed as a proxy for identifying inputs leading to performance regressions in v_{i+1}.

Identifying Code Change That Induce Performance Regression by Mining Execution Traces. The second key idea is to find the changes associated with the methods related to performance degradations. Specifically, PerfImpact obtains execution times of the invoked methods in v_{i+1} and v_i during profiling and compares their performance dif-

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

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

Figure 4.2: Examples of URLs and a chromosome in our GA implementation. Each number in the chromosome refers to a unique URL ID.

ferences respectively. The methods with increased execution time in v_{i+1}, for the same inputs as in v_i, are tagged as potentially "problematic". Given a code change, PerfImpact relies on dynamic change impact analysis (CIA) [176] to mine execution traces and estimate a set of methods (i.e., an impact set) that is potentially impacted by this code change. Then, all the changes between v_{i+1} and v_i are ranked based on the performance of the methods in their respective impact sets. The changes that have more "problematic" methods in their impact sets are ranked higher. Conversely, the changes that have fewer or no "problematic" methods in their impact sets are ranked lower. The heuristic is that the higher ranked changes usually have more significant impact on performance regressions.

4.2.2 Search-based Input Profiling for Performance Regressions

Search-based input profiling mines a large body of execution traces and utilizes GAs to automatically search the input space for possible combinations of inputs responsible for the performance regressions. GAs are evolutionary algorithms that mimic the natural selection process to search for the solutions to optimization problems [134, 216], and have been widely used to generate test cases in the software testing domain [127, 139, 130]. In GAs, a solution or an individual is represented as a chromosome, which contains a sequence of genes. Typically, the initial individuals are generated randomly, and then GAs exploit a pre-defined fitness function to evaluate each individual. The fitter ones (i.e., parents) that have larger fitness values are selected to generate the individuals for the next generation (i.e., offsprings) via genetic operators, such as crossover and mutation.

Parent 1: 2, 18, 36, 27, 11, 13, 6, 43, 64, 12, 85, 49, 12, 53, 44, 91, 79, 23, 3, 19
Parent 2: 23, 95, 1, 67, 35, 81, 7, 17, 51, 102, 56, 39, 72, 3, 54, 37, 13, 86, 47, 76

Child 1: 2, 18, 36, 27, 11, 13, 6, 17, 51, 102, 56, 39, 72, 3, 54, 37, 13, 86, 47, 76
Child 2: 23, 95, 1, 67, 35, 81, 7, 43, 64, 12, 85, 49, 12, 53, 44, 91, 79, 23, 3, 19

(a) The crossover operator in GAs.

Parent: 2, 18, 36, 27, 11, 13, 6, 43, 64, 12, 85, 49, 12, 53, 44, 91, 79, 23, 3, 19

Child: 2, 18, 36, 27, 11, 13, 6, 43, 64, 12, 85, 49, 73, 53, 44, 91, 79, 23, 3, 19

(b) The mutation operator in GAs.

Figure 4.3: The examples of GA operators, crossover and mutation.

The key idea behind our GA implementation is to identify the input combinations likely to expose performance regressions. In our implementation, an individual (i.e., a chromosome) refers to a test case (or a set of inputs). Each chromosome contains a sequence of genes, referring to the inputs with different parameters. In case of a web-based application that takes URLs as inputs, the example of a chromosome encoding is shown in Fig. 4.2. Each URL is assigned an unique ID and a chromosome encoding represents a sequence of URL IDs. An URL input containing different parameters (e.g., URL 3 and 4 shown in Fig. 4.2) will be assigned different IDs. The implementation of crossover and mutation operators is illustrated in Fig. 4.3. The crossover operator selects a pair of parent chromosomes (i.e., ID sequences) and randomly chooses a cut point to swap these two sequences. The mutation operator takes a chromosome and changes the value of a selected gene (i.e., an ID) with another random value. The probabilities of these two operations are predefined as the crossover and mutation rates.

We define a fitness function to evaluate inputs and promote the ones that are more likely to trigger performance regressions. PerfImpact first mines execution traces to extract time information for each combination of inputs, then measures the inputs using the *time difference*, which is defined as the difference between the times it takes v_{i+1} and v_i to execute with the same inputs. The larger the *time difference*, the higher the probability that the corresponding inputs might lead to performance regressions. We define the fitness function as shown in Eq. 4.1, where I_j is a set of inputs selected from the whole AUT input set (i.e., I_{all}), td_j is the *time difference* for input I_j, t_j is the time it takes AUT

to execute I_j, the superscripts 'i' and '$i+1$' refer to v_i and the v_{i+1} software releases respectively.

$$td_j = t_j^i - t_j^{i+1} \tag{4.1}$$

Our GA implementation is outlined in Alg. 3, which takes the whole AUT input set (I_{all}) and two releases (v_i, v_{i+1}) as inputs, and outputs the sets of inputs (I) for which performance regressions are observed. In detail, the initial population is selected randomly from I_{all} (1). Then crossover and mutation operators are executed with the pre-defined rates (r_c, r_m) on the initial population to generate new individuals (3-4). After that, each individual is sent as an input to v_i and v_{i+1}, and two traces are collected during the profiling (5-7). Then the fitness value is calculated based on the pre-defined fitness function (Eq. 4.1) for each individual (8-9). The fitter ones are selected to create the next generation (10). The above process repeats until the termination criterion is reached (2), and then sets of inputs (I) are returned (11-12). Typically, there are two types of termination criteria. One is a pre-defined maximum number of generations and the other one is the average fitness value. When the maximum number of generations is reached or the children's average fitness value does not increase significantly as compared to their parents' average fitness value (the increased percentage is less than a pre-defined threshold), the evolution process is terminated. The values of two types of termination criteria are settled experimentally (Section 4.3.3).

4.2.3 Identifying Performance Regression Inducing Changes via Mining

In general, performance regressions are exposed when some specific methods experience longer execution time in v_{i+1}. PerfImpact relies on path-based dynamic CIA [176] to identify the changes leading to performance regressions. For each change, the impact analysis is used to build an impact set containing all the methods that are potentially impacted by this change. PerfImpact mines execution traces to understand the performance of the impacted methods in two releases to rank the changes. *The key hypothesis here is that if the methods in the impact set exhibit longer execution times in v_{i+1} but not*

Algorithm 3: The Genetic Algorithm.

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

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

in v_i, for the same sets of inputs, then it is more likely that a change for this impact set is responsible for the observed performance regression. Obviously, there may be cases where multiple inputs and changes are responsible for one or multiple performance regression(s) (i.e., some fault interaction may be present [73]). Note that CIA may not be helpful to accurately locate the code causing performance regressions. However, our goal is to pinpoint a starting point (i.e., changes related to observed performance regressions) for a detailed root cause analysis that needs to be performed by developers. In our paper, the code changes are extracted at the method level granularity. In particular, we consider changes in a method between v_{i+1} and v_i involving additions, modifications or deletions to the body, signature, or a return type, excluding comments.

The impact analysis technique that we rely upon in our implementation considers a change's impact that propagates along any (and only) dynamic paths that pass through the change [176]. Given a change c, only the methods, which are called after c and which are in the call stack after c returns, are added into the impact set. For example, three execution traces are shown in Fig. 4.4. Given a method a, a_e represents a method's entry and a_r represents a method's return. x represents the execution termination. In fig. 4.4, in the first execution, m is called first, then m calls b, b calls c, c calls f, f and c return, b

Algorithm 4: Ranking changes for a given set of inputs.

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

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

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

Figure 4.4: Three sample execution traces of an AUT.

returns, m returns, and finally the execution terminates. Assuming that the method c has been changed, its impact set in the first execution is {b, f, m}, since f is called after c, and b, m are in the call stack after c returns. Similarly, its impact set is {a, f, m} in the second execution, and its impact set is {b, e, m} in the third execution. Thus, the final impact set for the method c is the union of these three sets, which is {a, b, e, f, m}.

In PerfImpact, a trace is collected for one set of inputs. We considered the trace segment of one distinct input (i.e., a URL) as an execution, so each trace can be divided into different executions corresponding to different inputs. In CIA, when one trace contains multiple executions, the backward and forward searching do not cross the termination symbol of each execution (i.e., x in Fig. 4.4). For a web application, one set of inputs refers to a sequence of URLs, thus a trace is collected for each sequence of URLs. Each trace can be divided into different trace segments for different URLs. For example, if there are 50 URLs in one set of inputs, the corresponding trace is divided into 50 trace segments, where each segment refers to one execution used in CIA.

For a given set of inputs, the impact set of each change is estimated using CIA. PerfImpact mines execution traces to obtain the performance differences of each method in the impact set and ranks the code changes based on their impacted methods' perfor-

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

4.2.4 **Workflow of** PerfImpact

The workflow of PerfImpact is shown in Fig. 4.5. Solid arrows indicate command and data flows between components, and the numbers in circles indicate the sequence of operations in the workflow. The dashed arrows denote transition in control flow once GA termination criteria is satisfied. Initially, sequences of inputs (i.e., individuals) are selected randomly for the first generation (1). While our paper starts this step (i.e., GA component) with random inputs, in practice, developers can also supply inputs that reveal performance bottlenecks in v_i (or any other inputs they would like to start with). JMeter [153] simulates users sending the inputs into two releases of the AUT automatically (2-4). *Profiler*$_i$ and *Profiler*$_{i+1}$ collect execution traces of each set of inputs on v_i and v_{i+1} respectively (5, 6). *Profilers* are implemented using Probekit [10], a lightweight profiling tool that injects

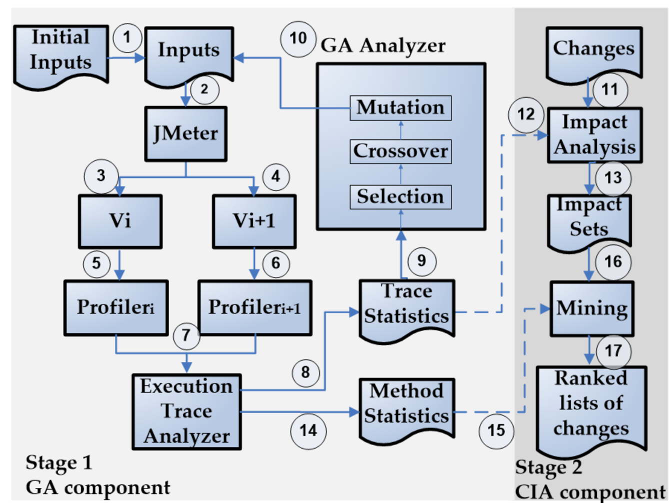


Figure 4.5: The workflow of PerfImpact.

the code fragments into specific points (e.g., method entry and exit) of the binary code for collecting the runtime data. *Execution Trace Analyzer* processes the execution traces (7) and extracts *Trace Statistics* (8) for *GA Analyzer* to evaluate each set of inputs (9). *GA analyzer* calculates the fitness value for each set of inputs according to Eq. 4.1 and selects the fitter ones to generate new inputs. The new inputs are sent back the AUT, starting the next iteration (10). GAs are implemented using JGAP [7].

After the GA component terminates, which means that PerfImpact finds the inputs likely to expose performance regressions, the second stage of PerfImpact (i.e., CIA component) is initiated with these inputs. By combining the *Change* information (e.g., full method names, signatures, return types) (11) and *Trace Statistics* (12), an *Impact Set* is derived for each change for the given inputs, using the *Impact Analysis* algorithm (13). *Method Statistics* are extracted to calculate the execution time in two releases for each method (14). In *Mining* phase, PerfImpact integrates *Method Statistics* (15) with *Impact Sets* (16), and uses the Alg. 4 to rank the changes for the given inputs (17). The changes ranked higher on the list are the ones likely leading to performance regressions. Note that the CIA component is initiated right after the GAs' search is terminated, since we expect mining execution traces for selected inputs to be useful to analyze the impact of each change on performance regressions. Alternatively, the CIA component can be also run

simultaneously while running the GA component. This usage of PerfImpact depends on two specific scenarios. In the first scenario, when stakeholders want to obtain the final ranked lists of changes, they can run the CIA component after GA component is terminated, as shown in Fig. 4.5. However, if stakeholders prefer to monitor the impact of inputs on performance changes, they can run the CIA component for the inputs that are selected at each generation (second scenario). To evaluate PerfImpact thoroughly, we choose the second scenario for our empirical study (section 4.3.3).

4.3 Evaluation

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

4.3.1 Research Questions

RQ$_1$: How effective is PerfImpact in finding inputs that likely expose performance regressions in v_{i+1}?

RQ$_2$: Can PerfImpact effectively recommend changes between v_i and v_{i+1} likely responsible for performance regressions in v_{i+1} for a given set of inputs?

To answer **RQ$_1$**, we introduced the following null (H_0) and alternative (H_1) hypotheses aimed at comparing inputs selected by PerfImpact with random inputs. Inputs with larger *time differences* (defined in Eq 4.1) are more likely to lead to performance regressions. The hypotheses are evaluated at a 0.05 level of significance:

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

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

To answer **RQ$_2$**, after GA component is finished and changes are ranked, we run AUTs with the selected inputs to further understand the changes' impact on performance of two

releases. We expect the changes ranked higher would lead to much longer execution time in v_{i+1} as compared to v_i.

4.3.2 Subject AUTs

We evaluated PerfImpact on two open-source web applications, JPetStore ($v_{3.0.0}$, $v_{4.0.5}$) and Agilefant ($v_{3.2}$, $v_{3.3}$, $v_{3.5}$). The statistics for all subjects are shown in Table 5.2. JPetStore [156] is a three-tier Java implementation of PetStore, which is widely used as performance benchmark [147, 148, 256, 98]. The GUI front end accepts users' URL requests, and the backend executes the requests and communicates with its database. Both JPetStore versions are deployed in Tomcat 6.0.35 and rely on Apache Derby 10.6.2.1 [2] as the backend database. Agilefant [1] is an open source application for managing agile software development, written in Java. All versions of Agilefant are deployed in Tomcat 7.0.47 with MySQL as the backend database.

4.3.3 Methodology

The *first goal* of the empirical study is to determine that whether the inputs selected by PerfImpact are likely to trigger performance regressions. To achieve this goal, we ran PerfImpact to obtain the inputs and compared them with randomly selected inputs. Random inputs are widely used in the testing field as they appear to be remarkably effective and reliable in test case generation [236, 122]. *Time difference* (see Eq. 4.1) was chosen to evaluate both the selected and random inputs. The inputs with larger *time differences* were more likely to trigger performance regressions.

Table 4.1: The stats of the subject programs.

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

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

We chose three pairs of AUT releases, JPetStore $v_{3.0.0}$ and $v_{4.0.5}$, Agilefant $v_{3.2}$ and $v_{3.3}$, and Agilefant $v_{3.2}$ and $v_{3.5}$, to evaluate PerfImpact . Two types of changes, *real* and *injected*, were involved. To extract the real changes, we computed diffs for each pair of releases [3]. Some changes were ignored since their inputs cannot be tested in our experiments (e.g., an input that triggers specific functionality that removes the same data from database and, hence, causes a database error). As a result, we extracted 68 changes between JPetStore $v_{3.0.0}$ and $v_{4.0.5}$, 24 changes between Agilefant $v_{3.2}$ and $v_{3.3}$, and 95 changes between Agilefant $v_{3.2}$ and $v_{3.5}$. Furthermore, we also wanted to determine how well PerfImpact is able to identify the known problematic changes. Thus, we also injected artificial changes in the second set of experiments. Injecting artificial changes to mimic

the real performance regressions has been widely used in evaluating the effectiveness of performance regression testing techniques [131, 226, 255]. We randomly injected nine artificial changes (three for each group) into the source code of v_{i+1} (JPetStore $v_{4.0.5}$, Agilefant $v_{3.3}$ or Agilefant $v_{3.5}$). All these changes will lead to the synchronization problems similar in nature to one explained in the illustrative example (section 4.1.2), which would lead to longer latency during execution. The complete information on the injected changes is provided in my online appendix [11].

The inputs in our study were URLs, since we focused on web applications. One sequence of URLs sent by one user is defined as a *transaction*. Once URLs are selected randomly or by PerfImpact, JMeter simulates multiple users sending transactions into two releases of the AUT, and their backends executing URL requests independently (see Fig. 4.5). Each transaction contained 50 URLs, and the number of users for the initial workload was set to five. Since PerfImpact selected random URLs to generate the initial population, it was necessary to conduct every experiment multiple times to avoid skewed results. Following the guidelines for using statistical tests to assess randomized algorithms [27, 26], we ran our experiments with the same configurations thirty times on JPetStore and ten times on Agilefant. That is, we ran JPetStore with random inputs thirty times and Agilefant with random inputs ten times. For each time, the number of combinations of inputs is equal to the number of individuals per generation. After identifying performance regression inducing changes, we also experiment with increased workloads (5, 10, 15, 20 and 25 users) to analyze these changes' impacts on performance regressions. The experiment with the same workload was run five times.

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

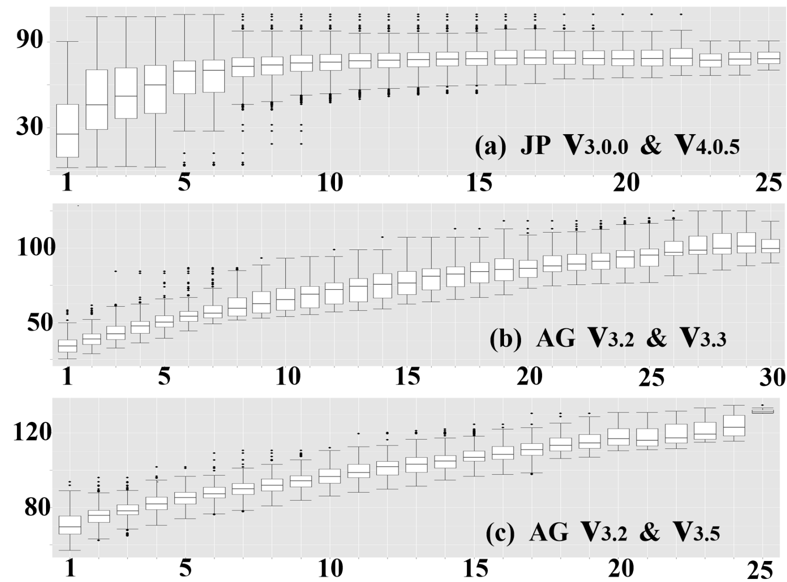


Figure 4.6: The box-and-whisker plots represent *time differences* between two released versions across generations on JPetStore (JP) and Agilefant (AG).

needs more than five days to finish one run on our hardware infrastructure).

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

4.4 Empirical Results

This section analyzes the results of our empirical study. More experimental results are available online [11].

4.4.1 Finding Performance Regression Inputs

Fig. 4.6 shows the results of *time differences* between two releases across GA generations on JPetStore and Agilefant. The x-axis represents the generations, and the y-axis represents *time differences* between two releases (in seconds). The central box represents the values from the lower to upper quartile (i.e., 25 to 75 percentile). The middle line represents the median. The vertical line extends from the minimum to the maximum

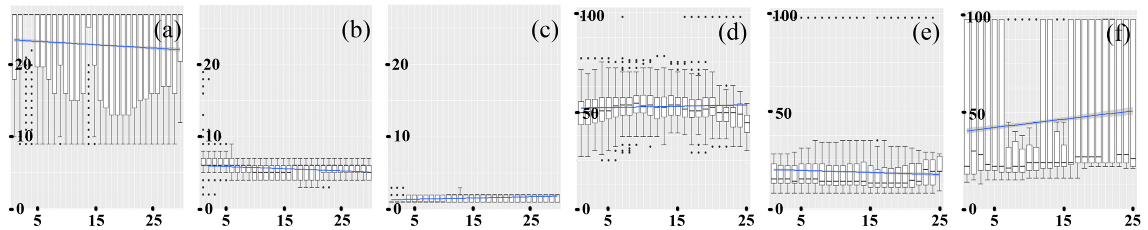


Figure 4.7: The box-and-whisker plots represent the ranks of the changes in Table 4.3. The x-axis represents the generations, and the y-axis represents the ranks. Smaller values that appear on y-axis imply higher ranks.

value. Note that, if a set of inputs leads to larger *time difference*, this set is likely to trigger performance regressions. As shown in Fig. 4.6, the *time difference* increases as the GAs progress, implying that PerfImpact steered execution of the AUTs to the paths which triggered performance regressions. Specifically, Table 4.2 compares the *time differences* of selected inputs in the last generation with the random inputs in the first generation. The average *time differences* for the selected inputs are significantly larger than the time differences for the random inputs ($162.35\% - 288.72\%$ increase), which clearly demonstrates that the inputs selected by PerfImpact were more likely to trigger performance regressions. The values of the standard deviation (SD) of the selected inputs are much smaller as compared to the random inputs for JPetStore. We suggest that the selected inputs converge to a stable subset of inputs. However, the values of SD of the selected inputs are larger as compared to the random inputs in Agilefant. Recall that Agilefant has relatively more sophisticated architecture than JPetStore. Thus, PerfImpact has more chances to steer the executions to different paths, leading to larger values of SD. Additionally, a paired t-test with one-tailed distribution was performed to compare the *time differences* of random inputs and selected inputs. The $p - value$ of these three groups are significantly smaller than 0.05. Based on these results we reject the null hypothesis. *These results demonstrate that* PerfImpact *can find the combinations of inputs that were significantly more effective as compared to random inputs in exposing these performance regressions*.

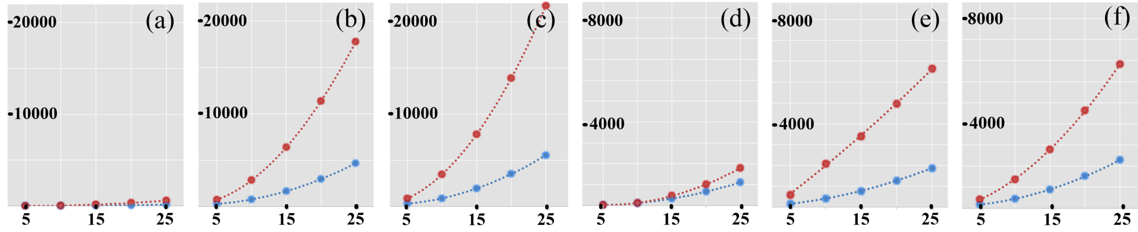


Figure 4.8: The figures show the average of total execution times of the changes in Table 4.3. This total execution time of one change is the total execution time of all methods in its respective impact set. The blue dots show the average of total execution time in old version of Agilefant ($v_{3.2}$), and the red dots show the average of total execution time in new version of Agilefant ($v_{3.3}$ or $v_{3.5}$). The curves are the fitting curves generated using Polynomial Function model. The inputs were selected in the last generation. The x-axis represents the average of total execution time, and the y-axis represents the number of users. Time is measured in seconds.

Table 4.2: The *time difference* between two versions for random inputs (Rd) and PerfImpact selected inputs (PI) in JPetStore (JP) and Agilefant (AF).

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

4.4.2 Identifying Code Changes

To evaluate PerfImpact's effectiveness in identifying problematic code changes, we provide the rankings of six randomly chosen code changes from Agilefant as examples, including five real and one injected change. The detailed information on the changes is shown in Table 4.3. Due to lack of space, the experimental results for other changes can be found in the online appendix [11]. Fig. 4.7 shows the ranks of these changes across generations. The central box represents the values from the lower to upper quartile (i.e., 25 to 75 percentile). The middle line represents the median. The vertical line extends from the minimum to the maximum value. The blue lines are the fitting lines generated using generalized linear model. For Agilefant, there are 27 changes (i.e., 24 real and three injected changes) between $v_{3.2}$ and $v_{3.3}$, and 98 changes (i.e., 95 real and three injected changes) between $v_{3.2}$ and $v_{3.5}$, thus the range of ranks in $v_{3.3}$ was from 1 to 27

(a)

V3.2

```
public String generateTree(){
    Set<Integer> selectedBacklogIds = this.getSelectedBacklogs();
    if(selectedBacklogIds == null || selectedBacklogIds.size() == 0) {

        addActionError("No backlogs selected.");
        return Action.ERROR;
    }  ......
    return Action.SUCCESS;
}
```

V3.5

```
public String generateTree(){
    Set<Integer> selectedBacklogIds = this.getSelectedBacklogs();
    if(selectedBacklogIds == null || selectedBacklogIds.size() == 0) {
        Collection<Product> products = new ArrayList<Product>();
        productBusiness.storeAllTimeSheets(products);
        for (Product product: products) {
            selectedBacklogIds.add(product.getId());
        }
    }   ......
     return Action.SUCCESS;
  }
```

(b)

V3.2

```
public StoryTreeBranchMetrics calculateStoryTreeMetrics(Story story) {  ......
   for(Story child : story.getChildren()) {

        StoryTreeBranchMetrics childMetrics = this.calculateStoryTreeMetrics(child);
    }  ......
    return metrics;
}
```

V3.5

```
public StoryTreeBranchMetrics calculateStoryTreeMetrics(Story story) {  ......
  for(Story child : story.getChildren()) {
     if (child.getId() == story.getId()) {
        continue;
     }
     StoryTreeBranchMetrics childMetrics = this.calculateStoryTreeMetrics(child);
  }  .....
  return metrics;
}
```

Figure 4.9: Examples of code changes in Agilefant. (a) shows the source code of change (f) in Table 4.3, and (b) shows the source code of change (d) in Table 4.3.

and the range of ranks in $v_{3.5}$ was from 1 to 98. Note that, the methods with smaller values (close to one) for ranks are ranked higher. Fig. 4.7 shows that the ranks for changes vary in the first generation, since the inputs are generated randomly. As the GAs progress, the executions are steered to the paths where the performance regressions are exposed, thus the ranks of some changes (e.g., change (b), (c), (d) and (e)) become more stable and converge to the final ranks.

Based on the stable ranks in the last generation, we can easily identify two types of changes. One change type that has relatively higher ranks (i.e., smaller values on y-axis in Fig. 4.7), such as changes (b), (c), and (e), is identified as representing problematic changes. Specially, change (c) is an injected change. We also checked the ranks of other injected changes. All of them were ranked on the top, demonstrating that PerfImpact can effectively identify the injected changes. The other change type that has noticeably lower ranks (i.e., larger values on y-axis in Fig. 4.7), such as change (d), is identified as the one less likely to trigger performance regressions. Unlike the changes that have stable ranks in the last generation, change (a) and (f) vary significantly. We further analyzed their ranks to understand the reason behind these variations. Change (f) had relatively higher median ranks (middle lines in boxplots), implying that it may trigger performance regressions for some specific inputs. We will discuss its source code later to show more details. However, the median ranks of change (a) were close to the bottom (i.e, rank 27

in $v_{3.3}$), implying that it was not invoked for most of the selected inputs and it had less contribution to performance regressions. PerfImpact tended to discard the inputs less likely to trigger performance regressions as the GAs progressed, thus the corresponding methods were not invoked. In conclusion, based on the ranks in the last generation, we can identify different types of changes.

Table 4.3: Examples of code changes in Agilefant.

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

To demonstrate that the changes with higher ranks were likely to trigger performance regressions, we ran the selected inputs on AUTs with different workloads (i.e, different numbers of users) and obtained the average total execution times for each change in two releases. In general, one change with longer total execution times in v_{i+1} is more likely to trigger performance degradation. As the results show in Fig. 4.8, the changes with higher ranks (e.g., changes (b), (c), (e) and (f)) have much larger averages of the total execution times in v_{i+1} (i.e., red lines in Fig. 4.8) as compared to the ones in v_i (i.e., blue lines in Fig. 4.8). We used polynomial functions to fit the results, demonstrating that the average of the total execution times increased nonlinearly when the workload increased. The polynomial functions for all examples in Table 4.3 are shown in our online appendix [11]. Conversely, the changes with lower ranks (e.g., changes (a) and (d)) have relatively shorter average total execution times in both v_i and v_{i+1}. Recall that change (a) was not invoked by most of selected inputs. Its averages of total execution times in $v_{3.2}$ and $v_{3.3}$ were close to zero. As expected, the changes with higher ranks led to longer execution

times in v_{i+1}, and the times increased nonlinearly given an increase in the workload.

To further demonstrate that PerfImpact identified the problematic changes effectively, we looked into the source code of each change. Fig. 4.9 shows two examples of such code changes. More examples are available in the online appendix [11]. Fig. 4.9 (a) shows the source code of change (f) in Table 4.3, which was ranked highly for some selected inputs. As expected, PerfImpact found the inputs that satisfied the *if* clauses, which led to different performance in two releases. In $v_{3.2}$, the method was returned directly with a $Action.ERROR$. Instead, in $v_{3.5}$, it called $storeaAllTime$- $Sheets$ to obtain a collection of $Products$, and added products' IDs into $selectdBacklogIds$. Then, the execution went through the following steps in change (f). Apparently, change (f) required more time to execute in $v_{3.5}$, especially when the size of the $products$ increased, leading to a performance regression. Note that the inputs that did not satisfy the *if* clause would not lead to performance degradation. This example demonstrates that PerfImpact can find specific inputs that trigger the performance regressions and effectively locate the problematic changes. Fig. 4.9 (b) shows the source code of change (d) in Table 4.3, which got relatively lower ranks in PerfImpact. The change was that, in the *for* loop, the current iteration would be skipped in $v_{3.5}$, when $story.getId$ was equal to $child.getId$. Apparently, change (d) would not degrade the performance in $v_{3.5}$, thus it was correctly ranked lower by PerfImpact. *These results show that* PerfImpact *can be used to effectively identify the changes that are responsible for performance regressions.*

4.5 Threats to Validity

First, our current implementation of PerfImpact only focuses on the identical input values that are valid for both releases, v_i and v_{i+1}. The differences in inputs between two releases, such as the new inputs in v_{i+1} that may no longer be valid in v_i, were not tested, since they cannot be sent into both of two releases for performance comparison. Moreover, when generating new inputs, some constraints (e.g., the order of URLs in a chro-

mosome) must be considered to guarantee that the new inputs are valid. However, our current implementation deals with some straightforward constraints, such as a login with a predefined username and the password at the beginning. Testing different inputs between two releases and considering other constraints are currently out of the scope of this paper and we leave them for future work.

Second, PerfImpact does not analyze root causes behind detected performance regressions and does not take into account potential interactions among performance regressions [73, 169]. Multiple inputs and changes may be responsible for one or many performance regressions, thus, our approach may not necessarily be able to capture cases where the behaviors of performance regressions are changing due to interactions among those regressions (e.g., a situation where one performance regression obscures effects of another regression for certain inputs). Also, if an AUT is multithreaded, even if it runs twice with the same input, the execution time may be different due to multithreaded interleavings.

Third, in our empirical study, we only applied PerfImpact to several releases of two open-source web applications. It is hard to generalize the results given that our experiments are based on the two applications (even though we considered five releases of these two apps in total). However, JPetStore has been widely used as a benchmark in performance testing [147, 148, 256, 98] and Agilefant is an enterprise-level real-world application. Thus, we believe that these applications are representative real-world software systems. Also, another potential threat is that we only considered one type of inputs (i.e., URL requests), since we experimented with web-based applications. However, PerfImpact can be used with other types of applications and inputs (the chromosomes can be reformatted to accommodate other types of inputs). We leave this extension for future work.

Finally, we only injected one type of artificial changes to simulate performance regressions. Also we had to discard some real changes since they can not be covered by PerfImpact. However, we extracted 187 different real changes in the subject appli-

cations. Thus, we believe that all the changes (real and injected) used in evaluation constitute a solid experimental design to support our current conclusions. Furthermore, PerfImpact only focuses on method-level changes in the native source code. Currently, PerfImpact does not take into account different granularity and possible changes in the underlying third-party or standard libraries. While analyzing the impact of changes in underlying libraries on the performance of a client application is an important problem [119], we leave it for the future work.

4.6 Related Work

Change Impact Analysis is a technique aimed at helping developers to understand the effects of a change on the rest of the source code [183, 181]. Many CIA approaches have been proposed [105, 39, 182, 78, 46]. Law and Rothermel proposed a dynamic path-based impact analysis, which assumes that a change has a potential impact on the code reachable from this change [176]. Following this approach, Apiwattanapong *et al.* presented a method that only considers essential dynamic information by using execute-after sequences [24]. Ren *et al.* presented a tool, Chianti, to identify the changes that induce the failure of one specific test [244]. Zhang *et al.* introduced FaultTracer, which adapts spectrum-based fault localization techniques with a CIA-based algorithm to rank the changes for identifying failure-inducing ones [307, 308, 309]. However, these approaches do not focus on performance regressions. To the best of our knowledge, PerfImpact is the first technique to combine CIA with search-based input profiling to analyze the impact of changes on an AUT's performance.

Regression Testing. The default approach for regression testing is to retest all test cases after releasing a new version, which is an expensive proposition. To solve this problem, a number of techniques for selecting regression tests have been proposed [91, 168, 62, 250, 108, 295, 301, 298, 229]. Table 4.4 shows approaches that have been proposed to support performance regression testing. There are *three* major dif-

ferences between these approaches (see Table 4.4). *First*, some approaches rely on profiling of the AUT and some do not. Profiling is a well-established and useful technique for analyzing the AUT's behaviors, and is widely used in performance testing field [179, 213]. PerfImpact uses differential profiling to run the same inputs in two software versions simultaneously, which enables accurate detections of performance regressions. *Second*, some approaches mine information from repositories to identify performance regressions [98]. However, many software systems may not necessarily maintain well-structured repositories. PerfImpact detects performance regressions without relying on the testing history, which makes it applicable to other contexts including testing legacy systems. *Third*, performance regression testing is not completed until the code changes responsible for performance regressions are identified. Yet, only a very few approaches address this concern. For instance, Huang *et al.* detect high-risk commits that may lead to performance regressions using static analysis [135]. However, this work relies on static analysis and focuses on specific types of performance regressions. A recent work analyzes root causes behind performance regressions, yet it requires the AUT to maintain an accurate set of unit tests [131]. On the contrary, PerfImpact does not require unit tests and relies on dynamic information to automatically and effectively identify actual bottlenecks (that can be observed and confirmed at run-time) as well as problematic changes.

4.7 Conclusion and Discussion

In this chapter, we propose a novel recommendation system, PerfImpact, aimed at automatically recommending code changes likely responsible for performance regressions. Our approach uses search-based input profiling to detect input combinations likely leading to performance regressions, and mines execution traces to estimate the impact of code changes on detected performance regressions. As compared to GA-Prof (see chapter 3), which uses GAs to search for input values leading to performance bottlenecks in a

Table 4.4: Performance regression testing approaches.

Approaches	Analysis Static	Analysis Dynamic	Profiling	Repository	Identify Changes
Our approach	·	●	●	·	●
Shang *et al.* [255]		●	·	·	·
Huang *et al.* [135]	●	·	·	·	●
Nguyen *et al.* [225]	·	●	·	●	●
Heger *et al.* [131]	·	●	●	·	●
Lee *et al.* [179]	·	●	●	·	●
Nguyen *et al.* [226]	·	●	·	·	·
Foo *et al.* [98]	·	●	·	●	·
Mostafa *et al.* [218]	·	●	●	●	●
Mi *et al.* [213]	·	●	●	·	·
Chen *et al.* [59]	·	●	·	·	·
Kalibera *et al.* [160]	·	●	·	·	·
Bulej *et al.* [49]	·	●	·	·	·
Yilmaz *et al.* [292]	·	●	·	·	·

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

We implemented PerfImpact and tested it on different releases of two open-source web applications. The results demonstrate that PerfImpact can effectively select the inputs exposing performance regressions. Also, the ranked lists of changes computed with PerfImpact are useful for stakeholders to identify potential changes behind performance regressions for further inspection and root cause analysis. In the future, we are planning on conducting further empirical studies to understand characteristics of performance

bottlenecks and tailor our proposed approaches to other granularities (e.g., feature-level [75, 76, 77, 240, 241, 246]) in addition to method-level granularity. For example, we plan to recover traceability links between performance bottlenecks with features, which would support software engineers to locate problematic features and further detect more relevant performance bottlenecks.

4.8 Bibliographical Notes

The work summarized in this chapter was done in collaboration with Mark Grechanik from the University of Illinois at Chicago, which is published in the following paper [197]:

- **Qi Luo**, Denys Poshyvanyk, and Mark Grechanik. "Mining performance regression inducing code changes in evolving software." In *the 13th International Conference on Mining Software Repositories (MSR)*, pp. 25-36. ACM, 2016.

Chapter 5

How Do Static and Dynamic Test Case Prioritization Techniques Perform on Modern Software Systems? An Extensive Study on GitHub Projects

Modern software evolves at a constant and rapid pace; developers continually add new features and fix bugs to ensure a satisfied user base. During this evolutionary process, it is crucial that developers do not introduce new bugs, known as software *regressions*. *Regression testing* is a methodology for efficiently and effectively validating software changes against an existing test suite aimed at detecting such bugs [192, 307]. One of the key tasks of the contemporary practice of continuous regression testing, is test case prioritization (TCP).

Regression test prioritization techniques reorder test executions in order to maximize a certain objective function, such as exposing faults earlier or reducing the execution

time cost [192]. This practice can be readily observed in applications to large industrial codebases such as at Microsoft, where researchers have built test prioritization systems for development and maintenance of Windows for a decade [261, 68]. In academia, there exists a large body of research that investigates the design and evaluate regression TCP techniques [271, 306, 248, 247, 192, 172]. Traditionally, TCP techniques leverage one of several code coverage measurements of tests from a pervious software version as a representation of test effectiveness on a more recent version. These approaches use this measured test adequacy criterion to iteratively compute each test's priority, and then rank them to generate a prioritized list. Researchers have proposed various forms of this traditional approach to TCP, including greedy (total and additional strategies) [306, 248, 247], adaptive random testing [145], and search-based strategies [184].

While dynamic TCP techniques can be useful in practice, they may not be always applicable due to certain notable shortcomings, including: 1) the time cost of executing an instrumented program to collect coverage information [106, 206]; 2) expensive storage and maintenance of coverage information [206, 311]; 3) imprecise coverage metrics due to code changes during evolution or thread scheduling of concurrent systems [180], and 4) the absence of coverage information for newly added tests [192] or systems/modules that disallow code instrumentation [180] (e.g., code instrumentation may break the time constraints of real-time systems). Thus, to offer alternative solutions that that do not exhibit many of these shortcomings, researchers have proposed a number of TCP techniques that rely solely upon *static* information extracted from the text of source and test code. Unfortunately, since the introduction of purely static TCP techniques, little research has been conducted to fully investigate the effectiveness of static techniques on modern software. This begs several important questions in the context of past work on dynamic techniques, such as: *How does the effectiveness of static and dynamic techniques compare on modern real-world software projects? Do static and dynamic techniques uncover similar faults? How efficient are static techniques when compared to one another?* The answers to these questions are of paramount importance as they will guide future research

directions related to TCP techniques.

Several empirical studies have been conducted in an attempt to examine and understand varying aspects of different TCP approaches [247, 92, 82, 243, 266]. However, there are clear limitations of prior studies that warrant further experimental work on TCP techniques: 1) recently proposed TCP techniques, particularly static techniques, have not been thoroughly evaluated against each other or against techniques that operate upon dynamic coverage information; 2) no previous study examining static TCP approaches has comprehensively examined the impact of different test granularities (e.g., prioritizing entire test classes or individual test methods), the efficiency of the techniques, or the similarities in terms of uncovered faults; 3) prior studies have typically failed to investigate the application of TCP techniques to sizable real-world software projects, and none of them have investigated the potential impact of program size (i.e., LOC) on the effectiveness of TCP techniques; 4) prior studies have not comprehensively investigated the impact of the quantities of faults used to evaluate TCP approaches; and 5) no previous study has attempted to gain an understanding of the impact of fault characteristics on TCP evaluations.

Each of these points are important considerations that call for thorough empirical investigation. For instance, studying the effectiveness and similarity of faults uncovered for *both* static and dynamic techniques could help inform researchers of potential opportunities to design more effective and robust TCP approaches. Additionally, evaluating a set of popular TCP techniques on a large group of sizable real-world java programs would help bolster the generalizability of performance results for these techniques. Another important consideration that arises from limitations of past studies is that an increasing number of studies use mutants as a proxy for real faults to evaluate performance characteristics of TCP techniques. Thus, understanding the effect that mutant quantities and operators have on mutation analysis-based TCP evaluations should help researchers design more effective and reliable experiments, or validate existing experimental settings for continued use in future work. Therefore, in this paper we evaluate the effectiveness of TCP

approaches in terms of detecting mutants.

To answer the unresolved questions related to the understanding of TCP techniques and address the current gap in the existing body of TCP research we perform an extensive empirical study comparing four popular static TCP techniques, i.e., call-graph-based (with total and additional strategies) [311], string-distance-based [178], and topic-model based techniques [266] to four state-of-the-art dynamic TCP techniques (i.e., the greedy-total [247], greedy-additional [247], adaptive random [145], and search-based techniques [184]) on 58 real-world software systems. All of the studied TCP techniques were implemented based on the papers that initially proposed them and the implementation details are explained in Section 5.2.4. It is important to note that different granularities of dynamic coverage information may impact the effectiveness of dynamic TCP techniques. In this paper, we examine statement-level coverage for dynamic techniques, since previous work [192, 206] has illustrated that statement-level coverage is *at least as effective* as other common coverage criteria (e.g., method and branch coverage) in the TCP domain. In our evaluation criteria we examine the effectiveness of the studied techniques in terms of the Average Percentage of Faults Detected (APFD) and its cost cognizant version APFDc. Additionally, we analyze the implications of these two metrics on efficacy measure of TCP techniques and discuss the implications of this analysis. We also analyze the impact of subject size and software evolution on the two studied metrics. Furthermore, during our empirical study, we vary the operator types and the quantities of injected mutation faults to investigate whether these factors significantly affect the evaluation of TCP approaches. We also examine the similarity of detected of faults for the resultant prioritized sets of test cases generated by our studied TCP techniques at different test granularities (e.g., both method and class levels). More specially, we investigate the total number and the relative percentages of different types of mutants detected by the most highly prioritized test cases for each TCP technique to further understand their capabilities in detecting faults with varying attributes. Finally, we examine the efficiency, in terms of execution time (i.e., the processing time for TCP technique), of static TCPs to

better understand the time cost associated with running these approaches.

Our study bears several notable findings. When measuring the average APFD values across our subject programs, we found that the call-graph-based (with "additional" strategy) technique outperforms all studied techniques at the test-class level. At the test-method level, the call-graph and topic-model based techniques perform better than other static techniques, but worse than two dynamic techniques, i.e., the additional and search-based techniques. Furthermore, results from these experiments indicate that different techniques perform differently, in a statistically significant manner, between all studied TCP techniques based on APFD values. *Our results demonstrate that APFDc values are generally consistent with APFD values at test-class level but relatively less consistent at test-method level.* When examining the effectiveness of TCP approaches in terms of the cost-cognizant APFDc values, we found that the call-graph-based (with "additional" strategy) technique outperforms all studied dynamic and static techniques at both test-class and test-method levels, indicating the limitations of dynamic execution information in reducing actual regression testing time costs. While when examining the APFDc values at the test-method level, we found that the additional and search-based (dynamic) techniques even perform worse than the call-graph-based (with "additional" strategy) technique. Additionally, while APFDc values vary dramatically across 58 subject programs, based on the results of our analysis, there are no statistically significant differences between TCP techniques based on APFDc values at both of test-class and test-method level when controlling for the subject program. *Furthermore, our results indicate that the test granularity dramatically impacts the effectiveness of TCP techniques*. While nearly all techniques perform better at method-level granularity based on both of APFD and APFDc values, the static techniques perform comparatively worse to dynamic techniques at method level as opposed to class level based on APFD values. Our study shows that subject size and software evolution tend not to largely impact experimental results measuring TCP performance. Our results also demonstrate that experimental settings regarding the fault quantities and types used in typical evaluations of TCP techniques tend

not to significantly impact the results of experiments measuring effectiveness. In terms of execution time, call-graph based techniques are the most efficient of the static TCP techniques. Finally, the results of our similarity analysis study suggest that there is minimal overlap between the uncovered faults of the studied dynamic and static TCPs, with the top 10% of prioritized test-cases only sharing $\approx$ 25% - 30% of uncovered faults. Thus, the most highly prioritized test cases from different TCP techniques exhibit dissimilar capabilities in detecting different types of mutants. This suggests that certain TCP techniques may be better at uncovering faults (or mutants) that exhibit certain characteristics, and that aspects of different TCP techniques may be combined together to alter performance characteristics. Both of these findings are promising avenues for future work. To summarize, this paper makes the following noteworthy contributions summarized in Table 5.1.

5.1 Background & Related Work

In this section we formally define the TCP problem, introduce our studied set of subject studied techniques, and further differentiate the novelty and research gap that our study fulfills. Rothermel et al. [248] formally defined the test prioritization problem as finding $T' \in P(T)$, such that $\forall T'', T'' \in P(T) \wedge T'' \neq T' \Rightarrow f(T') \geq f(T'')$, where $P(T)$ denotes the set of permutations of a given test suite T, and f denotes a function from $P(T)$ to real numbers. In the next two subsections, we introduce the underlying methodology utilized by our studied static TCP techniques (Section 5.1.1) and dynamic TCP techniques (Section 5.1.2). Details of our own re-implementation of these tools are discussed later in Section 3. All studied techniques attempt to address the TCP problem formally enumerated above with the objective function of uncovering the highest number of faults with the smallest set of most highly prioritized test cases. As defined in previous work [133, 266], a white-box TCP approach requires access to *both* the source code of subject programs, and other types of information (e.g., test code), whereas black-box techniques do not re-

Table 5.1: The List of Contributions

Contributions	Descriptions
Static vs. Dynamic TCP	To the best of the authors' knowledge, this is the first extensive empirical study that compares the effectiveness, efficiency, and similarity of uncovered faults of both static *and* dynamic TCP techniques on a large set of modern real-world programs;
Impact of Performance Metrics	We evaluate the performance of TCP techniques based on two popular metrics, APFD and APFDc, and understand the relationship between the performance of these two metrics for TCP evaluation;
Impact of Test Case Granularity	We evaluate the performance of TCP techniques at two different test granularities, and investigate the impacts of test granularities on TCP evaluation;
Impact of Program Subject Size	We evaluate the impacts of subject size on the effectiveness of the studied static and dynamic TCP techniques;
Impact of Software Evolution	We evaluate the impacts of software evolution on the effectiveness of the studied static and dynamic TCP techniques;
Impact of the Number of Studied Faults	We conduct the first study investigating the impact of different fault quantities used in the evaluation on the effectiveness of TCP techniques;
Impact of Fault Types	We conduct the first study investigating the impact of different fault types used in the evaluation on the effectiveness of TCP techniques;
Practical Guidelines for Future Research	We discuss the relevance and potential impact of the findings in the study, and provide a set of learned lessons to help guide future research in TCP;
Open Source Dataset	We provide a publicly available, extensive online appendix and dataset of the results of this study to ensure reproducibility and aid future research [11].

quire the source code or test code of subject programs, and grey-box techniques require access to only the test-code. Most dynamic techniques (including the ones considered in this study) are considered white-box techniques since they require access to the subject

system's source code. In our study, we limit our focus to white and grey-box static TCP techniques that require only source code and test cases, and the dynamic TCP techniques that only require dynamic coverage and test cases as inputs for two main reasons: 1) this represents fair comparison of similar techniques that leverage traditional inputs (e.g., test cases, source code and coverage info), and 2) the inputs needed by other techniques (e.g., requirements, code changes, user knowledge) are not always available in real-world subject programs. Additionally, we discuss existing empirical studies (Section 5.1.3).

5.1.1 Static TCP Techniques

Call-Graph-Based. This technique builds a call graph for each test case to obtain a set of transitively invoked methods, called *relevant methods* [311]. The test cases with a higher number of invoked methods in the corresponding call-graphs are assigned a higher test ability and thus are prioritized first. This approach is often implemented as one of two variant two sub-strategies, the *total* strategy prioritizes the test cases with higher test abilities earlier, and the *additional* strategy prioritizes the test cases with higher test abilities while excluding the methods that have already been covered by the prioritized test cases. Further research by Mei *et al.* extends this work to measure the test abilities of the test cases according to the number of invoked statements as opposed to the number of invoked methods [206]. The main intuition behind such an extension is that by allowing for a more granular representation of test ability (at the statement level) leads to a more effective overall prioritization scheme. This call-graph based technique is classified as a white box approach, whereas the other two studied static TCP techniques are grey-box approaches, requiring only test code. We consider both types of static techniques in this paper in order to thoroughly compare them to a set of techniques that require dynamic computation of coverage.

String-Distance-Based. The key idea underlying this technique is that test cases that are textually different from one another, as measured by similarity based on string-edit

distance, should be prioritized earlier [178]. The intuition behind this idea is that textually dissimilar test cases have a higher probability of executing different paths within a program. This technique is a *grey-box* static technique since the only information it requires is the test code. There are four major variants of this technique differentiated by the string-distance metric utilized to calculate the gap between each pair of test cases: Hamming, Levenshtein, Cartesian, and Manhattan distances. Based on prior experimental results [178], Manhattan distance performs best in terms of detecting faults. Thus, in our study, we implemented the string-based TCP based on the paper by Ledru *et al.* [178], and chose Manhattan distance as the representative string distance computation for this technique. Explicit details regarding our implementation are given in Section 6.2.

Topic-Based. This static black-box technique further abstracts the concept of using test case diversity for prioritization by utilizing semantic-level topic models to represent tests of differing functionality, and gives higher prioritization to test cases that contain different topics form those already executed [266]. The intuition behind this technique is that semantic topics, which abstract test cases' functionality, can capture more information than simple textual similarity metrics, and are robust in terms of accurately differentiating between dissimilar test cases. This technique constructs a vector based on the code of each test case, including the test case's correlation values with each semantically derived topic. It calculates the distances between these text case vectors using a Manhattan distance measure, and defines the distance between one test case and a set of test cases as the minimum distance between this test case and all test cases in the set. During the prioritization process, the test case which is farthest from all other test cases is firstly selected and put into the (originally empty) prioritized set. Then, the technique iteratively add the test case farthest from the prioritized set into the prioritized set until all tests have been added.

Other Approaches. In the literature, researchers have proposed various other techniques to prioritize tests based on software requirement documents [25] or system models [171]. Recently, Saha *et al.* proposed an approach that uses software trace links

between source code changes and test code derived via Information Retrieval (IR) techniques and sorts the test cases based according to the relationships inferred via the trace links, with tests more cloesly corresponding to changes being prioritized first [251]. These techniques require additional information, such as the requirement documents, system models, and code changes, which may be unavailable or challenging to collect. In this study, we center our focus on automated TCP techniques that require only the source code and the test code of subjects, including call-graph-based, string-based and topic-based techniques.

5.1.2 Dynamic TCP Techniques

Greedy Techniques. As explained in our overview of the Call-Graph-based approach, there are typically two variants of traditional "greedy" dynamic TCP techniques, the *total* strategy and *additional* strategy, that prioritize test cases based on code coverage information. The total strategy prioritizes test cases based on their absolute code coverage, whereas the additional strategy prioritizes test cases based on each test case's contribution to the total cumulative code coverage. In our study, we implemented these techniques based on prior work by Rothermel et al. [247]. The greedy-*additional* strategy has been widely considered as one of the most effective TCP techniques in previous work [145, 306]. Recently, Zhang *et al.* proposed a novel approach to bridge the gap between the two greedy variants by unifying the strategies based on the fault detection probability [306, 124].

Given that these dynamic TCP techniques utilize code coverage information as a proxy for test effectiveness, and many different coverage metrics exist, studies have examined several of these metrics in the domain of TCP including statement coverage [247], basic block and method coverage [82], Fault-Exposing-Potential (FEP) coverage [92], transition and round-trip coverage [286]. For instance, Do *et al.* use both method and basic block coverage information to prioritize test cases [82]. Elbaum *et al.* proposed an approach

that prioritizes test cases based on their FEP and fault index coverage [92], where test cases exposing more potential faults will be assigned a higher priority. Kapfhammer *et al.* use software requirement coverage to measure the test abilities of test cases for test prioritization [163].

Adaptive Random Testing. Jiang *et al.* were the first to apply Adaptive Random Testing [58] to TCP and proposed a novel approach, called Adaptive Random Test Case Prioritization (ART) [145]. ART randomly selects a set of test cases iteratively to build a candidate set, then it selects from the candidate set the test case farthest away from the prioritized set. The whole process is repeated until all test cases have been selected. As a measure of distances between test cases, ART first calculates the distance between each pair of test cases using Jaccard distance based on their coverage, and then calculates the distance between each candidate test case and the prioritized set. Three different variants of this approach exist (*min*, *avg* and *max*), differentiated by the type of distance used to determine the similarity between one test case and the prioritized set. For example, *min* is the minimum distance between the test case and the prioritized test case. The results from Jiang et al's evaluation illustrates that ART with *min* distance performs best for TCP. Thus, in our empirical study, we implemented our ART based TCP strategy following Jiang *et al.*'s paper [145] and chose *min* distance to estimate the distance between one test case and the prioritized set.

Search-based Techniques. Search-based TCP techniques introduce meta-heuristic search algorithms into the TCP domain, exploring the state space of test case combinations to find the ranked list of test cases that detect faults more quickly [184]. Li *et al.* have proposed two variants of search-based TCP techniques, based upon hill-climbing and genetic algorithms. The hill-climbing-based technique evaluates all neighboring test cases in a given state space, locally searching the ones that can achieve largest increase in fitness. The genetic technique utilizes an evolutionary algorithm that halts evolution when a predefined termination condition is met, e.g., the fitness function value reaches a given value or a maximal number of iterations has been reached. In our empirical study,

we examine the genetic-based test prioritization approach as the representative search-based test case prioritization technique, as previous results demonstrate that genetic-based technique is more effective in detecting faults [184].

Other Approaches. Several other techniques that utilize dynamic program information have been proposed, but do not fit neatly into our classification system enumerated above [141, 267, 224]. Islam *et al.* presented an approach that reconciles information from traceability links between system requirements and test cases and dynamic information, such as execution cost and code coverage, to prioritize test cases [141]. Nguyen *et al.* have designed an approach that uses IR techniques to recover the traceability links between change descriptions and execution traces for test cases to identify the most relevant test cases for each change description [224]. Unfortunately, these TCP techniques require information beyond the test code and source code (e.g., execution cost, user knowledge, code changes) which may not be available or well maintained depending on the target software project. In this paper, we choose dynamic techniques that require only code coverage and test cases for comparison, which includes three techniques (i.e., Greedy (with total, additional strategies), ART, and Search-based). Recall that we do not aim to study the impact of coverage granularity on the effectiveness of dynamic TCPs, and opt to utilize only statement level coverage information in our experiments. This is because previous work has established that statement-level coverage is *at least as effective* as other coverage types [192, 206].

5.1.3 Empirical studies on TCP techniques

Several studies empirically evaluating TCP techniques [164, 247, 55, 273, 83, 95, 294, 259, 133, 192, 93, 92, 293, 95, 243] have been published. In this subsection we discuss the details of the studies most closely related to our own in order to illustrate the novelty of our work and research gap filled by our proposed study. Rothermel *et al.* conducted a study on unordered, random, and dynamic TCP techniques (e.g., coverage based, FEP-

based) applied to C programs, to evaluate their abilities of fault detection [247]. Elbaum *et al.* conducted a study on several dynamic TCP techniques applied to C programs in order to evaluate the impact of software evolution, program type, and code granularity on the effectiveness of TCP techniques [92]. Thomas et. al [266] compared the topic-based TCP technique to the static string-based, call-graph-based techniques as well as the greedy-additional dynamic technique at method-level on two subjects. However, this study is limited by a small set of subject programs, a comparison to only one dynamic technique at method-level only, and no investigation of fault detection similarity, the effects of software evolution or subject program size among the approaches.

Do *et al.* have presented a study of dynamic test prioritization techniques (e.g., random, optimal, coverage-based) on four Java programs with JUnit test suites. This study breaks from past studies that utilize only small C programs and demonstrates that these techniques can also be effective on Java programs. However, findings from this study also suggest that different languages and testing paradigms may lead to divergent behaviors [82]. This group also conducted an empirical study to analyze the effects of time constraints on TCP techniques [79]. Henard *et al.* recently conducted a study comparing white and black-box TCP techniques in which the effectiveness, similarity, efficiency, and performance degradation of several techniques was evaluated. While this is one of the most complete studies in terms of evaluation depth, it does not consider the static techniques considered in this paper. Thus, our study is differentiated by the unique goal of understanding the relationships between purely static and dynamic TCPs.

To summarize, while each of these studies offers valuable insights, none of them provides an in-depth evaluation and analysis of the effectiveness, efficiency, and similarity of detected faults for static TCP techniques and comparison to dynamic TCP techniques on a set of mature open source software systems. This highlights a clear research gap exists in prior work that conduct empirical studies measuring the efficacy of TCP techniques. The work conducted in this paper is meant to close this gap, and offer researchers and practitioners an extensive, rigorous evaluation of popular TCP techniques according to

extensive set of metrics and experimental investigations.

Table 5.2: The stats of the subject programs: Size: #Loc; TM: #test cases at method level; TC: #test cases at class level; All: #all mutation faults; Detected: #faults can be detected by test cases.

Subject Programs	Size	Tests		Mutants	
		#TM	#TC	Detected	All
P1-geojson-jackson	1,151	44	13	301	717
P2-statsd-jvm-profiler	1,355	29	12	290	708
P3-stateless4j	1,756	61	10	392	696
P4-jarchivelib	1,940	22	12	655	948
P5-JSONassert	1,957	121	10	935	1,116
P6-java-faker	2,069	28	11	392	600
P7-jackson-datatype-joda	2,409	57	8	675	1,212
P8-Java-apns	3,234	87	15	412	1,122
P9-pusher-websocket-java	3,259	199	11	851	1,470
P10-gson-fire	3,421	55	14	847	1,064
P11-jackson-datatype-guava	3,994	91	15	313	1,832
P12-dictomaton	4,099	53	11	2,024	10,857
P13-jackson-uuid-generator	4,158	45	6	802	2,039
P14-JAdventure	4,416	35	10	738	5,098
P15-exp4j	4,617	285	9	1,365	1,563
P16-jumblr	4,623	103	15	610	1,192
P17-efflux	4,940	41	10	1,190	2,840
P18-metrics-core	5,027	144	28	1,656	5,265
P19-low-gc-membuffers	5,198	51	18	1,861	3,654
P20-xembly	5,319	58	16	1,190	2,546
P21-scribe-java	5,355	99	18	563	1,622
P22-jpush-api-java-client	5,462	65	10	822	2,961
P23-gdx-artemis	6,043	31	20	968	1,687
P24-protoparser	6,074	171	14	3,346	4,640
P25-commons-cli	6,601	317	26	2,362	2,801
P26-mp3agic	6,939	205	19	3,362	6,391
P27-webbit	7,363	131	25	1,268	3,833
P28-RestFixture	7,421	268	30	2,234	3,278
P29-LastCalc	7,707	34	13	2,814	6,635
P30-jackson-dataformat-csv	7,850	98	27	1,693	6,795
P31-skype-java-api	8,264	24	16	885	6,494
P32-lambdaj	8,510	252	35	3,382	4,341
P33-jackson-dataformat-xml	8,648	134	45	1,706	4,149
P34-jopt-simple	8,778	511	79	2,325	2,525
P35-jline2	8,783	130	16	3,523	8,368
P36-javapoet	9,007	246	16	3,400	4,601
P37-Liqp	9,139	235	58	7,962	18,608
P38-cassandra-reaper	9,896	40	12	1,186	5,105
P39-JSqlParser	10,335	313	19	15,698	32,785
P40-raml-java-parser	11,126	190	36	4,678	6,431
P41-redline-smalltalk	11,228	37	9	1,834	10,763
P42-user-agent-utils	11,456	62	7	376	688
P43-javaewah	13,293	229	11	6,307	11,939
P44-jsoup-learning	13,505	380	25	7,761	13,230
P45-wsc	13,652	16	8	1,687	17,942
P46-rome	13,874	443	45	4,920	10,744
P47-JActor	14,171	54	43	132	1,375
P48-RoaringBitmap	16,341	286	15	9,709	13,574
P49-JavaFastPFOR	17,695	42	8	46,429	64,372
P50-jprotobuf	21,161	48	18	1,539	10,338
P51-worldguard	24,457	148	12	1,127	25,940
P52-commons-jxpath	24,910	411	39	13,611	24,369
P53-commons-io	27,263	1125	92	7,630	10,365
P54-nodebox	32,244	293	40	7,824	36,793
P55-asterisk-java	39,542	220	39	3,299	17,664
P56-ews-java-api	46,863	130	28	2,419	31,569
P57-commons-lang	61,518	2388	114	25,775	32,291
P58-joda-time	82,998	4,026	122	20,957	28,382
Total	714,414	15,441	1,463	245,012	542,927

5.1.4 Mutation Analysis

Fault detection effectiveness is almost universally accepted as the measurement by which to evaluate TCP approaches [19, 158, 192]. However, extracting a suitable set of representative real-world faults is typically prohibitively costly. Thus, researchers and developers commonly evaluate the effectiveness of TCP approaches using mutation analysis, in

which a set of program variants, called mutants, are generated by seeding a large number of small syntactic errors into a seemingly "correct" version of a program. For a given subject program, mutation operators are utilized to seed these faults (known as *mutants*) into an unmodified version of the program. It is said that a mutant is *killed* by a test case when this test case is able to detect a difference between the unmodified program and the mutant. In the context of TCP research, mutation analysis is applied to a subject program to generate a large set of mutants, each containing a minor fault, and then this set is used to evaluate the effectiveness of a set of prioritized test cases.

Preliminary studies have shown mutants to be suitable for simulating real bugs in software testing experiments in controlled contexts [20, 158], and mutation analysis has been used to evaluate many different types of testing approaches, including TCP techniques [92, 266, 133, 192, 272]. For example, Henard *et al.* utilized mutation analysis to compare white-box and black-box TCP techniques [133]. Lu *et al.* evaluated the test case prioritization techniques in the context of evolving software systems using mutation analysis [192]. Finally, Walcott *et al.* proposed a time-aware test prioritization technique and evaluated their approach using mutants [272].

Additionally, recent research has been undertaken that aims to understand the relationship between different *types* of mutants (e.g., operators) and whether or not they are a suitable proxy for *real* faults [170, 17, 159, 161]. Ammann *et al.* proposed a framework to reduce redundant mutants and determine a minimal set of mutants for properly evaluating test cases [17]. Kintis *et al.* introduced several alternatives to mutation testing strategies to establish whether they adversely affect measuring test effectiveness [170]. However, pervious studies do not provide comments on the following in the context of TCP: 1) none of these studies has investigated the impact of the quantity of mutants utilized in TCP experiments; and 2) previous work has not examined the impact of mutants seeded according to different operators on the effectiveness of TCP approaches. It is quite possible that TCP may perform differently when detecting different quantities or types of mutants, particularly across software projects. Addressing these current short-

comings of past studies would allow for the verification or refutation of previous widely used experimental settings for mutation-based TCP evaluations. Thus, we aim to evaluate the effectiveness of TCP techniques in terms of detecting different quantities and types of mutants in order to understand their impact on this quality metric.

5.1.5 Metrics for TCP techniques

The Average Percentage of Faults Detected (APFD) metric is a well-accepted metric in the TCP domain [247, 311, 248, 81, 86, 92, 89], which is used to measure the effectiveness, in terms of fault detection rate, for each studied test prioritization technique. Formally speaking, let T be a test suite and T' be a permutation of T, the APFD metric for T' is computed according to the following metric:

$$APFD = 1 - \frac{\sum_{i=1}^{m} TF_i}{n * m} + \frac{1}{2n} \tag{5.1}$$

where n is the number of test cases in T, m is the number of faults, and TF_i is the position of the first test case in T' that detects fault i.

Although APFD has been widely used for evaluating TCP techniques, it assumes that each test incur the same time cost, an assumption which often doesn't hold up in practice. Thus, Elbaum *et al.* introduced another metric, called APFDc [90]. APFDc is the cost-cognizant version of APFD, which considers both the test case execution cost and fault severity. While not as widely used as APFD, APFDc has also been used to evaluate TCP approaches, resulting in a more detailed evaluation. [96]. APFDc can be formally defined as follows: let $t_1, t_2, ..., t_n$ be the execution costs for all the n test cases. and $f_1, f_2, ..., f_m$ be the severities of the m detected faults. The APFDc metric is calculated according to the following equation:

$$APFDc = \frac{\sum_{i=1}^{m} f_i * (\sum_{j=TF_i}^{n} t_j - \frac{1}{2} t_{TF_i})}{\sum_{i=1}^{n} t_i * \sum_{i=1}^{m} f_i} \tag{5.2}$$

Similar to Equation 6.1, n is the number of test cases in T, m is the number of faults, and TF_i is the position of the first test case in T' that detects fault i. In our empirical study, we evaluate the performance of TCP techniques based on both of APFD and APFDc, in order to provide a complete picture of the performance of TCP techniques from the perspective of both effectiveness and efficiency. Additionally, we further examine the relationship between these two metrics and the resultant implications for the domain of TCP research.

5.2 Empirical Study

In this section, we state our research questions, and enumerate the subject programs, test suites, study design, and implementation of studied techniques in detail.

5.2.1 Research Questions (RQs):

Our empirical study addresses the following RQs:

RQ$_1$ How do static TCP techniques compare with each other and with dynamic techniques in terms of *effectiveness* measured by APFD?

RQ$_2$ How do static TCP techniques compare with each other and with dynamic techniques in terms of *effectiveness* measured by APFDc?

RQ$_3$ How does the test granularity impact the effectiveness of both the static and dynamic TCP techniques?

RQ$_4$ How does the program size (i.e., LOC) impact the effectiveness of both the static and dynamic TCP techniques?

RQ$_5$ How do static and dynamic TCP techniques perform as software evolves?

RQ$_6$ How does the quantity of mutants impact the effectiveness of the studied TCP techniques?

RQ$_7$ How does mutant type impact the effectiveness of the studied TCP techniques?

RQ$_8$ How *similar* are different TCP techniques in terms of detected faults?

RQ$_9$ How does the *efficiency* of static techniques compare with one another in terms of execution time cost?

To aid in answering **RQ**$_1$ and **RQ**$_2$, we introduce the following null and alternative hypotheses. The hypotheses are evaluated at the 0.05 level of significance:

H_0**:** There is no statistically significant difference in the effectiveness between the studied TCPs.

H_a**:** There is a statistically significant difference in the effectiveness between the studied TCPs.

5.2.2 Subject Programs, Test Suites and Faults

We conduct our study on 58 real-world Java programs from GitHub[6]. The program names and sizes in terms of lines of code (LOC) are shown in Table 5.2, where the sizes of subjects vary from 1,151 to 82,998 LoC. Our subjects are larger in size and quantity than previous work in the TCP domain [192, 133, 266, 178, 145]. Our methodology for collecting these subject programs is as follows. We first collect a set of 399 Java programs from GitHub that contain integrated JUnit test cases and can be compiled successfully. Then, we discarded programs which were relatively small in size (i.e., LOC is less than 1,000), or that had very small numbers of test cases (i.e., less than 15 test cases at method level and five test cases at class level). Finally, we ran a set of tools to collect both the static and dynamic information (Section 3.4) and discarded programs for which the tools were not applicable. After this process we obtained our set of 58 subject programs.

To perform this study, we checked out the most current master branch of each program, and provide the version IDs in my online appendix [11]. For each program, we used the original JUnit test suites for the corresponding program version. Since one of

Table 5.3: Muation Operators Used

ID	Mutation Operator
M0	Conditional Boundary Mutator
M1	Constructor Call Mutator
M2	Increments Mutator
M3	Inline Constant Mutator
M4	Invert Negs Mutator
M5	Math Mutator
M6	Negate Conditionals Mutator
M7	Non-Void Method Call Mutator
M8	Remove Conditional Mutator
M9	Return Vals Mutator
M10	Void Method Call Mutator
M11	Remove Increments Mutator
M12	Member Variable Mutator
M13	Switch Mutator
M14	Argument Propagation Mutator

the goals of this study is to understand the impact of test granularity on the effectiveness of TCP techniques, we introduce two groups of experiments in our empirical study based on two test-case granularities: (i) the test-method and (ii) the test-class granularity. The numbers of test cases on test-method level and test-class level are shown in Columns 3 & 4 of Table 5.2 respectively.

One goal of this empirical study is to compare the effectiveness of different test prioritization techniques by evaluating their fault detection capabilities. Thus, each technique will be evaluated on a set of program faults introduced using mutation analysis. As mutation analysis has been widely used in regression test prioritization evaluations [306, 80, 192, 310] and has been shown to be suitable in simulating real program faults [20, 158], this is a sensible method of introducing program defects. We applied all the 15 available mutation operators from the PIT [239] mutation tool (Version 1.1.7) to generate mutation faults for each project. All mutation operators are listed in Table 5.3 and their detailed definitions can be found on the PIT website [232] and on my online appendix [11]. We utilized PIT to determine the set of faults that can be detected by the test suites for each of our subject programs. When running the subject program's JUnit test suite via the PIT

Maven plugin, test cases are automatically executed against each mutant, PIT records the corresponding test cases capable of killing each mutant. By analyzing the PIT reports, we obtained the information (e.g., fault locations) for each mutation fault and all the test cases that can detect it. Note that the typical implementation of PIT stops executing any remaining tests against a mutant once the mutant is killed by some earlier test to save time. However, for the purpose of obtaining a set of ”killable” mutants, this is undesirable. Thus, we modified PIT to force it to execute the remaining tests against a mutant even when the mutant has been killed. Since not all produced mutation faults can be detected/covered by test cases, only mutants that can be detected by at least one test case are included in our study. The numbers of detected mutation faults and the numbers of all mutation faults are shown in Columns 5 and 6 of Table 5.2 respectively. As the table shows, the numbers of detected mutants range from 132 to 46,429. There are of course certain threats to validity introduced by such an analysis, namely the the potential bias introduced by the presence of equivalent and trivial mutants [22, 17]. We summarize the steps we take in our methodology to mitigate this threat in Section 6.4.

5.2.3 Design of the Empirical Study

As discussed previously (Section 5.1), we limit the focus of this study to TCP techniques that do not require additional inputs, such as code changes or software requirements that may require extra effort or time to collect or may be unavailable. We select two white-box and two black-box static techniques, and four white-box dynamic techniques with statement-level coverage as the subject techniques for this study, which are listed in Table 5.4. We sample from both white and black box approaches as the major goal of this study is to examine the effectiveness and trade-offs of static and dynamic TCPs under the assumption that both the source code of the subject application, as well as the test cases are available. It is worth noting that our evaluation employs *two* versions of the static topic model-based technique, as when contacting the authors of [266], they suggested that an implementation using the Mallet [202] tool would yield better results than their

initial implementation in R [266]. There are various potential coverage granularities for dynamic techniques, such as statement-level, method-level and class-level. Previous research showed that statement-level TCP techniques perform the best [206, 124]. Thus, in our study, we choose statement-level coverage for the dynamic TCP techniques. We now describe the experimental procedure utilized to answer each RQ posed above.

Table 5.4: Studied TCP Techniques

Type	Tag	Description
Static	TP_{cg-tot}	Call-graph-based (total strategy)
	TP_{cg-add}	Call-graph-based (additional strategy)
	TP_{str}	The string-distance-based
	$TP_{topic-r}$	Topic-model-based using R-lda package
	$TP_{topic-m}$	Topic-model-based using Mallet
Dynamic	TP_{total}	Greedy total (statement-level)
	TP_{add}	Greedy additional (statement-level)
	TP_{art}	Adaptive random (statement-level)
	TP_{search}	Search-based (statement-level)

RQ$_1$**:** The *goal* of **RQ**$_1$ is to compare the effectiveness of different TCP techniques, by evaluating their fault detection capabilities. Following existing work [306, 192], we fixed the number of faults for each subject program. That is, we randomly chose 500 different mutation faults and partitioned the set of all faults into groups of five (e.g., a mutant group) to simulate each faulty program version. Thus, 100 different faulty versions (i.e., 500/5 = 100) were generated for each program. If a program has less than 500 mutation faults, we use all detected mutation faults for this program and separate these faults into different groups (five faults per group). For the static techniques, we simply applied the techniques as described in Sections 5.1 & 5.2.4 to the test and source code of each program to obtain the list of prioritized test cases for each mutant group. For the dynamic techniques, we obtained the coverage information of the test-cases for each program. We then used this coverage information to implement the dynamic approaches as described in Sections 5.1 & 5.2.4. Then we are able to collect the fault detection information for each program according to the fault locations.

To measure the effectiveness in terms of rate of fault detection for each studied test prioritization technique, we utilize the well-accepted Average Percentage of Faults Detected (APFD) metric in TCP domain [247, 311, 248, 81, 86, 92, 89]. Recall that every

subject program has 100 mutant groups (five mutations per group). Thus, we created 100 faulty versions for each subject (each version contains five mutations) and ran all studied techniques over these 100 faulty versions. That is, running each technique 100 times for each subject. Then, we performed statistical analysis based on the APFD results of these 100 versions. To test whether there is a statistically significant difference between the effectiveness of different techniques, we first performed an one-way ANOVA analysis on the mean APFD values for all subjects and a Tukey HSD test [265], following the evaluation procedures utilized in related work [206, 192]. The ANOVA test illustrates whether there is a statistically significant variance between all studied techniques and the Tukey HSD test further distinguishes techniques that are significantly different from each other, as it classifies them into different groups based on their mean APFD values [265]. These statistical tests give a statistically relevant overview of whether the mean APFD values for the subject programs differ significantly. Additionally, we performed a Wilcoxon signed-rank test between each pair of TCP techniques for their average APFD value across all subject techniques, to further illustrate the relationship between individual subject programs. We choose to include this non-parametric test since we cannot make assumptions about wether or not the data under consideration is normally distributed.

RQ$_2$: Although APFD has been widely used for TCP evaluation, it assumes that each test takes the same amount of time, which may not be always accurate in practice. The *goal* of this **RQ** is to examine the effectiveness of TCP techniques in terms of the APFDc metric, which considers both the execution time and severities of detected faults. We also compare the results of the APFDc with those of the APFD for understanding the performance of different types of metrics in the TCP area. However, there is no clearly-defined way to estimate the severities for the detected faults, and no widely-used tool to collect this information, making it hard to measure fault severity. Therefore, following previous work [96], we consider all faults to share the same severity level. Thus, in the

context of our empirical study, APFDc reduces to the following equation:

$$APFDc = \frac{\sum_{i=1}^{m} \sum_{j=TF_i}^{n} t_j - \frac{1}{2} t_{TF_i}}{\sum_{i=1}^{n} t_i * m} \tag{5.3}$$

where n is the number of test cases in T, m is the number of faults, TF_i is the position of the first test case in T' that detects fault, and i, $t_1, t_2, ..., t_n$ are the execution costs for all the n test cases. To measure test execution costs, we use the Maven Surefire Plugin to trace the start and end events of each test to record the corresponding execution time. Similar as **RQ**$_1$, we performed both of an one-way ANOVA analysis on the mean AFPDc values for all subjects and a Tukey HSD test to further understand the whether there is a statistically significant variance between the performance of the studied techniques in terms of APFDc values. In addition, we further examined the relationship between the two metrics, AFPD and APFDc, to understand the differences in effectiveness of TCP techniques. We utilize the Kendall rank correlation coefficient τ [253] to compare the results of these two metrics. Kendall rank correlation coefficient τ is commonly used to examine the relationship between two ordering quantities (i.e., observations of two variables). The coefficient ranges in value from -1 to 1, with values closer to 1 indicating similarity and values closer to -1 indicating dissimilarity. When the value is close to 0, these two quantities are considered independent. For example, in the context of our study, we have two quantities, APFD and APFDc values. Thus, in the context of our study, if the values of APFD values across all TCP techniques are similar to APFDc values, the Kendall tau rank coefficient τ would be closer to 1. Otherwise, it would be closer to -1. Since there is no guarantee that the relationship between APFD and APFDc values are linear, we chose Kendall τ_b coefficient in our study, following prior work [107]:

$$\tau_b = \frac{n_c - n_d}{\sqrt{(n(n-1)/2 - \sum_i t_i(t_i - 1)/2)(n(n-1)/2 - \sum_j u_j(u_j - 1)/2)}} \tag{5.4}$$

where n_c refers to the number of concordant pairs, n_d refers to the number of discordant pairs, t_i refers to the number of tied values in ith tie group for the first quantity, and u_j

refers to the number of tied values in jth tie group for the second quantity.

RQ$_3$ The *goal* of this **RQ** is to analyze the impact of different test granularities on the effectiveness of TCP techniques. Thus, we choose two granularities: test-method and test-class levels. The test-method level treats each JUnit test method as a test case, while test-class level treats each JUnit test class as a test case. We examine both the effectiveness and similarity of detected faults for both granularities.

RQ$_4$ The *goal* of this **RQ** is to investigate the impact of different program sizes on the effectiveness of TCP techniques. Thus, we measure the size for each subject program in terms of its Lines of Code (LOC). To examine whether TCP technqiues tend to perform differently on programs of different sizes we classify the programs into two groups, a set of *smaller* programs and a set of *larger* programs. These two groups were created by ordering our subject programs in increasing order of LOC and splitting the ordered list in the middle. This results in two groups of 29 subject programs, the first group containing smaller programs and the second group containing larger programs.

RQ$_5$**:** The *goal* of this **RQ** is to understand the effectiveness of TCP techniques in a software evolution scenario. To accomplish this we apply different TCP techniques across different versions of each subject program. More specifically, tests are prioritized using the information from a given previous program version, and the prioritized set of test cases is then applied to faulty variants of the most recent program version. The faulty variants are created using the same methodology described for RQ$_1$. This methodology closely follows that of previous work [192] and allows us to investigate if the performance of TCP techniques remains stable, decreases, or increases as software evolves. In our study, we collect different versions for each subject program exactly following the methodology proposed in [192]. For each subject, we start from the most current version and collect one version per ten commits moving backward through the commit history. We then discard those programs that did not successfully compile and those that are not applicable to our tools. Due to the extremely large volume of data and the time cost of running these experiments, we randomly chose 12 subject programs to investigate this research

question. Note that the numbers of versions (i.e., 66) and subject programs (i.e., 12) used in this work are larger than all prior TCP work considering evolutionary scenarios (e.g., the recent work by Lu et.al. [192] used 53 versions of 8 real-world programs).

RQ$_6$**:** The *goal* of this **RQ** is to examine the impact of mutant quantity on the effectiveness for TCP techniques in terms of APFD and APFDc values. In our default experimental setting, we have 100 groups of mutation faults, and each group contains five mutants following prior work [206, 306, 124, 192]. However, in practice, the number of faults within a buggy version can be more than or less than five. Therefore, to better understand the impact of fault quantity per group, we generate different number of faults (i.e., 1 to 10) within each of the 100 constructed fault groups for each subject program. Note that we may have less than 100 fault groups when the number of mutants are small for some subjects. That is, we repeat all our prior experiments 10 times, each time recording the APFD and APFDc values for all studied techniques under 100 fault groups with a different number of faults (from 1 to 10). Finally, we perform Kendall rank τ_b coefficient analysis to understand the relationship between the results for the mutation groups with different sizes and the results of the default setting (i.e., with 5 faults within each group). That is, we perform Kendall analysis to compare each fault-quantity setting (i.e., 100 groups mutation faults and the size of each fault group varies from 1 to 10) to the mutation faults with the default setting. Intuitively, if the values of Kendall τ_b coefficient are close to 1, the TCP techniques perform similarly between fault groups of varying sizes and fault groups with the default size, implying that the quantity of mutation faults does not impact TCP evaluation.

RQ$_7$**:** The *goal* of this **RQ** is to understand the impact of the mutant types (i.e., those mutants generated with different operators) on the effectiveness for TCP techniques in terms of APFD and APFDc values. Intuitively, we first classified mutants into different groups based on their corresponding operators. That is, the mutation faults generated by the same operators would be classified into the same group. In our empirical study, we utilized all 15 built-in mutation operators in PIT. Thus, we have 15 types of mutation faults

for each subject program. We evaluate TCP techniques across these 15 types of mutation faults with the default setting, where for each operator we randomly choose 500 mutants and separate them into 100 groups (each group contains 5 mutation faults). Note that we may have less than 100 fault groups when the number of mutants are small for some mutant types. Then, TCP techniques are evaluated based on these groups of mutation faults. Finally, we compare the results for different types of mutation faults with our default fault seeding (i.e., randomly including different types of faults) under the same default setting (i.e., 100 mutated groups and each group contains 5 mutation faults). Similar as **RQ**$_4$, we chose Kendall rank tau coefficient to measure the relationship between them to check if the type of mutation fault impacts TCP evaluation.

RQ$_8$**:** The *goal* of this **RQ** is to analyze the similarity of detected faults for different techniques to better understand the level of equivalency of differing strategies. It is clear that this type of analysis is important, as while popular metrics such as APFD measure the *effectiveness* between two different techniques, this does not reveal the similarity of the test cases in terms of uncovered faults. For instance, let us consider two TCP techniques A and B. If technique A achieves an APFD of ≈ 60% and technique B achieves an APFD of ≈ 20%, while this gives a measure of relative effectiveness, the APFD does not reveal how similar or orthogonal the techniques are in terms of the faults detected. For instance, all of the faults uncovered by top ten test cases from technique B could be different than those discovered by top ten test cases from technique A, suggesting that the techniques may be *complimentary*. To evaluate the similarity between different TCP techniques, we utilize and build upon similarity analysis used in recent work [133, 132] and construct binary vector representations of detected faults for each technique and then calculate the distance between these vectors as a similarity measure.

We employ two methodologies in order to give a comprehensive view of the similarity of the studied TCPs. At the core of both of these techniques is a measure of similarity using the Jaccard distance to determine the distance between vectorized binary representations of detected faults (where a 1 signifies a found fault and a 0 signifies an undis-

covered fault) for different techniques across individual or groups of subject programs. We use the following definition [133]:

$$J(T_A^i, T_B^i) = \frac{\mid T_A^i \cap T_B^i \mid}{\mid T_A^i \cup T_B^i \mid} \tag{5.5}$$

where T_A^i represents the binary vectorized discovered faults of some studied technique A after the execution of the ith test case in the techniques prioritized set, and T_B^i represents the same meaning for some studied technique B and $0 \leq J(T_A^i, T_B^i) \leq 1$. While we use the same similarity metric as in [133], we report two types of results: 1) results comparing the similarity of the studied static and dynamic techniques using the average Jaccard coefficient across all subjects at different test-case granularities, and 2) results comparing each technique in a pair-wise manner for each subject program. For the second type of analysis, we examine each possible pair of techniques and rank each subject program according to Jaccard coefficient as highly similar (1.0 - 0.75), similar (0.749 - 0.5), dissimilar (0.49 - 0.25), or highly dissimilar (0.249-0). This gives a more informative view of how similar two techniques might be for different subject programs. To construct both types of binary fault vectors, we use the same fault selection methodology used to calculate the APFD, that is, we randomly sample 500 faults from the set of known discoverable faults for each subject.

In addition, we also want to understand whether the studied TCP techniques' most highly prioritized test cases uncover comparatively different numbers of mutants generated by different operators. Thus, for different cut points, particularly the top cut points (e.g. 10%), we examine the both the total number and relative percentages of different types of mutants detected by each TCP technique to better understand the types of mutants which are easily detected by most highly prioritized test cases for different TCP techniques.

RQ$_9$: The final *goal* of our study is to understand the efficiency of static techniques, in terms of execution costs. Note that, we only focus on the efficiency of static techniques, since dynamic techniques are typically run on the previous version of a program to collect

Table 5.5: Results for the ANOVA and Tukey HSD tests on the average APFD and APFDc values at test-class level, which are depicted in Figure 5.1. The last column shows the results for Kendall tau Rank Correlation Coefficient τ_b between the average APFDc and average APFD.

Metrics	Analysis	TP$_{cg-tot}$	TP$_{cg-add}$	TP$_{str}$	TP$_{topic-r}$	TP$_{topic-m}$	TP$_{total}$	TP$_{add}$	TP$_{art}$	TP$_{search}$	p-value	τ_b
APFD	Avg	0.778	0.790	0.777	0.675	0.745	0.738	0.769	0.633	0.765	1.777e-18	0.722
	HSD	A	A	A	B	A	A	A	B	A		
APFDc	Avg	0.652	0.679	0.667	0.574	0.657	0.614	0.650	0.612	0.649	0.154	
	HSD	A	A	A	A	A	A	A	A	A		

Table 5.6: Results for the ANOVA, and Tukey HSD tests on the average APFD and APFDc values at test-method level, which are depicted in Figure 5.2. The last column shows the results for Kendall tau Rank Correlation Coefficient τ_b between the average APFDc and average APFD.

Metrics	Analysis	TP$_{cg-tot}$	TP$_{cg-add}$	TP$_{str}$	TP$_{topic-r}$	TP$_{topic-m}$	TP$_{total}$	TP$_{add}$	TP$_{art}$	TP$_{search}$	p-value	τ_b
APFD	Avg	0.764	0.818	0.813	0.781	0.817	0.809	0.898	0.798	0.885	2.568e-28	0.556
	HSD	C	B	B	BC	B	B	A	BC	A		
APFDc	Avg	0.638	0.737	0.671	0.678	0.679	0.633	0.708	0.669	0.735	0.053	
	HSD	A	A	A	A	A	A	A	A	A		

coverage information, and thus the temporal overhead is quite high and well-studied. To evaluate the efficiency of static techniques, we collect two types of time information: the time for pre-processing and the time for prioritization. The time for pre-processing contains different phases for different techniques. For example, TP$_{cg-tot}$ and TP$_{cg-add}$ need to build the call graphs for each test case. TP$_{str}$ needs to analyze the source code to extract identifiers and comments for each test case. Besides, TP$_{topic}$ needs to pre-process extracted textual information and use the R-LDA package and Mallet [202] to build topic models. The time for prioritization refers to the time cost for TCP on different subjects.

5.2.4 Tools and Experimental Hardware

We reimplemented all of the studied dynamic and static TCPs in Java according to the specifications and descriptions in their corresponding papers, since the implementations were not available from the original authors and had to be adapted to our subjects. Three of the authors carefully reviewed and tested the code to make sure the reimplementation is reliable.

TP$_{cg-tot}$/TP$_{cg-add}$: Following the paper by Zhang *et al.* [311], we use the *IBM T. J. Watson Libraries for Analysis* (WALA) [270] to collect the RTA static call graph for each test, and traverse the call graphs to obtain a set of relevant methods for each test case. Then, we

implement two greedy strategies (i.e., total and additional) to prioritize test cases.

TP$_{str}$: Based on the paper by Ledru *et al.* [178], each test case is treated as one string without any preprocessing. Thus, we directly use JDT [143] to collect the textual test information for each JUnit test, and then calculate the Manhattan distances between test cases to select the one that is farthest from the prioritized test cases.

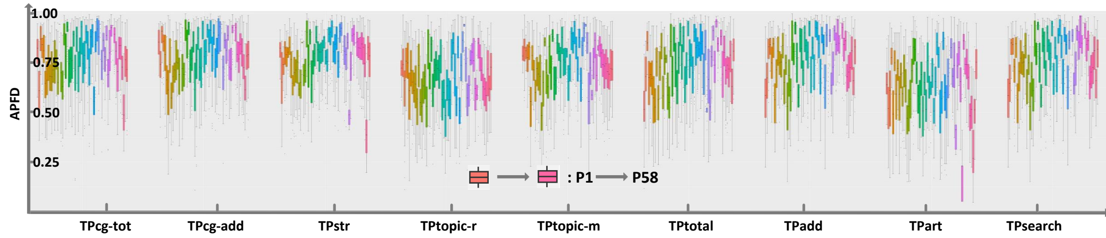
TP$_{topic-r}$ and TP$_{topic-m}$: Following the topic-based TCP paper [266], we first use JDT to extract identifiers and comments from each JUnit test, and then pre-process those (e.g., splitting, removing stop words, and stemming). To build topic models, we used the R-LDA package [177] for TP$_{topic-r}$ and Mallet [202] for TP$_{topic-m}$. All parameters are set with previously used values [266, 60]. Finally, we calculated the Manhattan distances between test cases, and selected the ones that are farthest from the prioritized test cases.

Dynamic TCP techniques: We use the ASM bytecode manipulation and analysis toolset [34] to collect the coverage information for each test. Specifically, in our empirical study, it obtains a set of statements that can be executed by each test method or test class. The greedy techniques are replicated based on the paper by Rothermel *et al.* [247]. For the ART and search-based techniques, we follow the methodology described in their respective papers [145, 184].

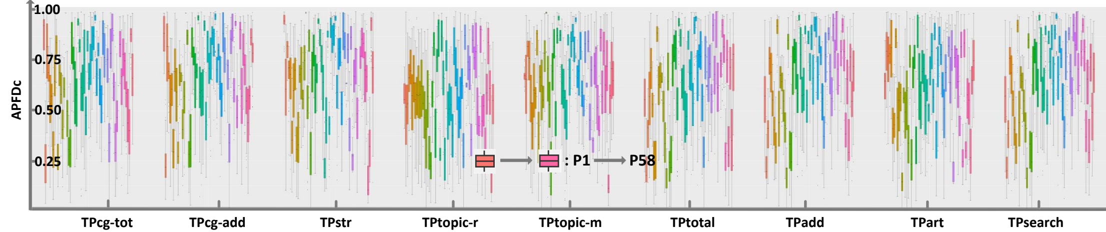
Experimental Hardware: The experiments were carried out on Thinkpad X1 laptop with Intel Core i5-4200 2.30 GHz processor and 8 GB DDR3 RAM and eight servers with 16, 3.3 GHz Intel(R) Xeon(R) E5-4627 CPUs, and 512 GB RAM, and one server with eight Intel X5672 CPUs and 192 GB RAM. All the execution time information (i.e., both of the execution time to run TCP techniques and the execution time for each test case) was collected on the laptop to ensure that the analysis for time costs is consistent.

5.3 Results

In this section, we outline the experimental results to answer the **RQs** listed in Section 6.2.



(a) The values of APFD on test-class level across all subject programs.



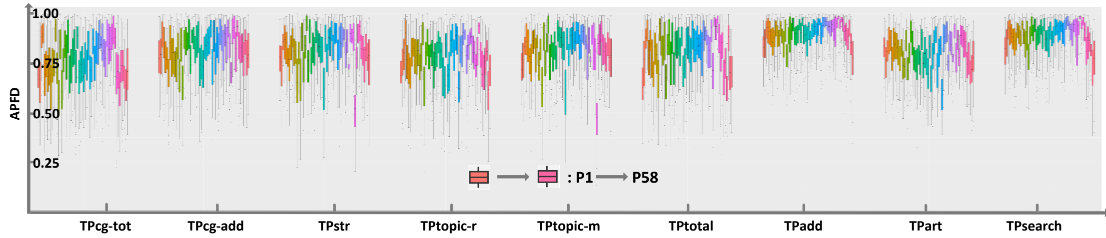
(b) The values of APFDc on test-class level across all subject programs.

Figure 5.1: The box-and-whisker plots represent the values of APFD and APFDc for different TCP techniques at test-class level. The x-axis represents the APFD and APFDc values. The y-axis represents the different techniques. The central box of each plot represents the values from the lower to upper quartile (i.e., 25 to 75 percentile).

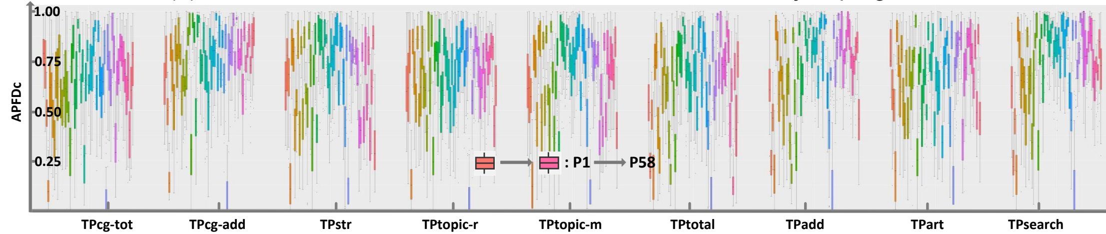
5.3.1 RQ$_1$ & RQ$_2$ & RQ$_3$: Effectiveness of Studied Techniques Measured by APFD and APFDc at Different Granularities

5.3.1.1 Results at Test Class Level

The values of APFD across all subjects at the test class level are shown in Figure 5.1(a) and Table 5.5. Based on the results, we observe that, somewhat surprisingly at the test-class level, the static TP$_{cg-add}$ technique performs the best across all studied TCP techniques (including all dynamic techniques) with an average APFD value of 0.790 (see Table 5.5). Among the static techniques, TP$_{cg-add}$ performs best, followed by TP$_{cg-tot}$, TP$_{str}$, TP$_{topic-m}$ and TP$_{topic-r}$. The best performing dynamic technique at class-level is TP$_{add}$ followed by TP$_{search}$, TP$_{total}$, and TP$_{art}$. It is notable that at test-class level granularity, the most effective static technique TP$_{cg-add}$ performs even better than the most effective dynamic technique TP$_{add}$ in terms of APFD, i.e., 0.790 versus 0.769. The experimental results on APFDc values further confirm the above finding. Shown in Figure 5.1(b)



(a) The values of APFD on test-method level across all subject programs.

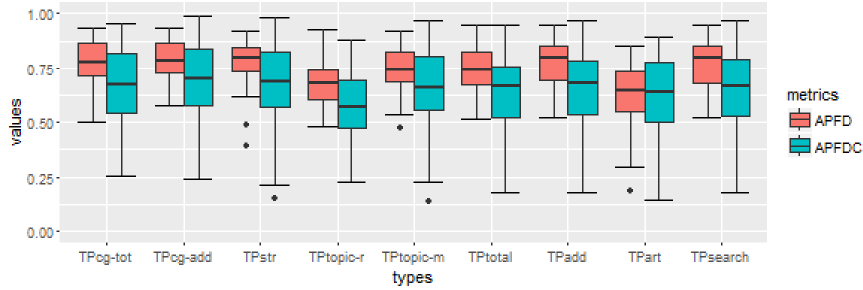


(b) The values of APFDc on test-method level across all subject programs.

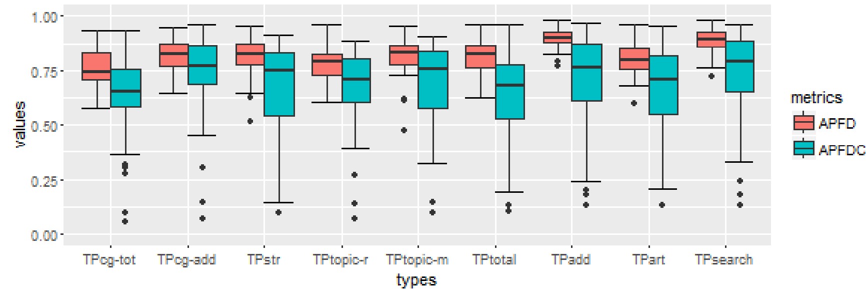
Figure 5.2: The box-and-whisker plots represent the values of APFD and APFDc for different TCP techniques at test-method level. The x-axis represents the APFD and APFDc values. The y-axis represents the different techniques. The central box of each plot represents the values from the lower to upper quartile (i.e., 25 to 75 percentile).

and Table 5.5, the static TP$_{cg-add}$ technique outperforms all the studied TCP techniques with an average APFDc value of 0.679, whereas even the most effective dynamic TP$_{add}$ only achieves an average APFDc value of 0.650. Furthermore, the Kendall τ_b Rank Correlation value of 0.722 also demonstrates that APFDc values are generally consistent with APFD values at the test class level. Therefore, at the test-class level, the call-graph based strategies can even outperform dynamic-coverage based strategies, which is notable. Additionally, overall the static techniques outperform the dynamic techniques at the test-class level. One potential reason for this is that many program statements are covered several times by tests at the test-class level, making the traditional dynamic techniques less precise, since they do not consider the number of times that a statement is covered.

While Figure 5.1 shows the detailed APFD and APFDc values for each studied subject at test-class level, Figure 5.3(a) further shows the ranges of APFD and APFDc values across all subjects at test-class level, reflecting the robustness of the studied approaches



(a) The values of APFD and APFDc for different TCP techniques across all subject programs on test-class level.



(b) The values of APFD and APFDc for different TCP techniques across all subject programs on test-method level.

Figure 5.3: The box-and-whisker plots represent the values of APFDc for different TCP techniques at different test granularities. The x-axis represents the APFDc values. The y-axis represents the different techniques. The central box of each plot represents the values from the lower to upper quartile (i.e., 25 to 75 percentile).

across both metrics. For APFD, the range of average values across all subjects at test-class level for TP$_{add}$ is the smallest (i.e.,0.523-0.947), implying that the performance of TP$_{add}$ is usually stable despite differing subjects for this metric. Conversely, the ranges of APFD values for TP$_{str}$ and TP$_{art}$ are much larger (0.391-0.917 for TP$_{str}$, 0.187-0.852 for TP$_{art}$), implying that their performance varies across different types of subjects. However, we observe different trends for the APFDc metric. The ranges of APFDc values are all much larger than those of APFD values. This is most likely due to the fact that APFDc considers execution times, which we found to be randomly distributed, resulting in a larger variation in results across different subjects.

To further investigate the finding that static techniques tend to have a higher variance in terms of effectiveness depending on the program type, we investigated further by in-

Table 5.7: The results of Wilcoxon signed rank test on the average APFD values for each pair of TCP techniques. The techniques T1 to T9 refer to TP$_{cg-tot}$, TP$_{cg-add}$, TP$_{str}$, TP$_{topic-r}$, TP$_{topic-m}$, TP$_{total}$, TP$_{add}$, TP$_{art}$, TP$_{search}$ respectively. For each pair of TCP techniques, there are two sub-cells. The first one refers to the p-value at test-class level and the second one refers to the p-value at test-method level. The p-values are classfied into three categories, 1) p$>$0.05, 2) 0.01$<$p$<$0.05, 3) p$<$0.01. The p-values for categories p$>$0.05 and p$<$0.01 are presented as p$>$0.05 and p$<$0.01 respectively. If a p-value is less than 0.05, the corresponding cell is shaded.

	T2		T3		T4		T5		T6		T7		T8		T9	
T1	0.02	<0.01	>0.05	<0.01	<0.01	>0.05	<0.01	<0.01	0.02	<0.01	>0.05	<0.01	<0.01	<0.01	>0.05	<0.01
T2	-	-	>0.05	>0.05	<0.01	<0.01	<0.01	>0.05	<0.01	>0.05	>0.05	<0.01	<0.01	>0.05	>0.05	<0.01
T3	-	-	-	-	<0.01	<0.01	<0.01	>0.05	<0.01	>0.05	>0.05	<0.01	<0.01	>0.05	>0.05	<0.01
T4	-	-	-	-	-	-	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	0.05	0.02	<0.01	<0.01
T5	-	-	-	-	-	-	-	-	>0.05	>0.05	0.03	<0.01	<0.01	0.04	0.05	<0.01
T6	-	-	-	-	-	-	-	-	-	-	<0.01	<0.01	<0.01	>0.05	<0.01	<0.01
T7	-	-	-	-	-	-	-	-	-	-	-	-	<0.01	<0.01	0.04	<0.01
T8	-	-	-	-	-	-	-	-	-	--	-	-	-	-	<0.01	<0.01

Table 5.8: The results of Wilcoxon signed rank test on the average APFDc values for each pair of TCP techniques. This table follows exactly the same format as Table 5.7.

	T2		T3		T4		T5		T6		T7		T8		T9	
T1	0.02	<0.01	0.03	>0.05	0.04	>0.05	>0.05	>0.05	>0.05	>0.05	>0.05	<0.01	>0.05	>0.05	>0.05	<0.01
T2	-	-	>0.05	0.01	<0.01	<0.01	>0.05	0.04	>0.05	<0.01	>0.05	>0.05	>0.05	<0.01	>0.05	>0.05
T3	-	-	-	-	0.02	>0.05	>0.05	>0.05	>0.05	0.05	>0.05	0.03	>0.05	>0.05	>0.05	<0.01
T4	-	-	-	-	-	-	<0.01	>0.05	>0.05	>0.05	<0.01	>0.05	>0.05	>0.05	<0.01	<0.01
T5	-	-	-	-	-	-	-	-	>0.05	0.04	>0.05	>0.05	>0.05	>0.05	>0.05	<0.01
T6	-	-	-	-	-	-	-	-	-	-	<0.01	<0.01	>0.05	>0.05	0.01	<0.01
T7	-	-	-	-	-	-	-	-	-	-	-	-	>0.05	>0.05	>0.05	>0.05
T8	-	-	-	-	-	-	-	-	-	-	-	-	-	-	>0.05	<0.01

specting several subject programs. One illustrative example is that *scribe-java* scores 0.646 and 0.606 for the average values of APFD under TP$_{str}$ and TP$_{topic-r}$ respectively, which are notably worse than the results of TP$_{cg-tot}$ (0.718) and TP$_{cg-add}$ (0.733). To understand the reason for this discrepancy, we analyzed the test code and found that *Scribe-java* is documented/written more poorly than other programs. For instance, the program uses meaningless comments and variable names such as '$param1$', '$param2$', '$v1$', '$v2$' etc. This confirms the previously held notion [266] that static techniques which aim to prioritize test-cases through text-based diversity metrics experience performance degradation when applied to test cases written in a poor/generic fashion. It also suggests that researchers may take the subject characteristics into account when choosing TCP techniques in future work.

5.3.1.2 Results at Test Method Level

To further answer **RQ**$_3$ we ran all of the subject TCP techniques on the subject programs at the test-method level so that we can compare to the results at the test-class level outlined above (see Section 5.3.1.1). The results are shown in Figure 5.2 and Table 5.6. In terms of APFD, when examining the static techniques with the *test-method granularity*, they perform differently as compared to the results on the test-class level. For example, although TP$_{cg-add}$ still performs the best among static techniques, it is inferior to the most effective dynamic technique TP$_{add}$ (0.818 versus 0.898). This finding is consistent with previous studies [124]. Also, surprisingly, TP$_{topic-m}$ (0.817) achieves almost the same average APFD values as TP$_{cg-add}$, followed by TP$_{str}$, TP$_{topic-r}$ and TP$_{cg-tot}$ respectively. It is worth noting that the effectiveness of the topic-model based technique varies quite dramatically depending on the tools used for its implementation: Mallet [202] significantly outperforms the R-based implementation. Also, there is less variation in the APFD values at the test-method level compared to those at the test-class level, as shown in Figure 5.2 and Figure 5.3.

In terms of APFDc, the results for test-method level are generally consistent with the results on test-class level. For example, while TP$_{search}$ tend to be the most effective dynamic technique, the static TP$_{cg-add}$ outperforms all the studied static and dynamic techniques. The likely reason is that dynamic techniques tend to favor tests with higher coverage, which tend to cost more time to execute, leading to limited effectiveness in actual time cost reduction. The results of the HSD analysis on the APFDc values at the test-method level, indicate that all techniques are grouped into the same level (level A), implying that different TCP techniques share similar performance based on APFDc values, which is also consistent with the results of APFDc values at the test-class level. When examining the ranges of APFDc values for the test-method level (see Fig. 5.2 and Fig. 5.3), we find the APFDc values vary dramatically between subject programs. When comparing the results of APFD and APFDc values at the test-method level, the Kendall

τ_b rank coefficient τ_b is 0.556, impling that the APFDc results are less consistent with the APFD results at the test-method level. The reason is that test execution time distributions which are uncontrolled have large impacts the more effective/stable test-method-level results.

In addition, as a whole, the effectiveness of the dynamic techniques outpaces that of the static techniques at method-level granularity for the APFD metric, with TP$_{add}$ performing the best of all studied techniques (0.898). For the cost-cognizant APFDc metric, although there are no clear trends, the static techniques tend to perform even better than dynamic techniques, indicating the limitations of dynamic information for actual regression testing time reduction. Overall, on average, almost all static and dynamic TCPs perform better on the test-method level as compared to the results on the test-class level in terms of both APFD and APFDc. Logically, this is not surprising, as using a finer level of granularity (e.g., prioritizing individual test-methods) gives each technique more flexibility, which leads to more accurate targeting and prioritization.

Finally, to check for statistically significant variations in the mean APFD and APFDc values across all subjects and confirm/deny our null hypothesis for **RQ**$_1$ and **RQ**$_2$, we examine the results of the one-way ANOVA and Tukey HSD tests. The ANOVA test for APFD values, given in the second to last column of Tables 5.5 & 5.6, and both values are well below our established significance threshold of 0.05, thus signifying that the subject programs are statistically different from one another. This rejects the null hypothesis H_0 and we conclude that there are statistically significant differences between different TCP techniques in terms of APFD. The results of the Tukey HSD test also illustrate the statistically significant differences between the static and dynamic techniques, by grouping the techniques into categories with *A* representing the best performance and the following letters (e.g., *B*) representing groups with worse performance. We see that the groupings are similar for static and dynamic techniques. In order to illustrate the individual relationships between strategies, we present the results of the Wilcoxon signed rank test for all pairs of techniques at both granularity levels in Tables 5.7 and 5.8. The shaded cells repre-

sent statistically significant differences between techniques across all the subjects (e.g., $p < 0.05$). The Wilcoxon signed rank test further confirms that different techniques have statistically different APFD values at both test-class and test-method levels, as indicated by the shaded boxes. On the contrary, the results for APFDc ANOVA and HSD tests lead to different observations – different techniques *generally* do not have statistically different APFDc values (as shown in Tables 5.5 & 5.6), indicating that both static and dynamic techniques tend to perform similalrly for APFDc values. The Wilcoxon signed rank test for APFDc values of all pairs of techniques is shown in Table 5.8. The small number of shaded cells (i.e., $p < 0.05$) further confirms that different techniques tend to perform equivalently for APFDc. The likely reason for this is that APFDc is impacted by an additional randomly-distributed factor, i.e., tests tend to have randomly distributed execution times, leading to the observed results. It should be noted that in contrast to our previous work [193], our results for the HSD show less variance between the different approaches for APFD at both test-class and test-method level. This means that the approaches were grouped in fewer differing groups by the HSD test, indicating performance that is more comparatively similar. This illustrates the affect of generalizing across more subject programs.

In summary we answer **RQ**$_1$, **RQ**$_2$ & **RQ**$_3$ as follows:

RQ$_1$**:** There is a statistically significant difference between the APFD values of the two types (e.g., static and dynamic) of studied techniques. On average, static technique TP$_{cg-add}$ is the most effective technique at test-class level, whereas dynamic technique TP$_{add}$ is the most effective technique at test-method level. Overall, the static techniques outperform the dynamic ones at test-class level, but the dynamic techniques outperform the static ones at test-method level.

Table 5.9: Results for the ANOVA and Tukey HSD tests on the average APFD and APFDc values at test-class level across smaller subject programs. The last column shows the results for Kendall tau Rank Correlation Coefficient τ_b between the average APFDc and average APFD.

Metrics	Analysis	TP$_{cg-tot}$	TP$_{cg-add}$	TP$_{str}$	TP$_{topic-r}$	TP$_{topic-m}$	TP$_{total}$	TP$_{add}$	TP$_{art}$	TP$_{search}$	p-value	τ_b
APFD	Avg	0.759	0.764	0.758	0.658	0.729	0.707	0.746	0.629	0.743	5.42E-8	0.5
	HSD	A	A	A	BC	AB	ABC	A	C	A		
APFDc	Avg	0.618	0.633	0.653	0.563	0.652	0.558	0.592	0.585	0.591	0.503	
	HSD	A	A	A	A	A	A	A	A	A		

RQ$_2$: For the APFDc values, there is no statistically significant difference between the types of studied techniques. APFDc values are generally consistent with APFD values at test-class level but relatively less consistent at test-method level. Similar to the results from RQ$_1$, on average, static TP$_{cg-add}$ technique is the most effective technique at the test-class level, and the static techniques outperform the dynamic ones as a whole at test-class level. However, at test-method level, TP$_{cg-add}$ also performs best overall at the test-method level, indicating the superiority of static techniques to dynamic techniques in actual regression testing time reduction. In addition, APFDc values vary more dramatically across all subject programs as compared to AFPD values.

RQ$_3$: The test granularity significantly impacts the effectiveness of TCP techniques in terms of both APFD and APFDc, although the APFDc metric is affected to a much lesser extent. All the studied techniques perform better at test-method level as compared to test-class level. There is also less variation in the APFD values at method-level as compared to class-level, which signifies that the performance as measured by this metric is more stable at test-method level across the studied techniques.

Table 5.10: Results for the ANOVA and Tukey HSD tests on the average APFD and APFDc values at test-class level across larger subject programs. The last column shows the results for Kendall tau Rank Correlation Coefficient τ_b between the average APFDc and average APFD.

Metrics	Analysis	TP$_{cg-tot}$	TP$_{cg-add}$	TP$_{str}$	TP$_{topic-r}$	TP$_{topic-m}$	TP$_{total}$	TP$_{add}$	TP$_{art}$	TP$_{search}$	p-value	τ_b
APFD	Avg	0.796	0.816	0.796	0.692	0.762	0.769	0.791	0.637	0.787	7.73E-10	0.667
	HSD	B	B	B	B	B	B	A	B	A		
APFDc	Avg	0.686	0.724	0.681	0.586	0.661	0.670	0.708	0.640	0.706	0.132	
	HSD	A	A	A	A	A	A	A	A	A		

Table 5.11: Results for the ANOVA and Tukey HSD tests on the average APFD and APFDc values at test-method level across smaller subject programs. The last column shows the results for Kendall tau Rank Correlation Coefficient τ_b between the average APFDc and average APFD.

Metrics	Analysis	TP$_{cg-tot}$	TP$_{cg-add}$	TP$_{str}$	TP$_{topic-r}$	TP$_{topic-m}$	TP$_{total}$	TP$_{add}$	TP$_{art}$	TP$_{search}$	p-value	τ_b
APFD	Avg	0.751	0.799	0.799	0.771	0.804	0.797	0.885	0.791	0.878	2.29E-15	0.444
	HSD	B	B	B	B	B	B	A	B	A		
APFDc	Avg	0.604	0.700	0.670	0.655	0.673	0.605	0.657	0.645	0.696	0.572	
	HSD	A	A	A	A	A	A	A	A	A		

5.3.2 Impact of Subject Program's Size

Since developers may apply TCP techniques to subject systems in various sizes in practice, it is important to understand the potential impact of program size on the performance of TCP techniques. Thus we examine the differences between the performance of our studied TCP techniques on our 29 smaller subject systems and 29 larger subject systems. Table 5.9 and Table 5.10 present the TCP results at the test-class level on smaller and larger subject systems, respectively; similarly, Table 5.11 and Table 5.12 present the TCP results at the test-method level on smaller and larger subject systems, respectively.

From the tables, we can make the following observations. First, TCP techniques tend to perform better on larger subject systems than smaller subject systems. For example, for both test-class and test-method level, all the studied TCP techniques perform better on larger subject systems in terms of both APFD and APFDc. One potential reason is that larger subject systems tend to have more tests, leaving enough room for TCP techniques to reach optimization thresholds. This finding also demonstrates the scalability of the studied TCP techniques. Second, at the test-class level, static TCP techniques tend to outperform dynamic TCP techniques in terms of both APFD and APFDc on both subsets of subject systems; in contrast, at the test-method level, static TCP techniques are inferior than dynamic TCP techniques on both subsets of subjects in terms of APFD,

Table 5.12: Results for the ANOVA and Tukey HSD tests on the average APFD and APFDc values at test-method level across larger subject programs. The last column shows the results for Kendall tau Rank Correlation Coefficient τ_b between the average APFDc and average APFD.

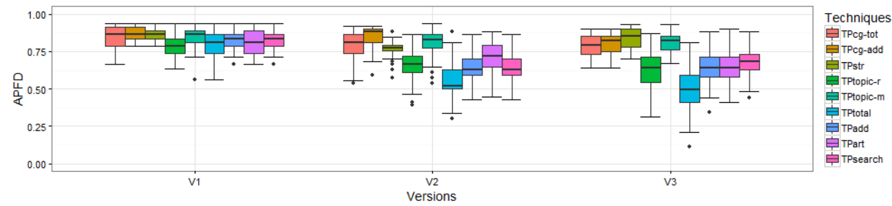
Metrics	Analysis	TP$_{cg-tot}$	TP$_{cg-add}$	TP$_{str}$	TP$_{topic-r}$	TP$_{topic-m}$	TP$_{total}$	TP$_{add}$	TP$_{art}$	TP$_{search}$	p-value	τ_b
APFD	Avg	0.777	0.837	0.827	0.791	0.829	0.821	0.912	0.805	0.892	3.21E-12	0.333
	HSD	C	BC	C	C	C	C	A	C	AB		
APFDc	Avg	0.673	0.774	0.671	0.702	0.686	0.660	0.759	0.692	0.774	0.086	
	HSD	A	A	A	A	A	A	A	A	A		

while TP$_{cg-add}$ outperforms all the other studied dynamic and static TCP techniques on both subsets of subjects in terms of APFDc. This finding is consistent with our findings in **RQ**$_1$ and **RQ**$_2$, indicating that subject size does not impact our findings when comparing the relative performance of the studied TCP techniques according to APFD and APFDc. Third, most studied TCP techniques perform better at test-method level as compared to test-class level in terms of both APFD and APFDc on both subsets of our subjects. This observation is also consistent with our comparative findings for **RQ**$_3$.

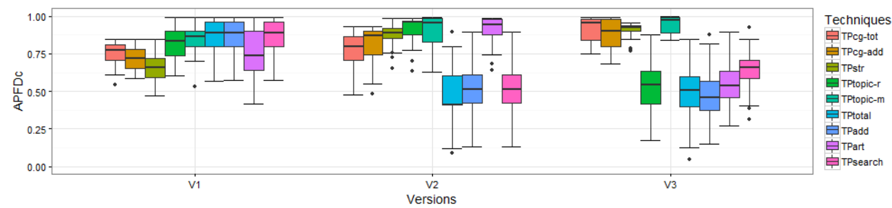
RQ$_4$**:** All the studied TCP techniques tend to perform better on larger subject systems, indicating the scalability of the studied TCP techniques. However, when comparing the performance of different TCP techniques to each other on either the large or small programs, we find results consistent to using the entire set of programs (both in terms of APFD(c) and differing test-case granularities). Thus we can conclude that program size has little effect when comparing the relative performance of TCP techniques on a given subject.

5.3.3 Impact of Software Evolution

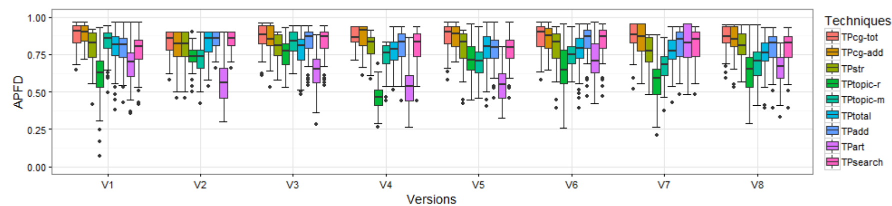
Figure 5.4 and Figure 5.5 present the impact of software evolution on the studied TCP techniques at the test-class and test-method levels, respectively. In each figure, each row presents both the APFD and APFDc results on the corresponding subject. In each sub-figure, the x-axis presents the versions used as the old version during software evolution (note that the most recent versions are always used as the new version during software evolution), while the y-axis presents the APFD or APFDc values. We show the APFD



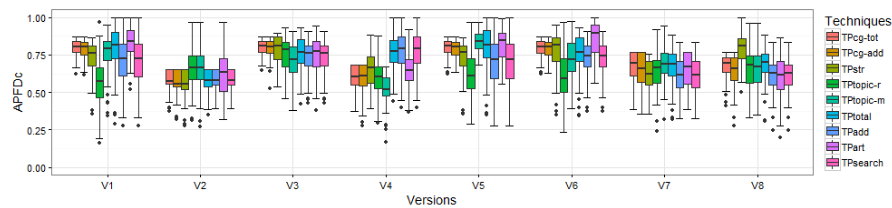
(a) TCP results on geojson-jackson (APFD)



(b) TCP results on geojson-jackson (APFDc)



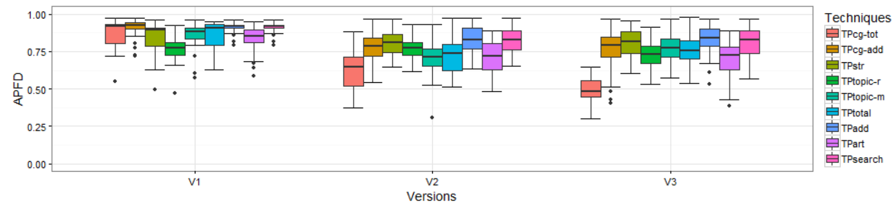
(c) TCP results on javapooet (APFD)



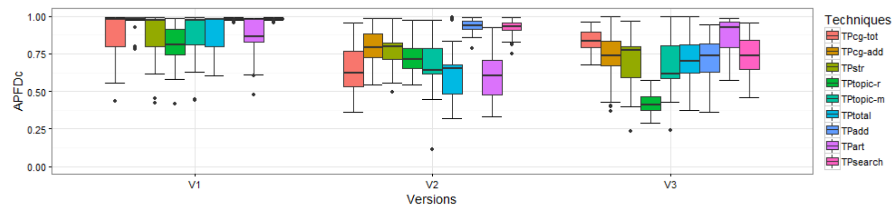
(d) TCP results on javapooet (APFDc)

Figure 5.4: Test-Class-level test prioritization in evolution

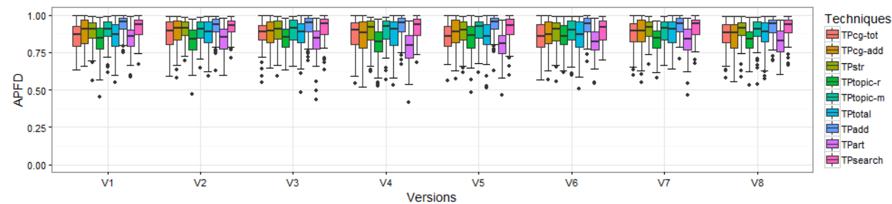
or APFDc distributions of different technique using box-plots of different colors, where the boxes represent the 25th to 75th percentiles, the centerlines represent the median values, and the dots represent the outlier points. Due to the limited space, we only show the results of two subject programs. The results of all twelve subject programs can be found in my online appendix [11]. Following prior work [192], using different versions as the old version during software evolution allows us to understand the impact of software evolution on TCP in details. To illustrate, for a project with n versions, where (n >2), we will have a set of $n - 1$ results for applying each studied TCP technique. That is, running



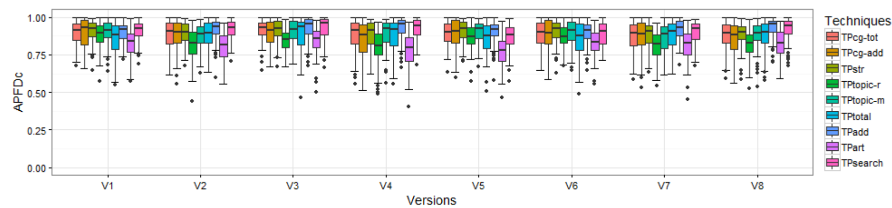
(a) TCP results on geojson-jackson (APFD)



(b) TCP results on geojson-jackson (APFDc)



(c) TCP results on javapooet (APFD)



(d) TCP results on javapooet (APFDc)

Figure 5.5: Test-Method-level test prioritization in evolution

TCPs on older program versions and then applying the prioritized set of test cases on the faulty variants of the most recent version (i.e., the latest versions with mutants) allows us to understand the performance of TCP techniques in evolutionary scenario. Note that more recent project versions may have tests not included in older project versions; we ignore such tests since the studied techniques would not be able to prioritize those tests based on old project versions.

If software evolution impacts TCP effectiveness, using earlier program versions for test prioritization would likely be less effective than using more up-to-date versions for test

prioritization due to code changes. In other words, APFD/APFDc values should increase when using newer versions for prioritization. However, we observe no such trend for either APFD or APFDc for any TCP technique on any studied subject at the test-class or test-method level. This observation confirms prior work [192, 133] that code changes do not impact the effectiveness of dynamic TCP techniques in terms of APFD. Furthermore, our work is the first to illustrate that the same finding holds for static TCP techniques as well as the more practical APFDc metric. These results most likely arise due to the fact that all studied TCP techniques approximate fault detection capabilities based on a certain set of criteria (such as call graphs, textual information, or code coverage), and software evolution usually does not result in large relative changes between commits for these different criteria (e.g., some tests may always have higher code coverage throughout project evolution).

We also find that the performance comparison in terms of APFD between dynamic and static TCP techniques is not impacted by software evolution. For instance, static TCP techniques tend to outperform dynamic TCP techniques in terms of both APFD at the test-class granularity on most subjects, while dynamic TCP techniques tend to outperform static TCP techniques in terms of APFD at the test-method granularity on most subjects. APFDc values tend to exhibit more variance during software evolution. For example, for the `javapoet` subject at test-class level, the static TP$_{str}$ technique outperforms all other techniques when using V_8 information to prioritize tests for V_8, while the dynamic TP$_{art}$ technique performs the best when using V_1 information to prioritize tests for V_8. One potential explanation for this observation is that tests with similar fault detection capabilities may have totally different execution times during evolution, causing high variances between APFDc values.

RQ$_5$: On average, software evolution does not have a clear impact on the measured effectiveness of the studied TCP techniques. Corroborating results of RQ$_1$ and RQ$_2$, we find that the APFD values for techniques tend to exhibit lower variance than

Table 5.13: Results for average APFD values on different sizes of mutation faults. The last column shows the results for Kendall tau Rank Correlation Coefficient τ_b between the average APFD values with different sizes of mutation faults and the average APFD values shown in Tables 5.5 and 5.6.

Granularity	Sizes	TP$_{cg-tot}$	TP$_{cg-add}$	TP$_{str}$	TP$_{topic-r}$	TP$_{topic-m}$	TP$_{total}$	TP$_{add}$	TP$_{art}$	TP$_{search}$	τ_b
Test-class	1	0.772	0.786	0.775	0.67	0.741	0.737	0.771	0.639	0.767	0.944
	2	0.776	0.788	0.775	0.671	0.745	0.738	0.771	0.639	0.768	1
	3	0.776	0.788	0.777	0.673	0.746	0.738	0.769	0.638	0.766	0.944
	4	0.777	0.789	0.777	0.675	0.745	0.739	0.771	0.637	0.768	0.944
	5	0.777	0.789	0.778	0.675	0.746	0.739	0.771	0.635	0.767	0.944
	6	0.777	0.79	0.777	0.675	0.746	0.738	0.77	0.634	0.767	1
	7	0.778	0.79	0.777	0.675	0.746	0.738	0.769	0.633	0.766	1
	8	0.778	0.79	0.777	0.676	0.746	0.738	0.77	0.633	0.766	1
	9	0.778	0.79	0.777	0.676	0.746	0.738	0.769	0.633	0.766	1
	10	0.778	0.79	0.777	0.676	0.747	0.738	0.77	0.634	0.766	1
Test-method	1	0.759	0.82	0.813	0.777	0.814	0.807	0.901	0.798	0.885	1
	2	0.763	0.82	0.816	0.781	0.818	0.809	0.902	0.802	0.887	1
	3	0.763	0.818	0.814	0.78	0.815	0.81	0.9	0.8	0.886	1
	4	0.764	0.819	0.814	0.782	0.817	0.811	0.901	0.802	0.887	1
	5	0.764	0.818	0.814	0.782	0.818	0.811	0.9	0.8	0.887	1
	6	0.762	0.817	0.813	0.781	0.816	0.809	0.9	0.8	0.886	1
	7	0.763	0.817	0.813	0.781	0.816	0.81	0.9	0.8	0.886	1
	8	0.763	0.818	0.812	0.781	0.816	0.81	0.899	0.8	0.886	1
	9	0.763	0.818	0.812	0.781	0.816	0.81	0.899	0.799	0.885	1
	10	0.764	0.818	0.813	0.781	0.816	0.809	0.899	0.799	0.885	1

APFDc values.

5.3.4 Impact of Mutant Quantities on TCP Effectiveness

Prior work examining TCP generally directly seeds a certain number of faults to form a faulty version (or groups of faulty versions) to investigate TCP effectiveness according to the APFD or APFDc, similar to the setup used in our study to answer RQ$_1$-RQ$_3$. However, we wish to further analyze the impacts of the quantity of mutants utilized in experimental settings and whether or not this impacts the effectiveness of techniques. The experimental results for APFD and APFDc are shown Tables 5.13 and 5.14, respectively. In each table, Column 1 lists the test-case granularities studied, column 2 lists the number of mutants seeded into each faulty version/group, columns 3-11 present the APFD/APFDc results for each studied technique, and finally the last column presents the Kendall τ_b Rank Correlation Coefficient between the average values with each fault quantity and our default settings (shown in Tables 5.5 and 5.6). From the tables, we make the following ob-

Table 5.14: Results for average APFDc values on different sizes of mutation faults. This table follows the same format as Table 5.13.

Granularity	Sizes	TP$_{cg-tot}$	TP$_{cg-add}$	TP$_{str}$	TP$_{topic-r}$	TP$_{topic-m}$	TP$_{total}$	TP$_{add}$	TP$_{art}$	TP$_{search}$	τ_b
Test-class	1	0.655	0.681	0.668	0.572	0.658	0.619	0.653	0.622	0.652	0.944
	2	0.653	0.679	0.668	0.571	0.66	0.616	0.652	0.621	0.652	0.944
	3	0.653	0.679	0.669	0.574	0.66	0.617	0.651	0.619	0.65	0.944
	4	0.652	0.678	0.667	0.575	0.657	0.617	0.652	0.615	0.651	1
	5	0.652	0.678	0.668	0.575	0.659	0.616	0.651	0.614	0.651	1
	6	0.653	0.68	0.667	0.575	0.658	0.616	0.65	0.613	0.65	1
	7	0.653	0.68	0.667	0.575	0.658	0.616	0.65	0.613	0.65	1
	8	0.654	0.681	0.667	0.576	0.659	0.617	0.651	0.613	0.65	1
	9	0.654	0.68	0.668	0.576	0.659	0.617	0.651	0.613	0.65	1
	10	0.654	0.68	0.668	0.576	0.659	0.616	0.651	0.613	0.65	1
Test-method	1	0.637	0.745	0.671	0.681	0.681	0.634	0.715	0.675	0.739	0.889
	2	0.639	0.739	0.675	0.683	0.684	0.634	0.714	0.675	0.739	0.944
	3	0.639	0.738	0.672	0.681	0.679	0.634	0.711	0.672	0.738	0.889
	4	0.638	0.737	0.671	0.68	0.68	0.634	0.711	0.672	0.737	0.944
	5	0.638	0.737	0.672	0.68	0.681	0.635	0.711	0.671	0.738	0.944
	6	0.636	0.738	0.671	0.678	0.68	0.634	0.711	0.671	0.737	0.944
	7	0.637	0.738	0.671	0.679	0.68	0.635	0.711	0.671	0.737	1
	8	0.638	0.739	0.67	0.679	0.68	0.635	0.71	0.671	0.737	0.944
	9	0.638	0.738	0.67	0.679	0.679	0.635	0.71	0.67	0.737	0.944
	10	0.638	0.737	0.67	0.678	0.679	0.634	0.709	0.67	0.736	1

servations. For both APFD and APFDc values, the mutant quantity does not dramatically impact the results for all of the studied techniques. For example, at the test-class level, the average APFD values of TP$_{cg-add}$ range from 0.786 to 0.790 for all the studied fault quantity settings, while its APFDc values range from 0.678 to 0.681. This finding indicates that the effectiveness of the studied techniques when seeding any number of mutants into each group will be roughly equivalent, demonstrating the validity of the mutant seeding processes of prior TCP work [192, 206, 306, 124]. The largest impact that fault quantities had were for the APFDc metric at the test-method level. The likely reason for this is that the test-method level techniques prioritize tests at a finer granularity, and thus are more sensitive to the impact of execution time. For example, APFDc of fault groups with only one fault in each group only considers the time to detect only the first fault (while APFDc of fault groups with 5 faults in each group considers the time to detect all the 5 faults), leading to the higher variance.

RQ$_6$: The quantity of mutants used, as stipulated in the experimental settings of mutation analysis-based evaluations of TCP approaches, does not significantly im-

Table 5.15: Results for average APFD values on different types of mutation faults. The last column shows the results for Kendall tau Rank Correlation Coefficient τ_b between the average APFD values with different types of mutation faults and the average APFD values shown in Tables 5.5 and 5.6.

Gra.	Types	TP$_{cg-tot}$	TP$_{cg-add}$	TP$_{str}$	TP$_{topic-r}$	TP$_{topic-m}$	TP$_{total}$	TP$_{add}$	TP$_{art}$	TP$_{search}$	τ_b
Class	NegateConditionals	0.785	0.796	0.792	0.699	0.762	0.749	0.788	0.677	0.784	0.889
	RemoveConditional	0.792	0.803	0.794	0.703	0.764	0.755	0.792	0.679	0.788	0.944
	ConstructorCall	0.784	0.797	0.787	0.682	0.757	0.74	0.772	0.663	0.766	0.944
	NonVoidMethodCall	0.774	0.783	0.774	0.668	0.74	0.731	0.761	0.626	0.756	1
	Math	0.782	0.787	0.776	0.685	0.748	0.706	0.775	0.657	0.772	1
	MemberVariable	0.798	0.816	0.772	0.69	0.752	0.776	0.794	0.676	0.79	0.778
	InlineConstant	0.76	0.778	0.776	0.687	0.752	0.707	0.752	0.641	0.75	0.889
	Increments	0.791	0.815	0.792	0.725	0.784	0.726	0.8	0.7	0.796	0.722
	ArgumentPropagation	0.775	0.787	0.775	0.676	0.751	0.738	0.769	0.624	0.766	1
	ConditionalsBoundary	0.787	0.81	0.809	0.709	0.776	0.737	0.78	0.69	0.778	0.944
	Switch	0.859	0.838	0.882	0.822	0.856	0.849	0.867	0.757	0.864	0.5
	VoidMethodCall	0.782	0.781	0.771	0.659	0.72	0.757	0.749	0.623	0.748	0.778
	InvertNegs	0.744	0.805	0.849	0.669	0.738	0.63	0.757	0.726	0.743	0.667
	ReturnVals	0.781	0.802	0.779	0.671	0.748	0.732	0.762	0.651	0.759	1
	RemoveIncrements	0.755	0.797	0.761	0.684	0.738	0.685	0.762	0.645	0.759	0.778
Method	NegateConditionals	0.787	0.842	0.845	0.809	0.851	0.829	0.921	0.828	0.911	0.889
	RemoveConditional	0.792	0.846	0.848	0.813	0.852	0.834	0.925	0.832	0.914	0.889
	ConstructorCall	0.756	0.816	0.8	0.782	0.806	0.791	0.886	0.808	0.873	0.833
	NonVoidMethodCall	0.755	0.809	0.811	0.766	0.815	0.804	0.891	0.783	0.874	0.889
	Math	0.745	0.801	0.789	0.776	0.795	0.777	0.902	0.809	0.882	0.778
	MemberVariable	0.799	0.858	0.837	0.809	0.843	0.838	0.914	0.832	0.905	0.944
	InlineConstant	0.735	0.777	0.785	0.766	0.788	0.774	0.881	0.794	0.869	0.667
	Increments	0.771	0.853	0.857	0.835	0.865	0.81	0.942	0.854	0.926	0.722
	ArgumentPropagation	0.752	0.821	0.824	0.768	0.829	0.81	0.899	0.793	0.881	0.889
	ConditionalsBoundary	0.761	0.825	0.827	0.806	0.833	0.809	0.901	0.819	0.89	0.833
	Switch	0.86	0.881	0.932	0.871	0.946	0.902	0.968	0.892	0.952	0.778
	VoidMethodCall	0.771	0.805	0.792	0.766	0.8	0.81	0.871	0.779	0.862	0.778
	InvertNegs	0.752	0.843	0.813	0.69	0.784	0.726	0.849	0.799	0.812	0.611
	ReturnVals	0.766	0.842	0.82	0.785	0.819	0.792	0.889	0.803	0.876	0.889
	RemoveIncrements	0.737	0.84	0.861	0.807	0.868	0.777	0.935	0.846	0.907	0.722

pact the effectiveness of TCP techniques in terms of either APFD or APFDc, demonstrating the validity of the fault seeding process of prior work in this context.

5.3.5 Impact of Mutant Types on TCP Effectiveness

In this RQ, we further investigate whether different mutant types may impact TCP results in terms of either APFD and APFDc. The experimental results for APFD and APFDc are shown Tables 5.15 and 5.16, respectively. In each table, column 1 lists the test-case granularities studied, column 2 lists the different types of mutants seeded into each faulty version, columns 3-11 present the APFD/APFDc results for each studied technique,

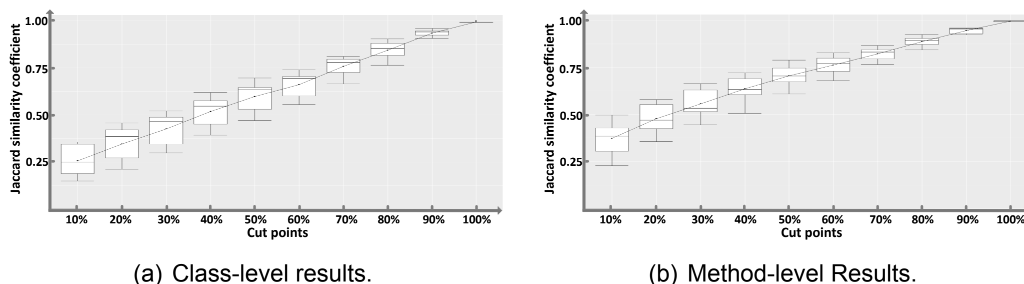
Table 5.16: Results for average APFDc values on different types of mutation faults. The last column shows the results for Kendall tau Rank Correlation Coefficient τ_b between the average APFDc values with different types of mutation faults and the average APFDc values shown in Tables 5.5 and 5.6.

Gra.	Types	TP$_{cg-tot}$	TP$_{cg-add}$	TP$_{str}$	TP$_{topic-r}$	TP$_{topic-m}$	TP$_{total}$	TP$_{add}$	TP$_{art}$	TP$_{search}$	τ_b
Class	NegateConditionals	0.646	0.67	0.671	0.591	0.666	0.614	0.653	0.639	0.652	0.778
	RemoveConditional	0.655	0.677	0.673	0.594	0.668	0.62	0.659	0.64	0.658	0.833
	ConstructorCall	0.651	0.682	0.672	0.575	0.664	0.609	0.644	0.627	0.64	0.944
	NonVoidMethodCall	0.642	0.665	0.652	0.556	0.645	0.598	0.628	0.593	0.625	1
	Math	0.658	0.66	0.653	0.576	0.636	0.559	0.635	0.611	0.634	0.778
	MemberVariable	0.668	0.699	0.656	0.587	0.66	0.649	0.671	0.641	0.667	0.556
	InlineConstant	0.62	0.647	0.651	0.57	0.644	0.567	0.617	0.607	0.619	0.778
	Increments	0.636	0.663	0.655	0.584	0.657	0.584	0.639	0.633	0.639	0.667
	ArgumentPropagation	0.655	0.668	0.653	0.571	0.66	0.613	0.644	0.593	0.644	0.889
	ConditionalsBoundary	0.64	0.673	0.674	0.591	0.655	0.604	0.643	0.639	0.647	0.722
	Switch	0.806	0.775	0.803	0.663	0.779	0.736	0.763	0.822	0.759	0.333
	VoidMethodCall	0.628	0.631	0.624	0.519	0.602	0.606	0.602	0.579	0.605	0.611
	InvertNegs	0.537	0.579	0.705	0.498	0.666	0.472	0.746	0.747	0.714	0
	ReturnVals	0.652	0.685	0.661	0.56	0.649	0.603	0.636	0.619	0.635	0.889
	RemoveIncrements	0.605	0.639	0.664	0.546	0.628	0.544	0.602	0.649	0.602	0.5
Method	NegateConditionals	0.647	0.749	0.691	0.699	0.7	0.637	0.724	0.694	0.757	0.889
	RemoveConditional	0.653	0.753	0.696	0.704	0.706	0.643	0.73	0.701	0.762	0.889
	ConstructorCall	0.632	0.728	0.651	0.667	0.658	0.621	0.698	0.68	0.715	0.778
	NonVoidMethodCall	0.617	0.716	0.653	0.648	0.664	0.619	0.688	0.637	0.709	0.889
	Math	0.631	0.723	0.655	0.679	0.683	0.581	0.722	0.709	0.748	0.778
	MemberVariable	0.673	0.773	0.701	0.708	0.703	0.657	0.731	0.715	0.761	0.778
	InlineConstant	0.598	0.683	0.622	0.654	0.636	0.573	0.67	0.659	0.699	0.722
	Increments	0.606	0.739	0.671	0.706	0.695	0.611	0.731	0.713	0.763	0.667
	ArgumentPropagation	0.609	0.709	0.662	0.646	0.674	0.611	0.698	0.64	0.713	0.833
	ConditionalsBoundary	0.605	0.716	0.641	0.682	0.661	0.612	0.695	0.68	0.73	0.722
	Switch	0.752	0.787	0.786	0.78	0.796	0.768	0.812	0.777	0.839	0.722
	VoidMethodCall	0.626	0.683	0.615	0.619	0.63	0.599	0.635	0.613	0.666	0.833
	InvertNegs	0.7	0.816	0.6	0.537	0.655	0.651	0.66	0.713	0.752	0.389
	ReturnVals	0.63	0.742	0.661	0.671	0.669	0.608	0.687	0.664	0.711	0.889
	RemoveIncrements	0.597	0.752	0.699	0.697	0.726	0.575	0.709	0.73	0.743	0.667

and finally the last column presents the Kendall τ_b Rank Correlation Coefficient between the average APFD/APFDc values with each fault type and our default settings (shown in Tables 5.5 and 5.6). From the tables, we can make the following observations. Overall, the vast majority of the studied mutant types tend to have a medium to high coefficient (i.e., the range of Kendall correlation coefficient values is from 0.5 to 1.0). This implies that the performance of TCP techniques when applied to detecting only certain mutant types highly correlates to the performance observed when applied to detecting all mutants. This finding indicates that the findings in prior work on TCP (including this work) generally hold across mutants seeded with differing mutation operators. Second, we also observe that there are several mutant types with low correlation with our default fault seeding, e.g., the `Invert Negs Mutator` and the `Switch Mutator` have the lowest correlation in both

studied test granularities for both APFD and APFDc. Upon further investigation, we found one likely explanation to be the small number of mutants generated by such mutators. For example, the number of `Invert Negs Mutator` is quite small as compared to other type of mutation faults (since it is only applicable to the cases of negative numbers), thus the results are dramatically different as compared to the results of mutation faults with the default setting. The `Switch Mutator` also has small Kendall correlation coefficient values as compared to other mutators. This is due to the fact that like the `Invert Negs Mutator` – the number of `Switch Mutator` faults is quite small as compared to other type of mutation faults (since it is only applicable to the cases of switches, which are not intensively used in common programs). Thus, the results are dramatically different as compared to the results of mutation faults with the default setting. Furthermore, we also observe that some mutation faults are more subtle than others. For example, the mutation faults created by `Invert Negs Mutator` tend to be more subtle than other types of mutation faults. For example, the `Invert Negs Mutator` operator simply inverts negation of an integer and floating point number (e.g., changing "`return -i;`" into "`return i;`"), while `Non-Void Method Call Mutator` or `Void Method Call Mutator` directly removes an entire method invocation. The subtle mutation faults introduced by `Invert Negs Mutator` can be harder to detect, making various static and dynamic techniques perform worse on those faults since the coverage or call graph information won't provide precise guide for detecting such subtle faults.

RQ$_7$: The mutation operators used to seed faults, as stipulated in the experimental settings of mutation analysis-based evaluations of TCP approaches, do not significantly impact the effectiveness of TCP techniques in terms of either APFD or APFDc, demonstrating the comparative validity of the fault seeding process of prior work in this context.



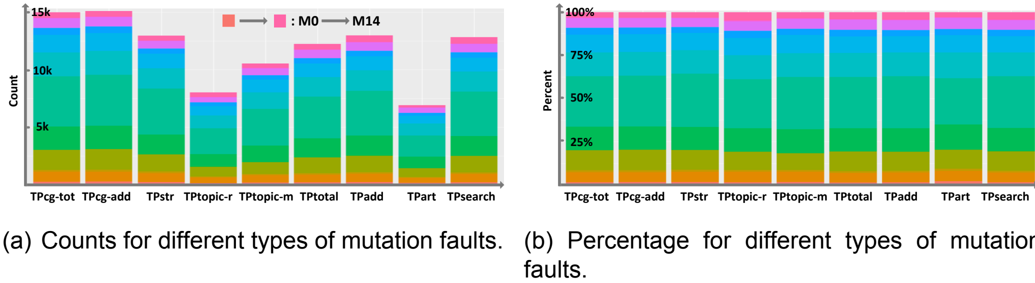
(a) Class-level results.

(b) Method-level Results.

Figure 5.6: Average Jaccard similarity of faults detected between static and dynamic techniques across all subjects at method and class-level granularity.

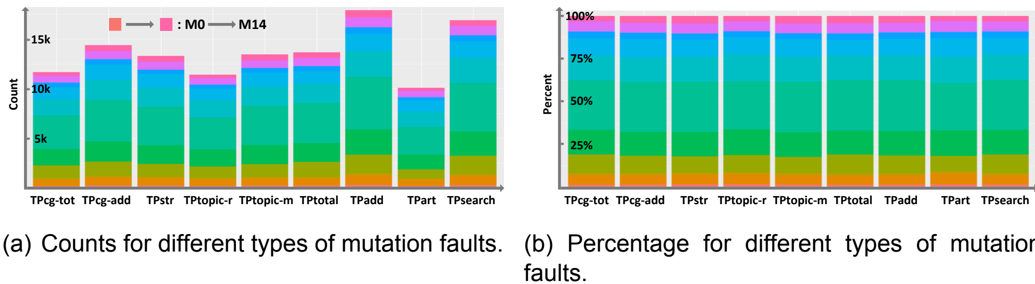
5.3.6 Similarity between Uncovered Faults for Different TCP techniques

The overall results for the similarity are shown in Figure 5.6. The two figures represent the results comparing the average Jaccard similarity of the studied static techniques to the studied dynamic techniques for all subject programs across 500 randomly sampled faults at different prioritization cut points. These results indicate that there is only a small amount of similarity between these two classifications of techniques at the higher level cut points. More specifically, for test-method level, only $\approx$ 30% of the detected faults are similar between the two types of techniques for the top 10% of the prioritized test cases, and at test-class level only about $\approx$ 25% are similar for the top 10% of prioritized test cases. This result illustrates one of the key findings of this study: The studied static and dynamic TCP techniques do not uncover similar program faults at the top cut points of prioritized test cases. The potential reason for these results is that different techniques use different types of information to prioritize test cases. For example, the studied static techniques typically aim to promote diversity between prioritized test cases using similarity/diversity metrics such as textual distance or call-graph information. In contrast, the studied dynamic TCPs consider statement-level dynamic coverage to prioritize test cases. This finding raises interesting questions for future work regarding the possibility of combining static and dynamic information and the relative *importance* of faults that differing techniques might uncover. It should be noted that different coverage granularities for dynamic TCPs may also effect the results of similarity, however we leave such an in-



(a) Counts for different types of mutation faults. (b) Percentage for different types of mutation faults.

Figure 5.7: Counts and percentage for different types of mutation faults across all subjects at cut point 10% for class-level granularity. The types of mutation faults are classified based on the mutation operators shown in Table 5.3.



(a) Counts for different types of mutation faults. (b) Percentage for different types of mutation faults.

Figure 5.8: Counts and percentage for different types of mutation faults across all subjects at cut point 10% for method-level granularity. The types of mutation faults are classified based on the mutation operators shown in Table 5.3.

vestigation for future work. From these figures we can also conclude that the techniques are slightly more similar at method level than at class level.

To further illustrate this point we calculated the Jaccard coefficients for each pair of TCPs for each subject program, and show the results in Table 5.17 and Table 5.18. For each pair of techniques we group the subjects into the categories described in Section 6.2. Due to space limitations, we only show results for the top 10% and top 50% of prioritized test-cases, a complete dataset can be found at [195]. The results confirm the conclusions drawn from Figure 5.6. It is clear that when comparing the studied static and dynamic techniques, more subjects are classified into the *highly-dissimilar* and *dissimilar* categories at the cut point top 10% on both of test-method and test-class levels. Another relevant conclusion that can be made is that the dissimilarity between techniques is not universal across all subjects. That is, even though two techniques may be dissimilar across several subjects, there are some cases where similarity still exists. This suggests

that only certain types of programs that exhibit different characteristics may present the opportunity of performance improvement for TCPs by using both static and dynamic information. In addition, at the cut point for the top 50% of prioritized test cases, it obvious that fewer subjects are classified into the *highly-dissimilar* and *dissimilar* categories. This is not surprising, because as the cut point increases the different techniques tend to discover more faults, limiting the potential for variance.

There are two potential reasons why we might observe higher numbers of *dissimilar* faults detected at the highest cut points: 1) different types of mutants are being detected; and 2) mutants of the same type in different locations are being detected. To investigate whether our observations are due to different fault types, we examine the counts and the percentages for different types of mutants that are detected by top 10% test cases at both the test-class and test-method level. The results are shown in Figures 5.7 and 5.8. When observing that the ratio for different types of mutation faults detected by different TCP techniques, we find that, as a whole, all TCP techniques detect a similar ratio of each mutant type, implying that mutant type is generally not the cause for the dissimilar faults at the higher cut points, but rather, mutants of the same type present different locations in source code are the more likely explanation.

RQ$_8$: The studied static and dynamic TCP techniques tend to discover dissimilar faults for the most highly prioritized test cases. Specifically, at the test-method level static and dynamic techniques agree only on $\approx$ 35% of uncovered faults for the top 10% of prioritized test cases. Additionally, a subset of subjects exhibit higher levels of uncovered fault similarity, suggesting that only software systems with certain characteristics may benefit from differing TCP approaches. Furthermore, the most highly prioritized test cases by different TCP techniques share similar capabilities in detecting different types of mutation faults.

Table 5.17: The classification of subjects on different granularities using Jaccard distance. The four values in each cell are the numbers of subject projects, the faults of which detected by two techniques are highly dissimilar, dissimilar, similar and highly similar respectively. The technique enumeration is consistent with Table 5.7.

(a) This table shows the classification of subjects at the cut point 10% on test-class level.

	T1				T2				T3				T4				T5				T6				T7				T8				T9			
TP1	–	–	–	–	3	2	16	37	11	13	18	16	28	20	3	7	14	18	17	9	11	19	13	15	15	14	16	13	32	18	5	3	15	15	13	15
TP2	3	2	16	37	–	–	–	–	11	14	15	18	27	23	3	5	13	24	14	7	11	20	14	13	14	16	13	15	30	23	4	1	13	17	10	18
TP3	11	13	18	16	11	14	15	18	–	–	–	–	30	13	10	5	12	12	14	20	20	15	12	11	18	18	13	9	33	15	6	4	18	18	12	10
TP4	28	20	3	7	27	23	3	5	30	13	10	5	–	–	–	–	14	15	12	17	26	17	11	4	24	15	15	4	31	13	7	7	24	16	13	5
TP5	14	18	17	9	13	24	14	7	12	12	14	20	14	15	12	17	–	–	–	–	19	24	7	8	21	20	10	7	30	16	7	5	21	20	9	8
TP6	11	19	13	15	11	20	14	13	20	15	12	11	26	17	11	4	19	24	7	8	–	–	–	–	2	13	11	32	28	13	9	8	2	12	14	30
TP7	15	14	16	13	14	16	13	15	18	18	13	9	24	15	15	4	21	20	10	7	2	13	11	32	–	–	–	–	25	16	13	4	0	0	2	56
TP8	32	18	5	3	30	23	4	1	33	15	6	4	31	13	7	7	30	16	7	5	28	13	9	8	25	16	13	4	–	–	–	–	25	15	15	3
TP9	15	15	13	15	13	17	10	18	18	18	12	10	24	16	13	5	21	20	9	8	2	12	14	30	0	0	2	56	25	15	15	3	–	–	–	–
Total	129	119	101	115	122	139	89	114	153	118	100	93	204	132	74	54	144	149	90	81	119	133	91	121	119	112	93	140	234	129	66	35	118	113	88	145

(b) This table shows the classification of subjects at the cut point 10% on test-method level.

	T1				T2				T3				T4				T5				T6				T7				T8				T9			
TP1	–	–	–	–	3	14	28	13	11	23	17	7	11	29	15	3	12	20	20	6	6	14	19	19	6	19	22	11	21	23	11	3	3	21	20	14
TP2	3	14	28	13	–	–	–	–	7	18	24	9	5	22	26	5	6	19	28	5	1	21	25	11	2	14	23	19	14	21	19	4	3	14	25	16
TP3	11	23	17	7	7	18	24	9	–	–	–	–	4	21	27	6	0	3	17	38	7	16	23	12	4	12	26	16	15	22	20	1	5	12	29	12
TP4	11	29	15	3	5	22	26	5	4	21	27	6	–	–	–	–	6	22	26	4	7	27	21	3	5	26	23	4	12	25	19	2	4	26	22	6
TP5	12	20	20	6	6	19	28	5	0	3	17	38	6	22	26	4	–	–	–	–	7	17	24	10	6	5	34	13	13	24	20	1	7	7	34	10
TP6	6	14	19	19	1	21	25	11	7	16	23	12	7	27	21	3	7	17	24	10	–	–	–	–	1	11	29	17	19	21	16	2	2	11	26	19
TP7	6	19	22	11	2	14	23	19	4	12	26	16	5	26	23	4	6	5	34	13	1	11	29	17	–	–	–	–	10	19	26	3	1	3	6	48
TP8	21	23	11	3	14	21	19	4	15	22	20	1	12	25	19	2	13	24	20	1	19	21	16	2	10	19	26	3	–	–	–	–	13	17	23	5
TP9	3	21	20	14	3	14	25	16	5	12	29	12	4	26	22	6	7	7	34	10	2	11	26	19	1	3	6	48	13	17	23	5	–	–	–	–
Total	73	163	152	76	41	143	198	82	53	127	183	101	54	198	179	33	57	117	203	87	50	138	183	93	35	109	189	131	117	172	154	21	38	111	185	130

5.3.7 Efficiency of Static TCP Techniques

The results of time costs for the studied static techniques at both of test-method and test-class levels are shown in Table 5.19. Note that, the time of pre-processing for TP$_{cg-tot}$ and TP$_{cg-add}$ are the same for both method and class levels. As the table shows, all studied techniques require similar time to pre-process the data at both method and class levels and to rank test cases on class level. But the times for prioritization are quite different at method level. We find that TP$_{cg-tot}$ and TP$_{cg-add}$ take much less time to prioritize test cases (totaling 23.78 seconds and 37.02 seconds), as compared to TP$_{str}$ (totalling 78,835.97 seconds), TP$_{topic-r}$ (totalling 48,310.93 seconds) and TP$_{topic-m}$ (totalling 15,573.71 seconds). In particular, the following three techniques, TP$_{str}$, TP$_{topic-r}$, and TP$_{topic-m}$ take much longer time on some subjects (e.g., *P53* and *P58*). These subjects have a large number of test cases (see Table 5.2), implying that TP$_{str}$, TP$_{topic-r}$ and TP$_{topic-m}$ will take more time as the number of test cases increases. Overall, all techniques take a reasonable amount of time to preprocess data and prioritize test cases. At test-method level, TP$_{cg-tot}$ and TP$_{cg-add}$ are much more efficient. TP$_{str}$, TP$_{topic-r}$ and TP$_{topic-m}$ require more time to prioritize increasing numbers of test cases.

Table 5.18: The classification of subjects on different granularities using Jaccard distance. The four values in each cell are the numbers of subject projects, the faults of which detected by two techniques are highly dissimilar, dissimilar, similar and highly similar respectively. The technique enumeration is consistent with Table 5.7.

(a) This table shows the classification of subjects at the cut point 50% on test-class level.

	T1				T2				T3				T4			
TP1	–	–	–	–	0	2	10	46	0	1	10	47	1	6	30	21
TP2	0	2	10	46	–	–	–	–	1	0	9	48	0	6	28	24
TP3	0	1	10	47	1	0	9	48	–	–	–	–	1	6	23	28
TP4	1	6	30	21	0	6	28	24	1	6	23	28	–	–	–	–
TP5	1	3	17	37	0	2	17	39	0	2	11	45	0	7	20	31
TP6	0	6	17	35	0	2	23	33	1	4	19	34	0	7	32	19
TP7	0	2	22	34	0	1	17	40	1	1	16	40	1	3	29	25
TP8	2	9	29	18	2	7	24	25	2	9	21	26	4	9	19	26
TP9	0	2	22	34	0	1	16	41	1	1	16	40	1	3	29	25
Total	4	31	157	272	3	21	144	296	7	24	125	308	8	47	210	199

	T5				T6				T7				T8				T9			
TP1	1	3	17	37	0	6	17	35	0	2	22	34	2	9	29	18	0	2	22	34
TP2	0	2	17	39	0	2	23	33	0	1	17	40	2	7	24	25	0	1	16	41
TP3	0	2	11	45	1	4	19	34	1	1	16	40	2	9	21	26	1	1	16	40
TP4	0	7	20	31	0	7	32	19	1	3	29	25	4	9	19	26	1	3	29	25
TP5	–	–	–	–	0	7	20	31	0	4	17	37	3	7	25	23	0	4	18	36
TP6	0	7	20	31	–	–	–	–	0	2	18	38	2	15	28	13	0	2	18	38
TP7	0	4	17	37	0	2	18	38	–	–	–	–	2	11	13	32	0	0	0	58
TP8	3	7	25	23	2	15	28	13	2	11	13	32	–	–	–	–	2	10	13	33
TP9	0	4	18	36	0	2	18	38	0	0	0	58	2	10	13	33	–	–	–	–
Total	4	36	145	279	3	45	175	241	4	24	132	304	19	77	172	196	4	23	132	305

(b) This table shows the classification of subjects at the cut point 50% on test-method level.

	T1				T2				T3				T4			
TP1	–	–	–	–	1	0	20	37	0	3	20	35	1	1	31	25
TP2	1	0	20	37	–	–	–	–	0	1	10	47	0	1	16	41
TP3	0	3	20	35	0	1	10	47	–	–	–	–	0	2	18	38
TP4	1	1	31	25	0	1	16	41	0	2	18	38	–	–	–	–
TP5	1	3	17	37	0	0	15	43	0	0	3	55	0	2	12	44
TP6	0	1	13	44	0	2	16	40	0	1	12	45	0	2	20	36
TP7	0	1	17	40	0	0	9	49	0	0	6	52	0	0	13	45
TP8	1	1	23	33	0	0	12	46	0	1	10	47	0	0	14	44
TP9	0	1	17	40	0	0	10	48	0	0	8	50	0	0	14	44
Total	4	11	158	291	1	4	108	351	0	8	87	369	1	8	138	317

	T5				T6				T7				T8				T9			
TP1	1	3	17	37	0	1	13	44	0	1	17	40	1	1	23	33	0	1	17	40
TP2	0	0	15	43	0	2	16	40	0	0	9	49	0	0	12	46	0	0	10	48
TP3	0	0	3	55	0	1	12	45	0	0	6	52	0	1	10	47	0	0	8	50
TP4	0	2	12	44	0	2	20	36	0	0	13	45	0	0	14	44	0	0	14	44
TP5	–	–	–	–	0	0	16	42	0	0	5	53	0	1	7	50	0	0	6	52
TP6	0	0	16	42	–	–	–	–	0	0	10	48	0	0	19	39	0	0	9	49
TP7	0	0	5	53	0	0	10	48	–	–	–	–	0	0	3	55	0	0	2	56
TP8	0	1	7	50	0	0	19	39	0	0	3	55	–	–	–	–	0	0	2	56
TP9	0	0	6	52	0	0	9	49	0	0	2	56	0	0	2	56	–	–	–	–
Total	1	6	81	376	0	6	115	343	0	1	65	398	1	3	90	370	0	1	68	395

Table 5.19: Execution costs for the static TCP techniques. The table lists the average, min, max, and sum of costs across all subject programs for both test-class level and test-method level (i.e., cost at test-class level/cost at test-method level). Time is measured in second.

Techniques	Pre-processing				Test Prioritization			
	Avg.	Min	Max	Sum	Avg.	Min	Max	Sum
TP$_{cg-tot}$	244.14/244.14	1.21/1.21	13785.86/13785.86	14159.97/14159.97	0.20/0.41	0/0	3.10/10.58	11.37/23.78
TP$_{cg-add}$	244.14/244.14	1.21/1.21	113785.86/13785.86	14159.97/14159.97	0.18/0.64	0/0	2.87/19.98	10.59/37.02
TP$_{str}$	0.35/0.37	0.04/0.04	2.95/2.41	20.04/21.63	4.03/1,359.24	0.01/0.02	115.82/57,134.30	233.76/78,835.97
TP$_{topic-r}$	0.41/1.55	0.03/0.09	3.81/14.80	24.99/89.63	0.15/832.95	0/0.01	1.72/40,594.66	8.50/48,310.93
TP$_{topic-m}$	1.51/3.79	0.13/0.22	12.10/50.14	87.76/219.93	0.19/268.51	0/0.07	1.98/10,925.26	10.95/15,573.71

RQ$_9$: On test-method level, TP$_{cg-tot}$ and TP$_{cg-add}$ are much more efficient in prioritizing test cases. TP$_{str}$, TP$_{topic-r}$ and TP$_{topic-m}$ would take more time when the number of test cases increases. The time of pre-processing and prioritization on test class level for all static techniques are quite similar.

5.4 Threats to Validity

Threats to Internal Validity: In our implementation, we used PIT to generate mutation faults to simulate real program faults. One potential threat is that the mutation faults may not reflect all "natural" characteristics of real faults. However, mutation faults have been widely used in the domain of software engineering research and have, under proper cir-

cumstances, been demonstrated to be representative of the actual program faults [158]. Further threats related to mutation testing include the potential bias introduced by equivalent and trivial mutants. In the context of our experimental settings, equivalent mutants will not be detected by test cases. As explained in Section 6.2, we ignore all mutants that cannot be detected by test cases. Thus, we believe that this threat is sufficiently mitigated. To answer RQ_1-RQ_3, we randomly selected 500 faults (100 groups and five faults per group) for each subject system, which may impact the evaluation of TCP performance. However, this follows the guidelines and methodology of previous studies [306, 192], minimizing this threat. Additionally, we also introduce two research questions, RQ_4 and RQ_5, to investigate the impact of mutant quantities and type on TCP evaluation, allowing us to examine the validity of past evaluations of TCP effectiveness. In addition, there is a potential threat that dues to trivial/subsumed mutants (e.g., the ones that are easily distinguished from the original program) outlined in recent work [133, 234]. The trivial or subsumed mutants may potentially impact the results (e.g., inflate the APFD values). However, we do not specifically control the trivial and subsumed mutants, following the body of previous work in the TCP area [306, 206, 192], since in practice real faults may also be trivial or subsume each other. In addition, we involve a large set of randomly selected mutants, further mitigating this threat to validity. We encourage future studies to further examine this potential threat.

To perform this study we reimplemented eight TCP techniques presented in prior work. It is possible that there may be some slight differences between the original authors' implementations and our own. However, we performed this task closely following the technical details of the prior techniques and set parameters following the guidelines in the original publications. Additionally, the authors of this paper met for and open code review regarding the studied approaches. Furthermore, based on our general findings, we believe our implementations to be accurate.

Threats to External Validity: The main external threat to our study is that we experimented on 58 software systems, which may impact the generalizability of the results.

Involving more subject programs would make it easier to reason about how the studied TCP techniques would perform on software systems of different languages and purposes. However, we chose 58 systems with varying sizes (1.2 KLoC - 83.0 KLoC) and different numbers of detectable mutants (132 - 46,429), which makes for a highly representative set of Java programs, more so than any past study. Additionally, some subjects were used as benchmarks in recent papers [251]. Thus, we believe our study parameters have sufficiently mitigated this threat to a point where useful and actionable conclusions can be drawn in the context of our research questions. In addition, we seeded mutants using operators provided in PIT. It is possible that having different types of operators or using different mutation analysis tool may impact the results of our study. However, PIT is one of the most popular mutation analysis tools and has been widely used in software testing research. Thus, we believe our design of the study has mitigated this threat. Finally, while it would be interesting to investigate the effectiveness of TCPs on detecting real regression faults, this is a difficult task. A large set of real regression faults is notoriously hard to collect in practice. The reason is that during real-world software development, the developers usually run regression tests before committing new revisions to the code repositories, and will fix any regression faults before the commits, leaving few real regression faults recorded in the code repositories. On the other hand, mutants have been shown to be suitable for simulating real faults for software-testing experimentation [158, 19] (including test-prioritization experimentation [81]). Furthermore, mutation testing is widely used in recent TCP research work [192, 133]. Thus, in this paper we evaluate TCP effectiveness in terms of detecting mutants, and leave the investigation of TCP performance on real regression faults as future work.

Finally, we selected four static TCP techniques to experiment with in our empirical study. There are some other recent works proposing static TCP techniques [25, 251], but we focus only on those which do not require additional inputs, such as code changes or requirements in this empirical study. Also, we only compared the static techniques with four state-of-art dynamic TCP techniques with statement-level coverage. We do not study

the potential impact of different coverage granularities on dynamic TCPs. However, these four techniques are highly representative of dynamic techniques and have been widely used in TCP evaluation [192, 247, 82], and statement-level coverage has been shown to be *at least as effective* as other coverage types [192].

5.5 Lessons Learned

In this section we comment on the lessons learned from this study and their potential impact on future research:

Lesson 1: Our study illustrates that different test granularities impact the effectiveness of TCP techniques, and that the finer, method-level, granularity achieves better performance in terms of APFD and APFDc, detecting regression faults more quickly. This finding should encourage researchers and practitioners to use method-level granularity, and perhaps explore even finer granularities for regression test-case prioritization. Additionally, researchers should evaluate their newly proposed approaches on different test granularities to better understand the effectiveness of new approaches. Moreover, APFDc values are relatively less consistent with APFD values at test-method level and vary more dramatically as compared to AFPD values. This suggests that researchers should evaluate the novel TCP approaches in terms of different types of metrics to better investigate the effectiveness of the novel approaches.

Lesson 2: The performance of different TCPs varies across different subject programs. One technique may perform better on some subjects but perform worse on other subjects. For example, TP$_{topic}$ performs better than TP$_{cg-add}$ on $webbit$, but performs worse than TP$_{cg-add}$ on wsc. This finding suggests that the characteristics of each subject are important to finding suitable TCPs. Furthermore, we find that the selection of subject programs and the selection of implementation tools may carry a large impact regarding the results of the evaluation for TCPs (e.g., there can be large variance in the performance of different techniques depending on the subject, particularly for static approaches). This

finding illustrates that the researchers need to evaluate their newly proposed techniques on a *large* set of real subject programs to make their evaluation reliable. To facilitate this we provide links to download our subject programs and data at [195]. Additionally, a potential avenue for future research may be an adaptive TCP technique that is able to analyze certain characteristics of a subject program (e.g., complexity, test suite size, libraries used) and modify the prioritization technique to achieve peak performance.

Lesson 3: Our study demonstrates that while TCP techniques tend to perform better on larger programs, subject size does not significantly impact comparative measures of APFD and APFDc between TCP techniques. Thus, when the performance of TCP techniques are compared against each other on either large or small programs, similar results can be expected. This finding illustrates scalability of various TCP techniques. Also, our experimental results show that software evolution does not have clear impact on comparative TCP effectiveness.

Lesson 4: Our study demonstrates that mutant quantity and type selected in the experimental settings for measuring the effectiveness of TCP techniques does not dramatically impact the results in terms of APFD or APFDc metrics. This finding provides practical guidelines for researchers, confirming the comparative validity of the mutant seeding process of prior TCP work, and also provides evidence that the fault quantity and type factors are less important to investigate in future work.

Lesson 5: Our findings illustrate that the studied static and dynamic TCP techniques agree on only a small number of found faults for the top ranked test-methods and classes ranked by the techniques, and the most highly prioritized test cases by different TCP techniques share similar capabilities in detecting different types of mutation faults. This suggests several relevant avenues for future research. For instance, (i) it may be useful to investigate specific TCP techniques to detect important faults faster when considering the fault severity/importance [87, 269, 165] instead of fault types (e.g., different mutant types) during testing; (ii) differing TCP techniques could be used to target specific types of faults or even faults in specific locations of a program; and (iii) static and dynamic information

could *potentially* be combined in order to achieve higher levels of effectiveness. Furthermore, the similarity study performed in this paper has not been a core part of many TCP evaluations, and we assert that such an analysis should be encouraged moving forward. While APFD and APFDc provide a clear picture of the relative effectiveness of techniques, it cannot effectively illustrate *the difference set* of detected faults between two techniques. This is a critical piece of information when attempting to understand new techniques and how they relate to existing research.

5.6 Conclusion and Discussion

In this work, we perform an extensive study empirically comparing the effectiveness, efficiency, and similarity of detected faults for static and dynamic TCP techniques on 58 real-world Java programs mined from GitHub. The experiments were conducted at both test-method and test-class levels to understand the impact of different test granularities on the effectiveness of TCP techniques. The results indicate that the studied static techniques tend to outperform the studied dynamic techniques at the test-class level in terms of both APFD and APFDc metrics, whereas dynamic techniques tend to outperform the static techniques at test-method level in terms of APFD. APFDc values are generally consistent with APFD values at test-class level but relatively less consistent with APFDc at test-method level. In addition, APFDc values vary more dramatically across all subject programs as compared to APFD values. We also observed that subject size, software evolution, and mutant quantities and types within each faulty group/version do not significantly impact comparative measures of TCP effectiveness.Additionally, we found that the faults uncovered by static and dynamic techniques for the highest prioritized test cases uncover mostly dissimilar faults, which suggests promising avenues for future work. Finally, we found evidence suggesting that different TCP techniques tend to perform differently across subject programs, which suggests that certain program characteristics may be important when considering which type of TCP technique to use.

5.7 Bibliographical Notes

The empirical study presented in this chapter is published in one conference paper [193]. The extended version of this study is currently under journal review. These two papers are shown as follows:

- **Qi Luo**, Kevin Moran, and Denys Poshyvanyk. "A large-scale empirical comparison of static and dynamic test case prioritization techniques." In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (FSE), pp. 559-570. ACM, 2016.

- **Qi Luo**, Kevin Moran, Lingming Zhang and Denys Poshyvanyk. "How Do Static and Dynamic Test Case Prioritization Techniques Perform on Modern Software Systems? An Extensive Study on GitHub Projects" *IEEE Transaction on Software Engineering* (TSE) 2018.

Chapter 6

Assessing Test Case Prioritization on Mutants and Real Faults

Regression Testing is defined as the process of running a collection of compact tests, aimed at testing discrete functionality that underlies a software program, when that program changes. This type of testing allows for the discovery of software *regressions*, faults that cause an existing feature to cease functioning as expected. Such test suites tend to be large for complex projects and are often run every time code is checked into a repository, leading to longer than desired testing times in practice. For example, Google has reported that, across its code bases, there are more than twenty code changes per minute, with 50% of the files changing per month, leading to long testing times [245, 74]. While this may constitute an extreme example, the practice of running unit test suites upon code check-ins has been popularized by emerging continuous integration frameworks, and doing this in an efficient and effective manner is important for assuring the agility of projects. One potential solution to this problem is a set of approaches known as *Test Case Prioritization* techniques which aim to prioritize regression test cases in a test suite to detect regressions more quickly or reduce testing time.

A large body of research has been dedicated to designing and evaluating TCP techniques [25, 55, 68, 79, 80, 164, 243, 274, 293, 294, 275, 51, 214, 186]. This work en-

compasses dynamic techniques that prioritize test cases using code-coverage as a proxy for testing effectiveness [247, 145, 184], and static techniques that typically prioritize according to test case diversity or static call-graph coverage [178, 266, 311]. While the type of prioritization algorithm may differ across techniques, the evaluation metric for TCP techniques, the Average Percentage of Faults Detected (APFD) [247, 145], and its cost cognizant version APFDc [96, 90], are widely used to compare the overall effectiveness and efficiency of these techniques. APFD is a normalized measure of how quickly a TCP technique is able to detect a known set of faults (with techniques that uncover more faults with the highest prioritized test cases having higher APFD values), whereas APFDc further considers both test case execution costs and fault severity.

In order for the APFD(c) (*i.e.,* APFD and APFDc) metrics to measure the practical applicability of TCP techniques in a realistic setting, a reasonably large known set of *real faults* is required. Unfortunately, such fault sets may not be readily available for subject programs that TCP techniques would be evaluated on [157, 254, 158]. Thus, recent TCP research utilizing these performance metrics uses artificial faults, called *mutants*, each comprised of a simple syntactic change to the source code [133, 192, 124, 306, 81]. That is, researchers commonly perform mutation analysis to obtain a set of mutant programs and then evaluate TCP techniques by measuring how effective the prioritized set of test cases are at detecting the injected mutation faults. The use of mutants in the evaluation of TCP approaches gives rise to potential threats of validity impacting the practicality and applicability of the approaches in practice, where they would be used to detect *real faults*. In essence, the underlying assumption of such evaluations is that there is a strong correlation between prioritized sets of test cases that kill high numbers of mutants and sets that detect a high number of real faults. This assumption raises key questions: *How well do TCP techniques perform on real faults?*; *Is the performance of TCP techniques on mutation faults representative of their performance on real faults? What properties of mutants affect the representativeness of this performance?*

Previous studies have examined the relationship between real faults and mutation

faults in order to understand the applicability of mutants in software testing. In particular, these studies have investigated: (i) whether mutants are as difficult to detect as real faults [21, 23]; (ii) whether mutant detection correlates with real fault detection [69, 158]; (iii) whether mutants can be used to guide test case generation [254]; and (iv) whether tokens contained in patches for real-world faults can be expressed in terms of mutants [111].

Despite the studies outlined above, there is a clear gap in the existing literature concerning the applicability of mutants to evaluate TCP techniques. This gap can be summarized as follows: First, no previous study has investigated whether mutation faults are representative of real faults when evaluating TCP approaches. Indeed, such evaluations aim to measure the *rate* at which large sets of mutation faults are detected by *prioritized sets of test cases* according to APFD(c), fundamentally differing from the experimental parameters of the studies outlined above. For example, Just *et al.* [158] focus on the relationship between real faults and mutants measured by the ability of fault detection for a *whole test suite*, which may not imply a similar relationship in terms of APFD(c) values. Second, TCP approaches have not been previously evaluated in terms of their capability of detecting real-world faults, indicating that the true, practical performance of these techniques is largely unknown. Third, prior work has not thoroughly examined the impact of differing fault properties, such as operator types or mutant coupling levels, on the evaluation of TCP techniques. It is clear that analyzing how TCP techniques perform when applied to real faults and investigating the extent to which performance on mutation faults is representative of performance on real faults would help better illustrate the practicality of such approaches and aid researchers in evaluating TCP techniques more comprehensively. Furthermore, studying the impact that fault properties have on TCP performance would shed light on how mutation analysis can be properly applied to best represent performance on real faults.

To address this gap, we perform an extensive empirical study to understand the effectiveness of TCP techniques when evaluated in terms of real faults, and examine whether

mutation faults are representative of real faults when evaluating TCP performance. We implemented eight well-studied TCP techniques and applied them on (i) a dataset of real faults, Defects4J [157], containing 357 real faults from five large Java programs, and (ii) over 35k+ mutants seeded using the Pit [239] mutation testing tool. In the course of this study, we examine the performance and correlation, in terms of APFD(c), across real and mutant fault sets with and without controlling for trivial and subsumed mutants. We further examine how properties of faults, including mutant coupling and operator type, impact performance and the representativeness of mutants.

The results of this study bear several significant findings, from which we can derive best practices for future TCP evaluations and important directions for future work related to mutation testing. First, our results demonstrate that for the five subject programs studied, mutation-based performance of TCP techniques, as measured by APFD(c), tend to *overestimate* performance when compared to performance on real faults *unless trivial and subsumed mutants are removed*. When trivial and subsumed mutants are controlled for, the resulting mutants tend to *underestimate* performance compared to real faults. However, these findings tend to vary depending on the subject program. Furthermore, we find that, as a whole, static TCP techniques tend to outperform dynamic techniques on real faults, *contradicting* results from previous studies that use *only* mutation-based evaluation metrics. Second, we have found that the mutation-based APFDc metric exhibits a stronger positive correlation with real-fault performance than APFD; implying that APFDc, which considers test execution costs, may better illustrate real-world TCP performance. Third, when examining the fault sets in terms of mutant coupling and operator type, we found that the representativeness of mutants (compared to real faults) varies across different types of faults *within* programs. This suggests that different combinations of mutants could be derived to more closely resemble the real faults that are likely to surface in a program. Fourth, we found that the correlation of TCP performance between real and mutation faults differed *across* subject programs. Thus, indicating that in the context of TCP, the mutants used were more representative of real faults for some studied subjects

than for others. This result advocates that future TCP evaluation techniques should move toward deriving specific sets of mutation operators using data driven approaches based on the software domain.

To the best of tour knowledge, this is the first comprehensive empirical study that evaluates the performance of eight well-studied TCP techniques on a large set of *real faults* and compares the results to mutation-based performance in order to determine whether mutation faults are representative of real faults in TCP domain.

6.1 Background & Related Work

In this section we formally define the TCP problem, introduce the studied TCP techniques, and discuss the related work.

6.1.1 TCP Problem Formulation

TCP is formally described by Rothermel *et al.* [249] as finding a prioritized set of test cases $T' \in P(T)$, such that $\forall'', T'' \in P(T) \wedge T'' \neq T' \Rightarrow f(T') \geq f(T'')$, where $P(T)$ refers to the set of permutations of a given test suite T, and f refers to a function from $P(T)$ to real numbers. While there are many types of existing TCP techniques [83, 86, 124, 165, 260, 307], one common dichotomous classification, *static* [141, 311, 178] and *dynamic* techniques [82, 89, 93, 172, 224, 259, 271, 306], relates to the type of information used to perform the prioritization. Static approaches utilize information extracted from source and test code and dynamic techniques rely on information collected at runtime (*e.g.,* coverage information per test case) to prioritize test cases [193]. Additional classifications exist, such as the distinction between *white-box* and *black-box* techniques [133]. The techniques that require source code of subject programs (or information extracted from source code) are typically classified as *white-box* techniques [163]. Conversely, those that only require program input or output information are classified as *black-box* techniques. Approaches that only require test-code have been classified as black-box [193, 133], however, in this

paper we more accurately refer to these techniques as *grey-box* since they require access to test code with references to the underlying program. There are also other approaches [251, 144, 269] that use "non-traditional" information to perform the prioritization, such as code-changes and requirements, that do not fall neatly into these categories.

6.1.2 Studied TCP Techniques

In the context of our empirical investigation, we selected well-studied white and grey-box techniques that utilize "traditional" input information (*i.e.,* source code and test code for static techniques and coverage information for dynamic techniques). We make this decision for the following two reasons. First, "non-traditional" information, such as code changes or high-level requirements, are not always available for subject programs, which can limit the applicability of these approaches in certain contexts. Second, choosing well-studied and understood techniques allows for a more applicable comparison of our results to prior studies. Together we consider four dynamic white-box techniques that utilize run-time code coverage for prioritization, two static gray-box techniques that operate only on test code, and two static white-box approaches that use call-graph information. The four state-of-the-art dynamic TCP techniques include: (i-ii) greedy TCP (with total and additional strategies) [247], (iii) adaptive random TCP (ART) [145], and (iv) search-based TCPs [184]). The static techniques include: (i-ii) call-graph-based (with total and additional strategies) [311], (iii) string-based [178], and (iv) topic-model-based TCPs [266]). Details of the studied TCP techniques are presented in Chapter 5 and the details regarding our implementation of these techniques are in Section 6.2.4.

6.1.3 Threats to the Validity of Mutation-Based TCP Performance Evaluations

There is a large body of work that has addressed the problem of Test Case Prioritization [251, 144, 286, 92, 206]. The common link in the evaluations of these various techniques

is the utilization of *fault-detection rates*, typically in terms of the APFD(c) metrics [267, 133, 192]. However, due to the fact that finding and extracting real-world faults is an intellectually intensive and laborious task, real faults are rarely used when evaluating testing related research [158, 157], including existing work on TCP. Instead, *mutation analysis* can be utilized in calculating these fault-detection rates. During mutation analysis small, automatically-generated syntactic faults are seeded throughout subject programs according to a set of well-formed *mutation operators*, then APFD(c) values are calculated according to the number of mutants that are killed by prioritized test cases. However, to make results of such an evaluation generalizable in a realistic setting, the APFD(c) fault-detection rates should correlate with detection rates of *real faults*. Unfortunately, in the context of the typical methodology used to evaluate TCP techniques, the relation between performance on mutation faults and real faults is not well understood. This gives rise to the potential for threats to the validity of these evaluations relating to (i) the overall performance of TCP techniques on mutation vs. real faults (is performance over or under-estimated?); (ii) the correlation of performance on real vs. mutation faults (does a mutation based-analysis properly illustrate the most effective technique on real faults?); and (iii) the impact that different properties of mutants have on mutation-based performance of TCP techniques.

6.1.4 Studies Examining the Relationship Between Mutants and Real Faults

While our study is the first to examine the representativeness of mutation faults in terms of real faults as it pertains to the domain of TCP, we are not the first to investigate this relationship in a general sense. A small but growing number of studies have been dedicated to understanding the interconnection between mutants and real faults in the broader domain of software testing [21, 23, 69, 157, 254, 158, 57]. Daran and Thévenod-Fosse [69] performed the first empirical comparison between mutants and real faults, finding that the set of errors and failures they produced with a given test suite are quite similar. Andrews *et al.* [21, 23] compared the fault detection capability of test-suites on mutants, real-faults,

and hand-seeded faults, reaching two conclusions. First, mutants (if carefully selected) can provide a good indication of a test suite's ability to detect real faults. Second, the use of *hand-seeded* faults can produce an underestimation of a test suite's fault detection capability. Gopinath *et al.* conducted an empirical study that explored the characteristics of a large set of changes and bug-fixes and how these related to mutants [111]. The authors performed a statistical analysis on the distributions of tokens from these extracted faults and compared them to tokenized faults seeded by traditional mutation operators. This study concluded that the tokenization of a typical real fault is generally not equivalent to seeded mutation faults.

Just *et al.* studied whether a test suite's ability to detect mutants is coupled with its ability to detect real faults, controlling for code-coverage [158]. Their results indicate that mutant detection correlates more closely with real fault detection than with code coverage. Additionally, their study also provided suggestions regarding how mutant taxonomies can be improved to make them more representative of real faults through examination of how mutants are coupled to real faults. Just *et al.* introduced a valuable dataset of 357 real faults across five Java programs in an artifact called Defects4J [157], which we utilize in this paper. Shamshiri *et al.* conducted an empirical study of automatic test generation techniques to investigate their ability to detect real faults in the Defects4J [254]. Finally, Chekham *et al.* conducted a study examining how mutation, statement and branch coverage correlate to fault revelation [57]. They found that *strong* mutation testing has the highest fault revelation capability of these coverage criterion, and fault revelation only tends to significantly increase once high coverage levels are attained.

While the aforementioned research has investigated several aspects of the relationship between real-faults and mutants, there is no prior work examining the relationship between real faults and mutants in the context of a typical TCP evaluation methodology. Thus, it is unclear whether the performance assessment and comparison of TCP techniques, when performed on mutants, is valid when one considers real faults. While prior studies that examine the relationship between real faults and mutants more generally, it is

unclear whether results from these studies hold in the context of TCP, as the experimental settings in such a case (and hence in our study) *fundamentally differ* from past work. More specifically, we re-create the typical evaluation methodology used to assess the effectiveness of TCP techniques, which involves seeding mutants into a *single* version of a program and calculating the APFD(c) metrics. We then compare such metrics computed using mutants with those computed over real faults, and examine their correlation and overall utility in assessing TCP performance.

Prior work largely ignores how the performance of these techniques may vary across mutants with different levels of coupling to real faults or across specific types of mutation operators. Thus, we investigate whether different levels of mutation coupling or particular types of mutation operators impact effectiveness measures for mutants or correlation with real faults. Furthermore, we consider the test execution cost of prioritized test cases (APFDc) which past studies have not considered. To summarize, this study fills several existing gaps, both in TCP research, and regarding the relationship between real-faults and mutants. This leads to several important, unexpected findings, impacting future work in both areas.

6.2 Empirical Study

The *goal* of this study is to analyze the extent to which mutation analysis can support Test Case Prioritization (TCP), as opposed to using data from real faults. The study *context* consists of data from five Java open source projects (Defect4J [158, 157]), mutants generated by the PIT [239], and eight TCPs described in Section 6.1.

6.2.1 Research Questions (RQs):

The study aims at answering the following three **RQs**:

RQ$_1$: How *effective* are TCP techniques when applied to detecting real faults?

Table 6.1: The stats of the subject programs: #Real: #real-world faults; #All: #all mutation faults; #Detected: #mutation faults can be detected by test cases; #Subsuming: subsuming mutants.

Subject Programs	#Real	#Detected	#Subsuming	#All
JFreeChart	26	32,790	1,796	102,629
Closure Compiler	133	82,572	9,731	111,826
Commons Maths	106	80,059	5,016	113,680
Joda-Time	27	24,555	3,066	34,147
Commons Lang	65	25,173	2,129	31,214
Total	357	245,767	21,738	393,496

RQ$_2$: Is the performance of TCP techniques on mutants *representative* of performance on real faults?

RQ$_3$: How do the *properties* of real faults and mutants affect the performance of TCP techniques?

6.2.2 Study Context

In order to properly evaluate the performance of TCP approaches when applied to detecting real-world faults, our study requires a well understood set of verified, real program faults preferably containing coupling information between real faults and mutants. To satisfy this criteria, we utilize the Defects4J [157] dataset, which contains 357 real faults extracted from five Java subject programs, listed in Table 6.1, and has been utilized in past studies [158, 254]. Defects4J isolates the real faults from the version control and bug tracking system of each subject program. For each isolated fault there exists a faulty program version and a corresponding fixed version. Table 6.1 shows the distribution of isolated real faults and seeded mutation faults across the five subject programs, with Closure and Commons Maths representing the largest portion of real faults.

For each real fault (*i.e.,* faulty version), Defects4J provides a test suite including at least one test case that is able to trigger the fault but pass successfully in the corresponding fixed version. Additionally, it provides the code locations (*i.e.,* method and class names) that were modified to fix the fault. This facilitates qualitative analysis and root

cause determination for the set of real faults. Since a comprehensive test suite is required for the proper evaluation of TCP approaches, we opted to use the existing JUnit test cases provided for each (faulty or fixed) program version in Defect4J. Furthermore, test cases were extracted at test-method granularity rather than the test-class granularity, as TCP techniques have been shown to perform best under such experimental settings [193].

The *primary goal* of this study is to determine how well mutation-based analysis reflects the performance of TCP techniques on real faults. More generally, we aim to answer the following question: "If one prioritizes test cases using mutants, would the this prioritized set likely be as effective on real faults?" In order to properly explore this question we seeded mutants across the *fixed* versions of all subject systems using the PIT mutation tool [239] with all built-in operators enabled. We exclude the mutants which are not killed (*i.e.,* triggered the test case to fail), by any test cases in the existing JUnit test suites for two reasons: i) to mitigate a potential threat to validity from equivalent mutants, ii) based on Equations 6.1 and 6.2, these mutants will not affect the APFD(c) values. The number of detected mutants and the total number of seeded mutants are shown in columns 3 and 5 of Table 6.1 respectively (see our attached appendix for more detailed results). Furthermore, two recent works outline the potential impact of trivial/subsumed mutants for mutation-based analysis [234, 133]. Thus, we also compute results when trivial and subsumed mutants are removed from the original set of seeded mutants. We follow the methodology defined in prior work [234], which is the best approximation for the removal of subsuming mutants, as this has been proven an undecidable problem. The number of subusming mutants is shown in column 4 of Table 6.1. Thus we will discuss results in terms of two different *sets* of mutants: the *full mutant set*, and the *subsuming mutant set*. In this study, since the aim is to assess the performance of the *methodology* used in evaluating TCP techniques, we performed our mutant seeding in accordance with past studies [193, 192, 306], applying mutation analysis to the single, most recent *fixed* version of each subject program.

To perform a comparison between mutants and real faults, for each (real) faulty version of a program in Defects4J, we create one mutated program instance, with a randomly selected mutant. However, given that the selected mutant is a random variable, we repeat this process 100 times for each real fault to provide for a reliable statistical analysis and the best possible approximation for TCP evaluations from prior work (*e.g.,* for Closure: 133 versions with real faults $\times$ 1 mutant $\times$ 100 instances = 13,300 total mutants). For instance, taking JFreeChart program as an example, one mutant was randomly selected from the set of 32,790 mutants able to be detected by at least one test case, until a set of 26 mutant versions of JFreeChart were accumulated (matching the number of real faulty versions). This procedure is then repeated 100 times. This results in 100 groups of 26 mutants, or 2,600 mutants being evaluated for JFreeChart in total. In initial experiments, excluded due to space limitations, we computed the APFD(c) values using 5 randomly seeded mutants per instance (instead of one), following the settings of previous work [124, 178, 192, 193, 206, 305]. The results for this analysis generally agree with the presented results, and thus we do not expect that number of mutants per instance will dramatically impact the results. The intention behind choosing these experimental settings is to evaluate whether past mutant-based methodologies measuring TCP efficacy would hold for real faults.

6.2.3 Methodology

In this section we describe the experimental methodology used to answer our proposed research questions.

Table 6.2: Studied TCP Techniques.

Type	Tag	Description
Static	TCP_{cg-tot}	Call-graph-based (total strategy)
	TCP_{cg-add}	Call-graph-based (additional strategy)
	TCP_{str}	The string-distance-based
	TCP_{topic}	Topic-model-based
Dynamic	TCP_{total}	Greedy total (statement-level)
	TCP_{add}	Greedy additional (statement-level)
	TCP_{art}	Adaptive random (statement-level)
	TCP_{search}	Search-based (statement-level)

6.2.3.1 RQ$_1$: TCP Effectiveness on Real Faults

The *goal* of this research question is to investigate the performance of TCP techniques when they are applied to detect real faults. We first ran these eight TCP techniques on 357 program versions containing real faults to obtain ranked lists of test cases for each faulty version. The tests are run at the *test-method* level, since past work has shown method-level yields more effective TCP results [192, 193]. Then, to measure the effectiveness in terms of fault detection for each studied technique, we calculated two well-accepted metrics, the Average Percentage of Faults Detected (APFD) [247, 92] and its cost cognizant counterpart APFDc[96, 90]. Formally, APFD is defined as follows: Let T be a test suite and T' is a permutation of T, the APFD value for T' is given by

$$APFD = 1 - \frac{\sum_{i=1}^{m} TF_i}{n * m} + \frac{1}{2n} \tag{6.1}$$

where n is the number of test cases in T, m is the number of faults, and TF_i is the position of the first test case in T' that detects fault i. Intuitively, the higher the APFD value, the higher the rate of fault detection by the prioritized test cases. In order to derive a more holistic understanding of the relationship between TCP performance on real faults and mutants we also consider APFDc. This metric takes both execution cost and fault severity into account. Since there is no clearly-defined nor widely-used method for estimating fault severity, we consider severity to be the same for all faults. Therefore, in the context of this study APFDc reduces to the following formal definition:

$$APFDc = \frac{\sum_{i=1}^{m}(\sum_{j=TF_i}^{n} t_j - \frac{1}{2}t_{TF_i})}{\sum_{j=1}^{n} t_j * m} \tag{6.2}$$

where t_j is the execution cost for the test case ranked at position j in the ranked test suite. Intuitively, APFDc, as defined above, will be higher for prioritized test suites that both find faults faster and require less execution time. Since we study five subjects, each with a different number of real faults, we computed the average APFD(c) values of the

different versions across all five systems to understand the effectiveness of each studied approach. Additionally, to statistically analyze the differences between TCP techniques in terms of APFD and APFDc values, we perform an Analysis of Variance (ANOVA) and a Tukey Honest Significant Difference (HSD) test [265] on the average APFD(c) values across the five subjects. The ANOVA analysis is used to test whether there are statistically significant differences between the performance of TCP techniques when applied to real faults versus when applied to mutants. The Tukey HSD test classifies the TCP techniques into different groups based on their performance in terms of APFD(c) values, further illustrating the relationships between them. For both statistical procedures we consider a significance level $\alpha = 0.05$.

6.2.3.2 RQ$_2$: Representativeness of Mutants

The *goal* of RQ$_2$ is to understand whether mutants are representative of real faults in the evaluation of TCP techniques. Thus, we applied mutation analysis according to the description given in Section 6.2.2. As mentioned in Section 6.2.2, two sets of mutants are examined, the *full mutant set* and the *subsuming mutant set*. Then, we ran all studied TCP techniques on these mutant versions and calculated the average APFD(c) values (see Equations 6.1 and 6.2) across all 100 mutant groups and across all five subject programs, in order to examine the mutant-based performance of our studied TCP techniques.

At this point, we are able to evaluate the effectiveness of TCPs in terms of both real fault and mutant detection according to APFD(c). In order to compare these metrics, we rely on the Kendall rank correlation coefficient τ [167] to measure the relationship. This correlation metric is commonly used to measure the ordinal association between two quantities [166] and has been widely used in the area of software testing for such purpose [107, 235, 305]. Consider the APFD values of real fault detection and mutation fault detection across all studied techniques as a set of pairs (R, M), where R is the APFD/APFDc values of real fault detection and M is the APFD/APFDc values of mutation

fault detection. Any pair of (r_i, m_i) and (r_j, m_j) (APFD values for TCP$_i$), where $i \neq j$, are concordant if $r_i > r_j$ and $m_i > r_j$ or if $r_i < r_j$ and $m_i < m_j$. They are discordant if $r_i > r_j$ and $m_i < m_j$ or if $r_i < r_j$ and $m_i > m_j$ [223]. The Kendall τ rank correlation coefficient is formally defined as the ratio of the number of concordant pairs less the number of discordant pairs and the total number of pairs. Thus, its value ranges from -1.0 to 1.0. Results closer to 1.0 indicate the observations of two variables have similar rank (*e.g.,* which in the context of this study translates to similar rates of fault discovery), whereas when it is closer to -1.0 when the observations of two variables have dissimilar ranks (*e.g.,* suggesting a negative correlation between fault discovery rates). The two variables are independent when the value approximates to 0.0. Following the previous work [107], we chose Kendall τ_b statistic since it makes adjustments for ties and does not require a linear relationship.

6.2.3.3 RQ$_3$: Effects of Fault Properties

The *goal* of this research question is to understand how different *properties* of faults impact two phenomena in the context of TCP: (i) the performance of prioritization techniques, and (ii) the representativeness of mutants as a proxy for real faults (*i.e.,* performance correlation).

The *first* fault property we investigated is **the level of coupling between real faults and mutants**. In order to determine the level of coupling for real faults to mutation operators, we utilize Just *et al.*'s previous work [158], which classified the 357 real faults from the Defects4J dataset into four main coupling levels: (i) those coupled with mutants (denoted in the study using the keyword “Couple”), (ii) those requiring stronger mutation operators (denoted as “StrongerOP”), (iii) those requiring new mutation operators (denoted as “newOP”), and (iv) and those not coupled with mutants (denoted as “Limitation”) Formally speaking, a real fault (*i.e.,* a complex fault) is coupled with a set of mutants(*i.e.,* simple faults) if a test case that detects all the mutants also detects the real fault [158].

We contacted the authors to obtain this classification scheme, which includes 262 real faults coupled to mutants, 25 real faults requiring stronger mutation operators, seven real faults requiring new mutation operators, and 63 real faults not coupled to mutants. In order to examine the impact that fault coupling has on *performance*, pruning our initial dataset from the previous two research questions, we calculated APFD(c) values for each coupling level of real faults and for all mutants considered in RQ$_2$ for each subject program. In order to examine the *correlation* between these APFD(c) values, we again utilize Kendall's τ_b coefficient.

The *second* fault property we examined is the **mutation operator type**. Intuitively, this investigation should help shed light on which mutation operators are more representative of real faults in the context of TCP performance. We classified mutation faults based on their corresponding operators, that is, the mutation faults that are generated by the same mutation operator are classified into the same group. We consider the 15 built-in operators in PIT [239], and for each subject program classified all mutant versions into groups according to these operators. For each subject program and operator type, we then randomly sampled from these groups until we had a set of faults corresponding to the number of real faults existent in each subject program respectively. We then repeat this process 100 times. If there are not enough mutants to create 100 groups, we repeat this process until we exhaust the mutants. In the end, for each subject program, we derive 100 mutant groups for each type of operator (given enough mutants), with each group containing the same number of mutants (all of the same operator type) as real-faults for the subject. To understand the impact that different types of mutation operators have on the *performance* of TCP techniques, we calculated the APFD(c) values based on the new groups of mutants, and then used the Kendall τ_b coefficient to understand the *correlation* between the APFD(c) values calculated in terms of real faults and mutants. We also explored the effects of mutant locations, however, we found no significant trends or correlations. Thus, we forgo discussion of these results in this paper and point interested readers to our attached appendix.

Table 6.3: Average APFD & APFDc values for all eight TCP techniques, for both real, mutation fault and subsuming mutation fault detection, across all subject programs. Additionally, the grouping results for the Tukey HSD test are shown in capitalized letters (*e.g.,* AB). S.Mutants refers to subsuming mutants.

Faults	TCP$_{cg-tot}$		TCP$_{cg-add}$		TCP$_{str}$		TCP$_{topic}$		TCP$_{total}$		TCP$_{add}$		TCP$_{art}$		TCP$_{search}$	
	APFD	APFDc	APFD	APFDc	APFD	APFDc	APFD	APFDc	APFD	APFDc	APFD	APFDc	APFD	APFDc	APFD	APFDc
Real	0.594	0.480	0.597	0.591	0.696	0.594	0.7	0.635	0.61	0.419	0.583	0.454	0.657	0.677	0.6	0.556
	A	BC	A	ABC	A	ABC	A	AB	A	C	A	C	A	A	A	ABC
Mutant	0.743	0.598	0.818	0.835	0.834	0.788	0.832	0.802	0.757	0.549	0.897	0.829	0.8	0.841	0.784	0.725
	B	BC	AB	A	AB	AB	AB	A	B	C	A	A	AB	A	B	ABC
S.Mutant	0.561	0.407	0.612	0.639	0.620	0.572	0.612	0.570	0.534	0.305	0.664	0.565	0.622	0.671	0.578	0.508
	AB	BC	AB	A	AB	AB	AB	AB	B	C	A	AB	AB	A	AB	ABC

6.2.4 Experiment Tools and Hardware

6.2.4.1 Mutation Analysis

We applied PIT [239] on the latest *fixed* (*i.e.,* non-faulty) program version across the five subjects to perform mutation analysis. PIT is a mutation testing system, which is able to generate mutants for Java programs and run JUnit test cases automatically on the mutant versions to obtain a set of killed/survived test cases and coverage information.

6.2.4.2 Implementation of TCP Techniques

We reimplemented all studied TCPs in accordance with the technical descriptions in their corresponding papers (see Sec. 6.1). Three of the authors, and an external expert on TCP, carefully reviewed the source code, ensuring the reimplementation is reliable. To collect coverage information, we used the ASM bytecode manipulation and analysis toolset [34]. In our empirical study, we chose to use statement-level coverage information, which produces statements covered by each test method, as this allows for optimal performance of TCP techniques [193]. Furthermore, we utilize JDT [143] to extract textual information for each test method, which is used by string-based and topic-based approaches. Specifically for the topic-based approach, we use Mallet [202] to build an LDA topic model [177] for each test case, after pre-processing the textual information (*e.g.,* splitting, removing stop words and stemming). Following previous research [193], we use WALA [270] to build RTA call graphs [118] for each test method and traverse each call graph to obtain its static coverage in order to implement the TCP_{cg} techniques.

6.2.4.3 Hardware

The experiments were carried on eight servers with 16, 3.3 GHz Intel(R) Xeon(R) E5-4627 CPUs, and 512 GB RAM, and one server with eight Intel X5672 CPUs and 192 GB RAM.

6.3 Results

In this section, we describe the results of our empirical study as they relate to the proposed research questions. Additionally, we provide an attached appendix including additional results and figures. We provide a detailed online appendix with a complete dataset of our study results [11].

6.3.1 RQ$_1$: TCP Effectiveness on Real Faults

The values of the APFD(c) metrics for *real* faults are reported at the top of Table 6.3. From these experimental results we make the following observations. First, for real faults, all techniques tend to perform better when measured by APFD as compared to APFDc. This is not surprising, and it is due to the incorporation of execution cost. For some techniques, in particular TCP_{total} and TCP_{cg-tot}, the differences between APFD and APFDc are comparatively larger. This observation is most likely due to the fact that these techniques always prioritize test cases with higher coverage first, leading to longer execution costs for the top test cases. Thus they have comparatively lower APFDc values, compared to APFD values.

Second, somewhat surprisingly, the static TCP techniques perform better overall than dynamic TCP techniques for both metrics. TCP_{topic} performs best in terms of APFD (with a value of 0.700) and overall static techniques outperform dynamic ones (0.646 *avg* vs. 0.613 *avg* respectively). Whereas for APFDc, TCP_{art} performs best (with a value of 0.677), static approaches still outperform dynamic ones (0.575 *avg* vs. 0.526 *avg* respectively). When considering APFD, this finding is of particular interest as it *contradicts* prior studies that examined similar techniques [193, 192, 133] in the context of mutation

Table 6.4: Results of the ANOVA analysis and the Kendall τ_b Coefficient for the overall APFD(c) values shown in Table 6.3.

Faults	ANOVA p-value		τ_b	
	APFD	APFDc	APFD	APFDc
Real	0.011	3.22e-4	-	-
Mutant	8.02e-4	3.77e-5	0.143	0.571
S.Mutant	0.011	1.38e-4	-0.071	0.643

Table 6.5: Results for the Kendall τ_b Rank Correlation Coefficient between APFD(c) values for TCP techniques on detecting mutation faults and detecting each type of real faults described in Section 6.2.3.3.

Real Faults	Chart		Lang		Math		Time		Closure		Mean	
	APFD	APFDc	APFD	APFDc	APFD	APFDc	APFD	APFDc	APFD	APFDc	APFD	APFDc
Couple	0.429	0.786	-0.071	0.286	0.5	0.429	-0.143	0	0.714	0.714	0.2858	0.443
Limitation	-0.214	0.214	-0.214	0.143	0	0.286	-0.071	0.143	0.5	0.286	0.0002	0.2144
StrongerOP	0.214	0.857	0.182	0.327	0	0.357	-0.286	0.5	0.714	0.571	0.1648	0.5224
NewOP	0.109	0.286	-	-	-0.143	0.286	-	-	0.571	0.571	0.179	0.381

faults, which generally conclude that TCP_{add} performs best. This suggests that in general, past mutation-based results may not be indicative of techniques' *practical* performance on real faults.

Third, the TCP_{add} technique does not outperform the TCP_{tot} strategy, again contradicting findings from past studies where the TCP_{add} has been shown to perform best overall [193, 249, 92]. Fourth, the results of the Tukey HSD test suggest that for APFD the performance of the TCP programs do not vary in a statistically significant manner. However, for APFDc, we found statistically significant differences across techniques for statistical tests. However, it should be noted that the results of these tests are derived from a smaller dataset as compared to the traditional method of using thousands of mutation faults, due to the number of faults included in Defects4J.

6.3.2 RQ$_2$: Representativeness of Mutants

The values of the APFD(c) metrics for mutants across the different TCP techniques are also shown in Table 6.3. From this data, we can make several notable observations. First, the APFD and APFDc metrics calculated using the full mutant set generally tend to *overestimate* performance compared to real faults. This finding is relevant, as it implies that mutation-based evaluations measuring the performance of TCP techniques that do

not control for subsumed and trivial mutants tend to overestimate *real-world* applicability of these techniques. Conversely, there is a slight *underestimation* for the APFD(c) values calculated using the subsuming mutant set when compared to real faults. Moreover, as stated in the results of RQ_1, the studied TCP techniques perform differently across different fault sets, with the relative performance of both the full mutant set and subsumed mutant set differing from relative performance on real faults. This is a significant finding as it suggests that, according to results for our set of five subject programs, ***a TCP approach that performs well according to mutation analysis may not exhibit the same performance on a set of real faults for the same program(s)***. This suggests that mutation-based analyses comparing TCP techniques *against each other* on the same set of subjects may not generalize on real faults. As we will discuss later, this points to the need for *careful selection of mutants* when performing a mutation-based evaluation of TCP techniques in order to ensure that the results obtained also hold for *real faults*. In addition, removing subsumed mutants maybe good for practice, otherwise, performance will be greatly overestimated.

While the absolute performance in terms of APFD and APFDc may not be similar when comparing performance on mutants to performance on real faults, it is possible that that performance is *positively correlated* between the two fault sets. That is, better performance on mutation-based faults for TCP techniques, may generally point towards better performance in terms of real faults. To measure this, we examine the Kendall τ_b correlation coefficient across the results from the two types of faults (shown in Table 6.4). Note that two rankings are considered as independent to one another when τ_b is closer to zero. Our results indicate a very weak positive correlation between mutants and real faults when examining APFD (τ_b=0.143 for all-killed mutants and for τ_b=-0.071 subsuming mutants) and a medium to strong positive correlation when considering APFDc (τ_b=0.571 for all-killed mutants and for τ_b=0.643 subsuming mutants). Removing subsumed mutants does not impact the correlation results. This observation implies that, in general, a mutation-based TCP performance evaluation carried out in terms of APFDc will more

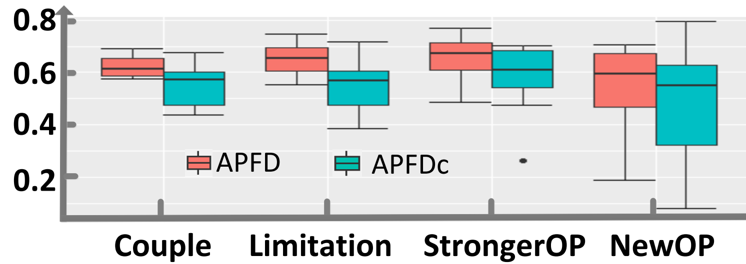


Figure 6.1: APFD(c) values for TCP techniques in terms of detecting different types of real faults.

strongly correlate to performance in terms of APFDc on real faults. However, when relying on a mutation-analysis based APFD evaluation, *as many previous studies do*, there is no guarantee that the results will correlate to similar levels of performance on real faults. However, as we illustrate in the course of answering RQ_3, this correlation tends to vary across both the studied techniques and mutation operators.

6.3.3 RQ_3: Effects of Fault Properties

In this subsection we investigate the effect that different fault properties have on the performance of TCP techniques, and on the relationship between mutants and real faults. As stated earlier, we discuss results for real faults in terms of different *coupling levels*, and mutation faults in terms of *operators*. In the context of $\mathbf{RQ}_3$, we keep all mutants for analysis instead of subsuming mutants, since the size of subsuming mutants is quite small, specially when grouping them based on mutation operators. To make sure the statistical reliability and diversity in terms of mutant types, we show the results for all mutants in this paper. Interested readers could find the results for subsuming mutants in our attached appendix.

6.3.3.1 Effects of Coupling Between Mutants and Real Faults

To investigate the effect that coupling has on performance of TCP across real-faults we consider four different fault types discussed in Section 6.2.3.3. The *performance* results

for the APFD(c) metrics broken down by coupling level are illustrated in Figure 6.1. The *correlation* results for APFD(c) across subjects are given in Table 6.5.

As Figure 6.1 shows, TCP techniques perform differently in terms of detecting different *types* (*e.g.,* coupling levels) of real faults. This result yields a few notable observations. First, TCP techniques tend to perform best (in terms of APFD and APFDc values) on real faults that are classified as needing *stronger operators* to be properly represented by mutants. This finding is encouraging, as it highlights that the studied approaches are capable of prioritization schemes that effectively uncover faults which are *not* closely coupled to mutants. When examining the correlation results, we find that the APFD τ_b coefficient values of *coupled* real faults are, unsurprisingly, substantially higher than for other types of real faults. This implies that TCP performance on real faults, which are more tightly coupled to mutants, is more strongly correlated with performance on mutation faults in TCP evaluations. However, for APFDc we find that real faults requiring stronger operators tend to exhibit the highest correlation. Finally, as Table 6.5 shows, the correlation results vary across different subject programs. For instance, on one hand, the τ_b values for Closure are quite large across all levels of coupling, implying that TCP performance on mutants is more indicative of performance on real faults for this particular subject. On the other hand, the τ_b values for Lang are much closer to zero.

6.3.3.2 Effect of Different Mutation Operators

The *performance* distributions for the APFD(c) metrics across different operators are depicted as box plots in Figure 6.3. The *correlation* results across operators between the performance of mutants and real faults are shown in Table 6.6. The observations that can be made from this data help further explain the results of RQ$_2$. The *performance* results illustrate that the different TCP techniques tend to exhibit slight performance variances across different types of mutation operators, with the the Switch and VoidMethodCall operators trending toward the positive and negative extremes respectively. This result implies that, even if a researcher is performing *only* a mutation-based analysis, the set of mutation

Table 6.6: Results for the Kendall τ_b Rank Correlation Coefficient between APFD(c) values for TCP techniques on detecting real faults and detecting each type of mutation faults.

Mutation Faults	Chart		Lang		Math		Time		Closure		Mean	
	APFD	APFDc	APFD	APFDc	APFD	APFDc	APFD	APFDc	APFD	APFDc	APFD	APFDc
NegateConditionals	0.357	0.857	-0.143	0.143	0.429	0.571	-0.214	0.286	0.643	0.643	**0.214**	**0.500**
RemoveConditional	0.5	0.857	-0.143	0.143	0.429	0.571	-0.214	0.286	0.643	0.643	**0.243**	**0.500**
ConstructorCall	-0.143	0.714	0	0.286	0.5	0.714	-0.214	0.071	0.714	0.5	**0.171**	**0.457**
NonVoidMethodCall	0.214	0.786	0	0.286	0.357	0.5	-0.214	0.286	0.714	0.5	**0.214**	**0.471**
Math	0.286	0.714	-0.286	0.286	0.286	0.5	-0.071	-0.071	0.786	0.643	**0.200**	**0.414**
MemberVariable	0.214	0.929	-0.786	0.357	0.429	0.5	0.286	0.357	0.5	0.357	**0.129**	**0.500**
InlineConstant	0.286	0.929	-0.143	0.286	0.429	0.429	0	0.071	0.786	0.571	**0.272**	**0.456**
Increments	0.214	0.714	-0.214	0.143	0.286	0.571	0	0	0.786	0.714	**0.214**	**0.428**
ArgumentPropagation	0.143	0.857	0	0.214	0.357	0.286	-0.429	0.286	0.643	0.643	**0.143**	**0.457**
ConditionalsBoundary	0.357	0.786	-0.071	0.286	0.429	0.643	0.071	0.214	0.714	0.571	**0.300**	**0.500**
Switch	0.214	0.714	-0.214	-0.071	0.429	0.357	-0.214	0.214	0.786	0.429	**0.200**	**0.329**
VoidMethodCall	-0.143	0.714	-0.214	0.071	0.214	0.571	-0.357	0.357	0.857	0.643	**0.071**	**0.471**
InvertNegs	0.357	0.857	0.143	0.071	0.143	0.5	-0.214	0	0.714	0.714	**0.229**	**0.428**
ReturnVals	0.357	0.857	-0.429	0.357	0.357	0.714	-0.214	0.286	0.786	0.571	**0.171**	**0.557**
RemoveIncrements	0.214	0.786	-0.143	0.071	0.429	0.429	0.143	0.071	0.714	0.714	**0.271**	**0.414**

```
- currentPropertyNames = implicitProto.getOwnPropertyNames();
+ if (implicitProto == null) (
+     currentPropertyNames = ImmutableSet.of();
+ }
+ else {
+     currentPropertyNames = implicitProto.getOwnpropertyNames();
+ }
```

(a) Closure-2 Bug Fix.

```
- System.arraycopy(array2, 0, joinedArray, array2.length, array2.length)
+ try {
+     System.arraycopy(array2, 0, joinedArray, array2.length, array2.length)
+ } catch (ArrayStoreException ase) {
+     ...
+     ...
+     throw ase; // No, so rethrow original
+ }
```

(b) Lang-37 Bug Fix.

Figure 6.2: Examples of bug fixing changes.

operators selected can cause variations in the results. More importantly, the *correlation* results indicate that the degree to which performance on mutation faults correlates to performance on real faults, in terms of APFD(c), *varies dramatically across* systems as a a whole, and across different operator types *within* a single subject. For instance, when examining APFD values for specific systems, we found that mutation-based performance for both Closure and Math exhibits a strong positive correlation to performance on real faults across nearly all mutation operators. At the same time, operators such as ConstructorCall and VoidMethodCall exhibit a negative correlation within Chart, which tends

to have a weak positive correlation overall. Besides differences found in individual programs, overall the mutation-based APFDc metric is more strongly coupled to real faults than APFD, corroborating RQ_2 results.

We wanted to further understand the variance between performance correlations across different subject programs. Intuitively, our findings suggest that the characteristics of a program may influence how representative mutation-analysis based TCP performance would be in terms of real faults. Therefore, we examined some of the bug fixing commits for a subject program that showed a strong correlation (Closure) and for a program that showed negative correlation (Lang). When looking into the fixing commits of these two subject programs, we found that Closure, being a compiler, trends heavily toward complex control flows managed by conditionals, compared to Lang, which trends more towards string and array manipulation. Two illustrative examples of bug fixes are shown in Figure 6.2. The first example for Closure (*i.e.,* Figure 6.2(a)) shows that developers fixed this bug by simply modifying conditionals. In particular, the bug shown in Figure 6.2(a) is exactly the same as one of the PIT mutation operators, *i.e.,* the NonVoidMethodCall mutator. Bug fixing in Lang mostly involved other more complex changes such as adding exception handlers due to the nature of its domain (see Figure 6.2(b)). This investigation suggests that TCP performance correlation between real faults and mutants is low when it is not possible to seed mutants properly reflecting faults occurring in a given domain or a program.

6.4 Threats to Validity

Threats to Internal Validity concern potentially confounding factors of the experiments that might introduce observed effects. One such factor is represented by faults that were seeded into the programs. We chose PIT to perform mutation analysis, which has different types of mutation operators compared to other mutation tools, such as Major [199]. While PIT features many operators common across other tools, it is possible that different

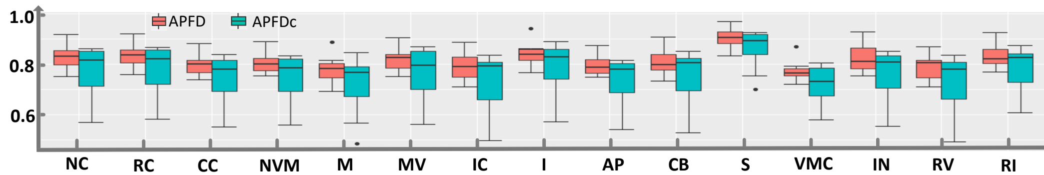


Figure 6.3: Average APFD(c) values across different mutation operators referenced as: NC = NegateConditional, RC = RemoveConditional, CC = ConstructorCall, NVM = NonVoidMethodCall, M = Math, MV = MemberVariable, IC = InlineConstant, I = Increments, AP = ArgumentPropagation, CB = ConditionalsBoundary, S = Switch, VMC = VoidMethodCall, IN = InvertNegs, RV = ReturnVals, and RI = RemoveIncrements.

mutation tools might have led to different observations. We leave exploration of additional tools as future work.

Internal validity threats also arise due to the assumptions made about the validity of the coupling between mutation faults and real faults obtained from Just et. al.'s study [158]. This is because mutants seeded in the same code where the fault occurred may differ from the real fault, hence leading to possible misclassifications. Future research could further examine the affects of real fault-mutant coupling relationships to further mitigate this threat.

Another potential confounding factor is the fact the studied test suites are written by developers. However, these test suites have been shown by past work [158] to generally be of high quality, exhibiting high coverage, mitigating this threat. Additionally, threats may arise due to the difference between test-suites for real faults and test suites of the mutation faults. This is because the mutation faults were seeded into the *latest* version of the subject programs and in order to perform this study, we had to use the test suite corresponding to the faulty/fixed versions for real faults. However, previous studies have shown that the results of mutation-based TCP techniques tend to be similar across program versions [192, 133], mitigating this threat.

Threats to Construct Validity concern the relation between experimental theory/constructs, and potential effects on observed results. As explained in Section 6.2.3, in the context of this study we re-implemented all of the TCP techniques following the approach descriptions in their respective papers. Our re-implementations may differ slightly from the original versions. However, we closely followed the methodology of the previous work,

and three authors and one external TCP expert reviewed the code to ensure a reliable implementation. Executing the studied TCP techniques on all of the program versions was time-consuming, totaling more than five months of computation time. To make the GA-based technique tractable we reduced the maximum number of generations to 50. However, the main goal of this study is to understand the differences between real and mutation faults in TCP evaluation and search-based TCP shares the same settings on both of faulty and fixed versions, thus, this detail should not effect the conclusions drawn from our study.

Threats to External Validity. We limited our focus to eight TCP techniques, which require only source code, test code, and coverage information to perform prioritization. These eight TCP techniques are well-understood and widely used/studied in recent research work [193, 192], and since we aimed to understand how techniques differed from previous studies when applied to real faults, this is a suitable set of techniques to study. We encourage researchers to extend this study to additional TCP approaches.

In order to provide a rigorous experimental procedure, we applied mutation analysis to TCP techniques in particular experimental settings discussed in Sec. 6.2.2 and 6.2.3. Thus, it is possible that these results may differ for different TCP evaluation methodologies. However, we chose the experimental methodology set forth in this paper due to the fact that it has been widely used in previous studies [124, 178, 192, 193, 206, 305] and is likely to be used in the future.

We use the Defects4J dataset to understand the effectiveness of TCP techniques in terms of real-fault detection. It is quite possible that there are different types of faults (varying in complexity) in other subject programs written in other program languages compared to those in Defects4J. However, Defect4J is the largest and most reliable database of real faults publicly available, containing 357 faults extracted from real-world software systems and is commonly used in previous research work [157, 254, 158].

We utilize Pit [239] and hence our results are representative of a certain set of mutants. While Pit utilizes many of the same standard operators as other tools, this study could

be expanded in the future to examine additional mutation testing frameworks. It is also possible that the larger number of mutant versions considered in our study may impact the results of analysis. However, to mitigate this threat, we fixed the number of mutant program versions to analyze whether the mutation faults are representative of real faults.

6.5 Lessons Learned

In this section we summarize the pertinent findings of our study into discrete *learned lessons* and discuss their potential impact on future work in the TCP area.

Lesson 1: *Relative Performance of TCP techniques on mutants may not indicate similar performance on real faults, depending on the subject program.* Our study indicates that, for the subject programs studied, the relative performance of TCP techniques (following the popular methodology utilized in our experiments) is not similar between mutants and real faults. This indicates that a technique which outperformed competing techniques under the experimental setting of mutation analysis may not achieve similar relative performance on real faults. This illustrates a potential threat to validity for mutation-based assessments of TCP approaches, impacting the generalizability of TCP performance comparisons to real program faults. This suggests that future work should proceed in two directions. First, techniques for carefully selecting mutants should be pursued (see Lesson 3). Second, there is a clear need for comprehensive datasets of real faults, such as Defects4J, in order to properly evaluate TCP approaches. Therefore, researchers could focus on developing reliable automated or semi-automated techniques to extract and isolate real faults from existing open source software projects, which clearly calls for a community-wide effort.

Lesson 2: *The metrics utilized in mutation-based evaluations of TCP techniques impact the representativeness of performance on real faults.* We found that mutation-based APFD values generally exhibit only a *weak positive correlation* to APFD values calculated in terms of real fault discovery, whereas for APFDc this correlation was medium to

strong. However, such results varied across subjects programs. This is important as it signals that when considering the extremely popular APFD metric, mutation-based performance of a particular TCP technique will *generally* be independent of its performance on real faults. This means, that in most cases, one cannot use mutation-based APFD to predict the *practical* performance of TCP technique on real faults, undermining the utility of carrying out such a performance evaluation in the first place. While one could use the more strongly correlated APFDc metric, researchers may not always want to include execution cost in their performance evaluation, instead focusing solely on fault-detection capabilities. While researchers should carefully consider this threat to validity, there are exceptions to this general result as indicated by certain subjects. This brings up the third and most important lesson learned.

Lesson 3: *The types of mutation operators utilized for TCP performance evaluations must be carefully selected or derived in order for the results to be representative of performance on real faults.* Our results indicate that correlations in TCP performance between mutants and real faults vary both across subject programs and across different types of mutation operators within a specific subject program. This suggests that different characteristics of subject programs most likely play a role in determining the representativeness of certain mutation operators for a particular subject (or domain). This is actually a positive outcome when considering the future applicability of mutation testing to TCP evaluations, as it shows that *under the right circumstances* mutation-based TCP can, in fact, be realistic. However, in order to properly achieve the "correct circumstances" for mutation operators to be applied, there is future research that needs to be done. This specifically illustrates the need for the following interconnected research threads: (i) deriving, either manually or automatically, fault models for specific software systems or domains; (ii) developing tailored mutation operators based on such fault models; and (iii) seeding mutation faults by relying on rigorous statistical methods according to observed distributions of faults. If thoroughly pursued, we believe that research efforts directed toward these goals will provide for future tools capable of generating mutants that are more representative of

real faults, not only in the context of TCP, but also in other areas of software testing.

6.6 Conclusion and Discussion

In this work we conducted the first empirical study investigating the extent to which mutation-based evaluations of TCP approaches are *realistic*. We examined the performance, in terms of both the APFD and APFDc metrics, of eight different TCP approaches applied to a dataset of 357 real world faults from the Defects4J dataset and a set of over 35k mutants. Our results indicate that typical mutation-based evaluations of TCP techniques tend to *overestimate* performance on real faults. Furthermore, for the APFD metric in general, performance on mutation faults is not representative of performance on real faults, though this varied across the studied subject programs. Our results highlight the need for future work in deriving mutation operators that are tailored toward specific subject programs or domains, allowing for mutation-based TCP evaluations to be more indicative of performance on real faults.

6.7 Bibliographical Notes

The empirical study presented in this chapter is currently under review:

- **Qi Luo**, Kevin Moran, Massimiliano Di Penta, and Denys Poshyvanyk. “Assessing Test Case Prioritization on Mutants and Real Faults”, under review.

Chapter 7

Conclusion

This dissertation presents a set of contributions to performance testing and test case prioritization. It proposes three novel approaches to select specific input data for potentially exposing performance problems in two scenarios: single-version scenario and evolution scenario. In single-version scenario, the proposed approaches analyze the corresponding execution traces of the specific input data to locate problematic methods that have unexpected worse performance (e.g., longer execution time). In evolution scenario, the novel approach relies on change impact analysis to understand the impact of code changes between versions on performance regressions for identifying the potential problematic code changes. In addition, this dissertation conducts two empirical studies on a large set of real-world Java programs to deeply understand the performances (e.g., effectiveness and efficiency) of TCP techniques and investigate the correlation between mutation-based analysis and real-fault-based analysis in the TCP domain. These key contributions are summarized as follows:

- **Input Sensitive Performance Testing.** This dissertation proposes an adaptive, feedback-directed rule-based learning system, namely FOREPOST, to analyze execution traces for extracting human readable rules which express the relationship between input data and software performance. These rules guide the selection of a subset of input data for exposing performance bottlenecks. In addition, it proposes

an alternative framework of FOREPOST, called FOREPOST$_{RAND}$, which involves some random input data in addition to the specific input data selected based on rules to trigger performance bottlenecks. The intuition here is that random input data is potentially helpful to enlarge the testing coverage to avoid skewing the results. Both of FOREPOST and FOREPOST$_{RAND}$ are implemented and applied to a medium-size commercial subject program and two open-source programs. Based on our experimental results, FOREPOST is able to automatically find more bottlenecks as compared to random testing. Some of the bottlenecks were confirmed by experienced testers and developers. FOREPOST$_{RAND}$ tends to identify the input data with less computationally intensive execution time, but thanks to its partially non-deterministic selection of input data it is able to detect more performance bottlenecks as compare to FOREPOST. Furthermore, this dissertation proposes a novel technique, called GA-Prof, which utilizes genetic algorithms to search the large input data space for finding the specific ones that expose performance bottlenecks. Our experimental results show that GA-Prof is able to find test cases triggering more computationally intensive executions and detect more performance bottlenecks as compared to FOREPOST.

Besides the performance testing in single-version scenario, this dissertation proposes a novel approach, called PerfImpact, which utilizes genetic algorithms to expose performance regressions in the context of an evolving system. It first identifies the specific input data to maximize the time differences between two software versions for exposing performance degradation. Then it uses change impact analysis to investigate the impact of code changes between two versions on the performance degradation for locating the problematic changes. PerfImapct was implemented on five versions across two open-source subject programs. The experimental results show that PerfImpact is able to accurately identify input data for exposing regressions and effectively recommend the potential problematic code changes leading to

the performance regressions.

- **Empirical Studies for Understanding the Performance of TCP Techniques and the Correlation between Mutation-Based and Real-Fault-Based Analysis.** To the best of our knowledge, this dissertation conducts the first empirical study on a large set of real-world Java programs to compare static and dynamic TCP techniques at different test case granularities, in terms of effectiveness, efficiency, and similarity. In addition, it examines the impact of mutation characteristics (e.g., size and type of mutation faults) and software characteristics (i.e., size of software programs) on TCP evaluation. Finally, it analyzes the effectiveness of TCP techniques in terms of APFD and APFDc in software evolution. The experimental results show that call-graph-based TCP with additional strategy outperforms other TCP techniques at test-class level and coverage-additional technique performs the best at test method level. In general, the efficiency of TCP techniques at test-class level is much better than the efficiency at test method level. These two observations imply that test case granularities impact the performance of TCP techniques and suggest that developers/researchers need to consider both of these two granularities in the future. Program size has little effect when evaluating the relative performance of TCP techniques on a given subject. Thus, when the performance of TCP techniques are compared against each other on either large or small programs, similar results can be expected. Moreover, the top test cases prioritized from different TCP techniques share few common detected faults. However, TCP techniques share similar performances across different types of mutation faults, implying that the default setting regarding mutation characteristics will not impact the conclusions in TCP evaluation. Finally, the software evolution does not have clear impact on TCP evaluation.

  In general, TCP techniques are evaluated in terms of APFD and/or APFDc based on the effectiveness of mutation fault detection. However, the evaluation on mutation

fault detection may not be realistic in practice. It is unclear how the TCP techniques perform on detecting real faults and whether these mutation-fault-based results are representative for real-fault-based results. Thus, this dissertation presents the first empirical study to evaluate TCP techniques in terms of real fault detection based on APFD(c) metrics, and compares the results to the mutation-based results for understanding the representative of mutation faults. The results show that mutation faults are not representative for real faults in TCP domain, especially based on APFD values. The correlation between mutation and real faults are stronger based on APFDc values, encouraging future researchers and developers to utilize APFDc instead of APFD to evaluate TCP techniques. Finally, the representativeness varies across different subject programs but keeps consistent within one subject program. This observation suggests that different characteristics of subject programs most likely play a key role in determining whether certain mutation operators will be representative of real faults for that particular subject (or domain). It is promising to extract fault models from subject programs and seed mutants based on these models to make the mutation faults more representative in TCP domain or even in software testing area. This dissertation extracts several learned lessons (see in Chapter 1, 5, and 6) from these findings and observations, which can guide future researchers to design more realistic experiments and highlight potential ideas for future research.

Bibliography

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

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

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

[4] Dell dvd store, http://linux.dell.com/dvdstore/.

[5] Functional testing, https://en.wikipedia.org/wiki/Functional_testing.

[6] Github https://github.com.

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

[8] Mysql, http://www.mysql.com/.

[9] Nonfunctional testing, https://en.wikipedia.org/wiki/Non-functional_testing.

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

[11] Qi luo dissertation online appendix, https://sites.google.com/email.wm.edu/qi-dissertation/.

[12] Tomcat, http://tomcat.apache.org/.

[13] Wasif Afzal, Richard Torkar, and Robert Feldt. A systematic review of search-based testing for non-functional system properties. *Inform. Softw. Tech.*, 51(6):957–976, 2009.

[14] Marcos Kawazoe Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. In *SOSP '03*, pages 74–89, 2003.

[15] Shaukat Ali, Lionel C Briand, Hadi Hemmati, and Rajwinder Kaur Panesar-Walawege. A systematic review of the application and empirical investigation of search-based test case generation. *TSE*, 36(6):742–762, 2010.

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

[17] Paul Ammann, Marcio Eduardo Delamaro, and Jeff Offutt. Establishing theoretical minimal sets of mutants. In *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation*, ICST '14, pages 21–30, Washington, DC, USA, 2014. IEEE Computer Society.

[18] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.

[19] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 402–411, New York, NY, USA, 2005. ACM.

[20] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *ICSE*, pages 402–411, 2005.

[21] James H. Andrews, Lionel C. Briand, and Yvan Labiche. Is mutation an appropriate tool for testing experiments? In *27th International Conference on Software Engi-*

neering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA, pages 402–411, 2005.

[22] James H. Andrews, Lionel C. Briand, Yvan Labiche, and Akbar Siami Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Trans. Softw. Eng.*, 32(8):608–624, August 2006.

[23] James H. Andrews, Lionel C. Briand, Yvan Labiche, and Akbar Siami Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Trans. Software Eng.*, 32(8):608–624, 2006.

[24] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. Efficient and precise dynamic impact analysis using execute-after sequences. In *ICSE '05*, pages 432–441.

[25] Md. Junaid Arafeen and Hyunsook Do. Test case prioritization using requirements based clustering. In *Proc. ICST*, pages 312–321, 2013.

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

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

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

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

[30] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*.

[31] Shay Artzi, Julian Dolby, Frank Tip, and Marco Pistoia. Directed test generation for effective fault localization. In *ISSTA '10*, pages 49–60, 2010.

[32] Shay Artzi, Julian Dolby, Frank Tip, and Marco Pistoia. Practical fault localization for dynamic web applications. In *ICSE '10*, pages 49–60, 2010.

[33] Caroline Ashley. Application performance management market offers attractive benefits to european service providers. *The Yankee Group*, August 2006.

[34] ASM. http://asm.ow2.org/.

[35] Alberto Avritzer, E de Souza e Silva, Rosa Maria Meri Leão, and Elaine J Weyuker. Automated generation of test cases using a performability model. *Software, IET*, 5(2):113–119, 2011.

[36] Alberto Avritzer and Elaine J. Weyuker. Generating test suites for software load testing. In *ISSTA*, pages 44–57, 1994.

[37] Alberto Avritzer and Elaine J. Weyuker. Deriving workloads for performance testing. volume 26, pages 613–633, New York, NY, USA, June 1996. John Wiley & Sons, Inc.

[38] Arthur Baars, Mark Harman, Youssef Hassoun, Kiran Lakhotia, Phil McMinn, Paolo Tonella, and Tanja Vos. Symbolic search-based testing. In *ASE '11*, pages 53–62, 2011.

[39] Linda Badri, Mourad Badri, and Daniel St-Yves. Supporting predictive change impact analysis: A control call graph based technique. In *APSEC '05*, pages 167–175.

[40] Cornel Barna, Marin Litoiu, and Hamoun Ghanbari. Autonomic load-testing framework. In *Proceedings of the 8th ACM International Conference on Autonomic Computing*, ICAC '11, pages 91–100, New York, NY, USA, 2011. ACM.

[41] Mohamad Bayan and João W. Cangussu. Automatic feedback, control-based, stress and load testing. In *Proceedings of the 2008 ACM Symposium on Applied Computing*, SAC '08, pages 661–666, New York, NY, USA, 2008. ACM.

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

[43] D. L. Bird and C. U. Munoz. Automatic generation of random self-checking test cases. *IBM Syst. J.*, 22:229–245, September 1983.

[44] Ken Birman, Gregory Chockler, and Robbert van Renesse. Toward a cloud computing research agenda. *SIGACT News'09*.

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

[46] Joshua Branchaud, Suzette Person, and Neha Rungta. A change impact analysis to characterize evolving program behaviors. In *ICSM '12*, pages 109–118.

[47] Lionel C. Briand, Yvan Labiche, and Marwa Shousha. Stress testing real-time systems with genetic algorithms. In *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation*, GECCO '05, pages 1021–1028, New York, NY, USA, 2005. ACM.

[48] Lionel C Briand, Yvan Labiche, and Marwa Shousha. Stress testing real-time systems with genetic algorithms. In *GECCO '05*, pages 1021–1028, 2005.

[49] Lubomír Bulej, Tomáš Kalibera, and Petr Tma. Repeated results analysis for middleware regression benchmarking. *Perform. Eval.*, 60(1-4):345–358, 2005.

[50] Jacob Burnim, Sudeep Juvekar, and Koushik Sen. Wise: Automated test generation for worst-case complexity. In *ICSE '09*, pages 463–473, 2009.

[51] Benjamin Busjaeger and Tao Xie. Learning for test prioritization: An industrial case study. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium*

on Foundations of Software Engineering, FSE 2016, pages 975–980, New York, NY, USA, 2016. ACM.

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

[53] Harold W Cain, Barton P Miller, and Brian JN Wylie. A callgraph-based search strategy for automated performance diagnosis. In *Euro-Par '00*, pages 108–122, 2000.

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

[55] Cagatay Catal and Deepti Mishra. Test case prioritization: a systematic mapping study. *Software Quality Journal*, 21(3):445–478, 2013.

[56] Ned Chapin, Joanne E. Hale, Khaled Md. Kham, Juan F. Ramil, and Wui-Gee Tan. Types of software evolution and software maintenance. *J. of Softw. Maint. and Evo. R. P.*, 13(1):3–30, 2001.

[57] Thierry Titcheu Chekam, Mike Papadakis, Yves Le Traon, and Mark Harman. An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 597–608, Piscataway, NJ, USA, 2017. IEEE Press.

[58] T. Y. Chen, H. Leung, and I. K. Mak. Adaptive random testing. In *Asian Computing Science Conference*, pages 320–329, 2004.

[59] Tim Chen, Leonid I Ananiev, and Alexander V Tikhonov. Keeping kernel performance from regressions. In *Linux Symposium*, volume 1, pages 93–102, 2007.

[60] TSE HSUN Chen. Studying software quality using topic models. 2013.

[61] Xi Chen, Chin Pang Ho, Rasha Osman, Peter G. Harrison, and William J. Knottenbelt. Understanding, modelling, and improving the performance of web applications in multicore virtualised environments. In *ICPE '14*, pages 197–207, 2014.

[62] Yih-Farn Chen, David S. Rosenblum, and Kiem-Phong Vo. Testtube: A system for selective regression testing. In *ICSE '94*, pages 211–220.

[63] Trishul M Chilimbi, Ben Liblit, Krishna Mehra, Aditya V Nori, and Kapil Vaswani. Holmes: Effective statistical debugging via efficient path profiling. In *ICSE '09*, pages 34–44, 2009.

[64] Jacob Cohen. *Statistical power analysis for the behavioral sciences*. Academic press, 2013.

[65] William W. Cohen. Fast effective rule induction. In *Twelfth ICML*, pages 115–123, 1995.

[66] Emilio Coppa, Camil Demetrescu, and Irene Finocchi. Input-sensitive profiling. In *PLDI '12*, pages 89–98, 2012.

[67] Wieger Cornelissen, Ad Klaassen, Aart Matsinger, and Gerhard van Wee. How to make intuitive testing more systematic. *IEEE Softw.*, 12(5):87–89, 1995.

[68] Jacek Czerwonka, Rajiv Das, Nachiappan Nagappan, Alex Tarvo, and Alex Teterev. Crane: Failure prediction, change analysis and test prioritization in practice – experiences from windows. In *Proceedings of the 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, ICST '11, pages 357–366, 2011.

[69] Muriel Daran and Pascale Thévenod-Fosse. Software error analysis: A real case study involving real faults and mutations. In *Proceedings of the 1996 International*

Symposium on Software Testing and Analysis, ISSTA 1996, San Diego, CA, USA, January 8-10, 1996, pages 158–171, 1996.

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

[71] Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. Mining hot calling contexts in small space. In *PLDI '11*, pages 516–527, 2011.

[72] William Dickinson, David Leon, and Andy Podgurski. Finding failures by cluster analysis of execution profiles. In *ICSE*, pages 339–348, 2001.

[73] Nicholas DiGiuseppe and James A Jones. Fault interaction and its repercussions. In *ICSM '11*, pages 3–12.

[74] Nima Dini, Allison Sullivan, Milos Gligoric, and Gregg Rothermel. The effect of test suite type on regression test selection. In *Software Reliability Engineering (ISSRE), 2016 IEEE 27th International Symposium on*, pages 47–58. IEEE, 2016.

[75] Bogdan Dit, Evan Moritz, and Denys Poshyvanyk. A tracelab-based solution for creating, conducting, and sharing feature location experiments. In *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*, pages 203–208, 2012.

[76] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process*, 25(1):53–95, 2013.

[77] Bogdan Dit, Meghan Revelle, and Denys Poshyvanyk. Integrating information retrieval, execution and link analysis algorithms to improve feature location in software. *Empirical Software Engineering*, 18(2):277–309, 2013.

[78] Bogdan Dit, Michael Wagner, Shasha Wen, Weilin Wang, Mario Linares-Vásquez, Denys Poshyvanyk, and Huzefa Kagdi. Impactminer: A tool for change impact analysis. In *ICSE '14*, pages 540–543.

[79] H. Do, S. Mirarab, L. Tahvildari, and G. Rothermel. An empirical study of the effect of time constraints on the cost-benefits of regression testing. In *FSE*, pages 71–82, 2008.

[80] H. Do and G. Rothermel. A controlled experiment assessing test case prioritization techniques via mutation faults. In *ICSM*, pages 411–420, 2005.

[81] H. Do and G. Rothermel. On the use of mutation faults in empirical assessments of test case prioritization techniques. *TSE*, 32(9):733–752, 2006.

[82] H. Do, G. Rothermel, and A. Kinneer. Empirical studies of test case prioritization in a JUnit testing environment. In *ISSRE*, pages 113–124, 2004.

[83] Hyunsook Do and Gregg Rothermel. An empirical study of regression testing techniques incorporating context and lifecycle factors and improved cost-benefit models. In *Proc. FSE*, pages 141–151, 2006.

[84] Guozhu Dong and James Bailey. *Contrast Data Mining: Concepts, Algorithms, and Applications*. 1st edition, 2012.

[85] Dirk Draheim, John Grundy, John Hosking, Christof Lutteroth, and Gerald Weber. Realistic load testing of web applications. In *Conference on Software Maintenance and Reengineering (CSMR'06)*, pages 11–pp. IEEE, 2006.

[86] S. Elbaum, A. Malishevsky, and G. Rothermel. Prioritizing test cases for regression testing. In *ISSTA*, pages 102–112, 2000.

[87] S. Elbaum, A. Malishevsky, and G. Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *ICSE*, pages 329–338, 2001.

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

[89] Sebastian Elbaum, Praveen Kallakuri, Alexey Malishevsky, Gregg Rothermel, and Satya Kanduri. Understanding the effects of changes on the cost-effectiveness of regression testing techniques. *Software testing, verification and reliability*, 13(2):65–83, 2003.

[90] Sebastian Elbaum, Alexey Malishevsky, and Gregg Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, pages 329–338, Washington, DC, USA, 2001. IEEE Computer Society.

[91] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. Prioritizing test cases for regression testing. In *ISSTA '00*, pages 102–112.

[92] Sebastian Elbaum, Alexey G Malishevsky, and Gregg Rothermel. Test case prioritization: A family of empirical studies. *Software Engineering, IEEE Transactions on*, 28(2):159–182, 2002.

[93] Sebastian Elbaum, Gregg Rothermel, Satya Kanduri, and Alexey G Malishevsky. Selecting a cost-effective test case prioritization technique. *Software Quality Journal*, 12(3):185–210, 2004.

[94] Irene Finocchi Emilio Coppa, Camil Demetrescu. Input-sensitive profiling. *TSE*, 40(12):1185–1205, 2014.

[95] Michael G Epitropakis, Shin Yoo, Mark Harman, and Edmund K Burke. Empirical evaluation of pareto efficient multi-objective regression test case prioritisation. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 234–245. ACM, 2015.

[96] Michael G. Epitropakis, Shin Yoo, Mark Harman, and Edmund K. Burke. Empirical evaluation of pareto efficient multi-objective regression test case prioritisation. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 234–245, New York, NY, USA, 2015. ACM.

[97] Laura Faulkner. Beyond the five-user assumption: Benefits of increased sample sizes in usability testing. *Behavior Research Methods, Instruments, & Computers*, 35(3):379–383, 2003.

[98] King Foo, Zhen Ming Jiang, B. Adams, A.E. Hassan, Ying Zou, and P. Flora. Mining performance regression testing repositories for automated performance analysis. In *QSIC '10*, pages 32–41.

[99] King Chun Foo, Zhen Ming Jiang, Bram Adams, Ahmed E Hassan, Ying Zou, and Parminder Flora. Mining performance regression testing repositories for automated performance analysis. In *Quality Software (QSIC), 2010 10th International Conference on*, pages 32–41. IEEE, 2010.

[100] Gordon Fraser, Andrea Arcuri, and Phil McMinn. A memetic algorithm for whole test suite generation. *JSS*, 2014.

[101] Sören Frey, Florian Fittkau, and Wilhelm Hasselbring. Search-based genetic optimization for deployment and reconfiguration of software in the cloud. In *ICSE '13*, pages 512–521, 2013.

[102] Johannes Furnkranz and Gerhard Widmer. Incremental reduced error pruning. In *International Conference on Machine Learning*, pages 70–77, 1994.

[103] Jean-Pierre Garbani. Market overview: The application performance management market. *Forrester Research*, October 2008.

[104] David Gelperin and Bill Hetzel. The growth of software testing. *Communications of the ACM*, 31(6):687–695, 1988.

[105] Malcom Gethers, Bogdan Dit, Huzefa Kagdi, and Denys Poshyvanyk. Integrated impact analysis for managing software changes. In *ICSE '12*, pages 430–440.

[106] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. Practical regression test selection with dynamic file dependencies. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 211–222. ACM, 2015.

[107] Milos Gligoric, Alex Groce, Chaoqiang Zhang, Rohan Sharma, Mohammad Amin Alipour, and Darko Marinov. Comparing non-adequate test suites using coverage criteria. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 302–313, New York, NY, USA, 2013. ACM.

[108] Milos Gligoric, Rupak Majumdar, Rohan Sharma, Lamyaa Eloussi, and Darko Marinov. Regression test selection for distributed software histories. In *CAV '14*, pages 293–309.

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

[110] Google. Auto scaling on the google cloud platform. https://cloud.google.com/resources/articles/auto-scaling-on-the-google-cloud-platform.

[111] Rahul Gopinath, Carlos Jensen, and Alex Groce. Mutations: How close are they to real faults? In *Proceedings of the 2014 IEEE 25th International Symposium on Software Reliability Engineering*, ISSRE '14, pages 189–200, Washington, DC, USA, 2014. IEEE Computer Society.

[112] Scott Grant, James R. Cordy, and David Skillicorn. Automated concept location using independent component analysis. In *WCRE '08*, pages 138–142, 2008.

[113] Scott Grant, James R Cordy, and David Skillicorn. Automated concept location using independent component analysis. In *WCRE'08*, pages 138–142, 2008.

[114] Mark Grechanik, Chen Fu, and Qing Xie. Automatically finding performance problems with feedback-directed learning software testing. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 156–166, Piscataway, NJ, USA, 2012. IEEE Press.

[115] Mark Grechanik, Chen Fu, and Qing Xie. Automatically finding performance problems with feedback-directed learning software testing. In *ICSE '12*, pages 156–166, 2012.

[116] Mark Grechanik, Qi Luo, Denys Poshyvanyk, and Adam Porter. Enhancing rules for cloud resource provisioning via learned software performance models. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, ICPE '16, pages 209–214, New York, NY, USA, 2016. ACM.

[117] The Yankee Group. Enterprise application management survey. *The Yankee Group*, 2005.

[118] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object-oriented languages. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '97, pages 108–124, New York, NY, USA, 1997. ACM.

[119] Jiaping Gui, Stuart Mcilroy, Meiyappan Nagappan, and William GJ Halfond. Truth in advertising: The hidden cost of mobile ads for software developers. 2015.

[120] Dick Hamlet. When only random testing will do. In *Proceedings of the 1st International Workshop on Random Testing*, RT '06, pages 1–9, New York, NY, USA, 2006. ACM.

[121] Richard Hamlet. Random testing. In *Encyclopedia of Software Engineering*, pages 970–978. Wiley, 1994.

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

[123] Shi Han, Yingnong Dang, Song Ge, Dongmei Zhang, and Tao Xie. Performance debugging in the large via mining millions of stack traces. In *ICSE '12*, pages 145–155, 2012.

[124] Dan Hao, Lingming Zhang, Lu Zhang, Gregg Rothermel, and Hong Mei. A unified test case prioritization approach. *TOSEM*, 10:1–31, 2014.

[125] Murali Haran, Alan Karr, Alessandro Orso, Adam Porter, and Ashish Sanil. Applying classification techniques to remotely-collected program execution data. In *ESEC/FSE-13*, pages 146–155, 2005.

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

[127] Mark Harman, Yue Jia, and William B Langdon. Strong higher order mutation-based test data generation. In *FSE '11*, pages 212–222.

[128] Mark Harman, Kiran Lakhotia, Jeremy Singer, David R White, and Shin Yoo. Cloud engineering is search based software engineering too. *JSS*, 86(9):2225–2241, 2013.

[129] Mark Harman, S. Afshin Mansouri, and Yuanyuan Zhang. Search-based software engineering: Trends, techniques and applications. *CSUR*, 45(1):11:1–11:61, December 2012.

[130] Mark Harman and Phil McMinn. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *TSE*, 36(2):226–247, March 2010.

[131] Christoph Heger, Jens Happe, and Roozbeh Farahbod. Automated root cause isolation of performance regressions during software development. In *ICPE '13*, pages 27–38.

[132] C. Henard, M. Papadakis, G. Perrouin, J. Klein, P. Heymans, and Y. Le Traon. Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test configurations for software product lines. *IEEE Transactions on Software Engineering*, 40(7):650–670, July 2014.

[133] Christopher Henard, Mike Papadakis, Mark Harman, Yue Jia, and Yves Le Traon. Comparing white-box and black-box test prioritization. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE 2016, page to appear, New York, NY, USA, 2016. ACM.

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

[135] Peng Huang, Xiao Ma, Dongcai Shen, and Yuanyuan Zhou. Performance regression testing target prioritization via performance risk analysis. In *ICSE 2014*, pages 60–71.

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

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

[138] IEEE. *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries.* January 1991.

[139] Muhammad Zohaib Iqbal, Andrea Arcuri, and Lionel Briand. Empirical investigation of search algorithms for environment model-based testing of real-time embedded software. In *ISSTA '12*, pages 199–209.

[140] Rebecca Isaacs and Paul Barham. Performance analysis in loosely-coupled distributed systems. In *In 7th CaberNet Radicals Workshop*, 2002.

[141] Md Mahfuzul Islam, Alessandro Marchetto, Angelo Susi, and Giuseppe Scanniello. A multi-objective technique to prioritize test cases based on latent semantic indexing. In *2012 16th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 21–30, 2012.

[142] Sadeka Islam, Kevin Lee, Alan Fekete, and Anna Liu. How a consumer can measure elasticity for cloud platforms. In *ICPE '12*.

[143] JDT. http://www.eclipse.org/jdt/.

[144] Bo Jiang and W.K. Chan. Bypassing code coverage approximation limitations via effective input-based randomized test case prioritization. In *Proc. COMPSAC*, pages 190–199, 2013.

[145] Bo Jiang, Zhenyu Zhang, W. K. Chan, and T. H. Tse. Adaptive random test case prioritization. In *ASE*, pages 257–266, 2009.

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

[147] Zhen Ming Jiang, Ahmed E Hassan, Gilbert Hamann, and Parminder Flora. Automated performance analysis of load tests. In *ICSM '09*, pages 125–134.

[148] Zhen Ming Jiang, Ahmed E. Hassan, Gilbert Hamann, and Parminder Flora. Automatic identification of load testing problems. In *ICSM '08*, pages 307–316.

[149] Zhen Ming Jiang, Ahmed E. Hassan, Gilbert Hamann, and Parminder Flora. Automatic identification of load testing problems. In *ICSM '08*, pages 307–316, 2008.

[150] Zhen Ming Jiang, Ahmed E. Hassan, Gilbert Hamann, and Parminder Flora. Automated performance analysis of load tests. In *ICSM*, pages 125–134, 2009.

[151] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. *PIDI '12*, pages 77–88, 2012.

[152] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 77–88, 2012.

[153] JMeter. https://jmeter.apache.org.

[154] Pallavi Joshi, Chang-Seo Park, Koushik Sen, and Mayur Naik. A randomized dynamic program analysis technique for detecting real deadlocks. *SIGPLAN Not.*, 44(6):110–120, 2009.

[155] Milan Jovic, Andrea Adamoli, and Matthias Hauswirth. Catch me if you can: performance bug detection in the wild. *ACM SIGPLAN Notices*, 46(10):155–170, 2011.

[156] JPetStore. http://sourceforge.net/projects/ibatisjpetstore.

[157] René Just, Darioush Jalali, and Michael D. Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 437–440, New York, NY, USA, 2014. ACM.

[158] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 654–665, New York, NY, USA, 2014. ACM.

[159] Rene Just, Gregory M. Kapfhammer, and Franz Schweiggert. Do redundant mutants affect the effectiveness and efficiency of mutation analysis? In *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, ICST '12, pages 720–725, Washington, DC, USA, 2012. IEEE Computer Society.

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

[161] Gary Kaminski, Paul Ammann, and Jeff Offutt. Improving logic-based testing. *J. Syst. Softw.*, 86(8):2002–2012, August 2013.

[162] Cem Kaner. What is a good test case? In *Software Testing Analysis & Review Conference (STAR) East*, 2003.

[163] Gregory M Kapfhammer and Mary Lou Soffa. Using coverage effectiveness to evaluate test suite prioritizations. In *Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies: held in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007*, pages 19–20. ACM, 2007.

[164] Jussi Kasurinen, Ossi Taipale, and Kari Smolander. Test case selection and prioritization: Risk-based or design-based? In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, page 10. ACM, 2010.

[165] R Kavitha and N Sureshkumar. Test case prioritization for regression testing based on severity of fault. *International Journal on Computer Science and Engineering*, 2(5):1462–1466, 2010.

[166] M. Kendall. A new measure of rank correlation. *Biometrika*, 30(1-2):81–89, 1938.

[167] Maurice G Kendall. A new measure of rank correlation. *Biometrika*, 30(1/2):81–93, 1938.

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

[169] Jung-Min Kim, Adam Porter, and Gregg Rothermel. An empirical study of regression test application frequency. In *ICSE '00*, pages 126–135.

[170] Marinos Kintis, Mike Papadakis, and Nicos Malevris. Evaluating mutation testing alternatives: A collateral experiment. In *Software Engineering Conference (APSEC), 2010 17th Asia Pacific*, pages 300–309. IEEE, 2010.

[171] B. Korel, G. Koutsogiannakis, and L. Tahat. Application of system models in regression test suite prioritization. In *ICSM*, pages 247–256, 2008.

[172] B. Korel, L. H. Tahat, and M. Harman. Test prioritization using system models. In *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pages 559–568, Sept 2005.

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

[174] Heiko Koziolek. Operational profiles for software reliability. In *Seminar on Dependability Engineering, Germany*. Citeseer, 2005.

[175] Tilman Küstner, Josef Weidendorfer, and Tobias Weinzierl. Argument controlled profiling. In *Euro-Par '09*, pages 177–184.

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

[177] LDA. https://cran.r-project.org/web/packages/lda/.

[178] Yves Ledru, Alexandre Petrenko, Sergiy Boroday, and Nadine Mandran. Prioritizing test cases with string distances. *Automated Software Engineering*, 19(1):65–95, 2012.

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

[180] Owolabi Legunsen, Farah Hariri, August Shi, Yafeng Lu, Lingming Zhang, and Darko Marinov. An extensive study of static regression test selection in modern software evolution. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 583–594. ACM, 2016.

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

[182] Bixin Li, Xiaobing Sun, and Hareton Leung. Combining concept lattice with call graph for impact analysis. *Adv. Eng. Softw.*, 53:1–13, 2012.

[183] Bixin Li, Xiaobing Sun, Hareton Leung, and Sai Zhang. A survey of code-based change impact analysis techniques. *Softw. Test., Verif. Reliab.*, 23(8):613–646, 2013.

[184] Z. Li, M. Harman, and R. Hierons. Search algorithms for regression test case prioritisation. *IEEE Trans. Software Eng.*, 33(4):225–237, 2007.

[185] Zheng Li, Mark Harman, and Robert M Hierons. Search algorithms for regression test case prioritization. *TSE*, 33(4):225–237, 2007.

[186] Jingjing Liang, Sebastian Elbaum, and Greg Rothermel. Redefining prioritization: Continuous prioritization for continuous integration. In *Proceedings of the 40th International Conference on Software Engineering*, page to appear, 2018.

[187] Mario Linares-Vásquez, Carlos Bernal-Cárdenas, Kevin Moran, and Denys Poshyvanyk. How do developers test android applications? In *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*, pages 613–622. IEEE, 2017.

[188] Mario Linares-Vasquez, Christopher Vendome, Qi Luo, and Denys Poshyvanyk. How developers detect and fix performance bottlenecks in android apps. In *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, ICSME '15, pages 352–361, Washington, DC, USA, 2015. IEEE Computer Society.

[189] Mario Linares-Vasquez, Christopher Vendome, Michele Tufano, and Denys Poshyvanyk. How developers micro-optimize android apps. *Journal of Systems and Software*, 130:1 – 23, 2017.

[190] Xu Liu, Jianfeng Zhan, Kunlin Zhan, Weisong Shi, Lin Yuan, Dan Meng, and Lei Wang. Automatic performance debugging of spmd-style parallel programs. *JPDC*, 71(7):925–937, 2011.

[191] Richard Lowry. *Concepts and applications of inferential statistics*. R. Lowry, 2014.

[192] Yafeng Lu, Yiling Lou, Shiyang Cheng, Lingming Zhang, Dan Hao, Yangfan Zhou, and Lu Zhang. How does regression test prioritization perform in real-world software evolution? In *In Proc. of the ACM/IEEE International Conference on Software Engineering, ICSE'16*, 2016.

[193] Qi Luo, Kevin Moran, and Denys Poshyvanyk. A large-scale empirical comparison of static and dynamic test case prioritization techniques. *Proceeding of the 24th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, 2016.

[194] Qi Luo, Kevin Moran, Lingming Zhang, and Denys Poshyvanyk. How do static and dynamic test case prioritization techniques perform on modern software systems? an extensive study on github projects. *IEEE Transactions on Software Engineering (TSE)*, page to appear, 2018.

[195] Qi Luo, Kevin Moran, Lingming Zhang, and Denys Poshyvanyk. Tse'17 online appendix. http://www.cs.wm.edu/semeru/data/TSE17-TCPSTUDY/.

[196] Qi Luo, Aswathy Nair, Mark Grechanik, and Denys Poshyvanyk. Forepost: Finding performance problems automatically with feedback-directed learning software testing. *Empirical Software Engineering*, pages 1–51, 2016.

[197] Qi Luo, Denys Poshyvanyk, and Mark Grechanik. Mining performance regression inducing code changes in evolving software. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 25–36, New York, NY, USA, 2016. ACM.

[198] Qi Luo, Denys Poshyvanyk, Aswathy Nair, and Mark Grechanik. Forepost: A tool for detecting performance problems with feedback-driven learning software testing. In *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, pages 593–596, New York, NY, USA, 2016. ACM.

[199] Major. http://mutation-testing.org/.

[200] Haroon Malik, Bram Adams, and Ahmed E Hassan. Pinpointing the subsystems responsible for the performance deviations in a load test. In *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*, pages 201–210. IEEE, 2010.

[201] Haroon Malik, Hadi Hemmati, and Ahmed E Hassan. Automatic detection of performance deviations in the load testing of large scale systems. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 1012–1021. IEEE Press, 2013.

[202] Mallet. http://mallet.cs.umass.edu/.

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

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

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

[206] Hong Mei, Dan Hao, Lingming Zhang, Lu Zhang, Ji Zhou, and Gregg Rothermel. A static approach to prioritizing junit test cases. *IEEE Trans. Softw. Eng.*, 38(6):1258–1275, November 2012.

[207] Atif Memon, Adithya Nagarajan, and Qing Xie. Automating regression testing for evolving gui software. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(1):27–64, 2005.

[208] Atif Memon, Adam Porter, Cemal Yilmaz, Adithya Nagarajan, D Schmidt, and Balachandran Natarajan. Skoll: Distributed continuous quality assurance. In *ICSE '04*, pages 459–468, 2004.

[209] Atif M Memon and Qing Xie. Empirical evaluation of the fault-detection effectiveness of smoke regression test cases for gui-based software. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pages 8–17. IEEE, 2004.

[210] Daniel A Menascé. Load testing, benchmarking, and application performance management for the web. In *Int. CMG Conference*, pages 271–282, 2002.

[211] Daniel A Menascé. Load testing of web sites. *IEEE Internet Computing*, 6(4):70–74, 2002.

[212] Haim Mendelson. Economies of scale in computing: Grosch's law revisited. *Commun. ACM*.

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

[214] Breno Miranda, Emilio Cruciani, Roberto Verdecchia, and Antonia Bertolino. Fast approaches to scalable similarity-based test case prioritization. In *Proceedings of the 40th International Conference on Software Engineering*, page to appear, 2018.

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

[216] Melanie Mitchell. *An Introduction to Genetic Algorithms*. 1998.

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

[218] Nagy Mostafa and Chandra Krintz. Tracking performance across software revisions. In *PPPJ '09*, pages 162–171.

[219] Thomas E. Murphy. Managing test data for maximum productivity. Technical report, 2008.

[220] John D. Musa. Operational profiles in software-reliability engineering. volume 10, pages 14–32, Los Alamitos, CA, USA, March 1993. IEEE Computer Society Press.

[221] Glenford J. Myers. *Art of Software Testing*, volume 10. 1979.

[222] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F Sweeney. Evaluating the accuracy of java profilers. In *PLDI '10*, pages 187–197, 2010.

[223] RB Nelson. Kendall tau metric. *Encyclopedia of Mathematics*, 2011.

[224] Cu D. Nguyen, Alessandro Marchetto, and Paolo Tonella. Test case prioritization for audit testing of evolving web services using information retrieval techniques. In *Proc. ICWS*, pages 636–643, 2011.

[225] Thanh H. D. Nguyen, Meiyappan Nagappan, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. An industrial case study of automatically identifying performance regression-causes. In *MSR '14*, pages 232–241.

[226] Thanh H.D. Nguyen, Bram Adams, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. Automated detection of performance regressions using statistical process control techniques. In *ICPE '12*, pages 299–310.

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

[228] Adrian Nistor, Po-Chun Chang, Cosmin Radoi, and Shan Lu. Caramel: Detecting and fixing performance problems that have non-intrusive fixes. ICSE, 2015.

[229] Adrian Nistor, Tian Jiang, and Lin Tan. Discovering, reporting, and fixing performance bugs. In *MSR '13*, pages 237–246.

[230] Adrian Nistor, Tian Jiang, and Lin Tan. Discovering, reporting, and fixing performance bugs. In *Proceedings of the Tenth International Workshop on Mining Software Repositories*, pages 237–246, 2013.

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

[232] PIT Operators. http://pitest.org/quickstart/mutators/.

[233] Annibale Panichella, Bogdan Dit, Rocco Oliveto, Massimiliano Di Penta, Denys Poshyvanyk, and Andrea De Lucia. How to effectively use topic models for software

engineering tasks? an approach based on genetic algorithms. In *ICSE '13*, pages 522–531.

[234] Mike Papadakis, Christopher Henard, Mark Harman, Yue Jia, and Yves Le Traon. Threats to the validity of mutation-based test assessment. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 354–365, New York, NY, USA, 2016. ACM.

[235] Mike Papadakis, Christopher Henard, and Yves Le Traon. Sampling program inputs with mutation analysis: Going beyond combinatorial interaction testing. In *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation*, ICST '14, pages 1–10, Washington, DC, USA, 2014. IEEE Computer Society.

[236] Sangmin Park, B. M. Mainul Hossain, Ishtiaque Hussain, Christoph Csallner, Mark Grechanik, Kunal Taneja, Chen Fu, and Qing Xie. Carfast: Achieving higher statement coverage faster. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 35:1–35:11, New York, NY, USA, 2012. ACM.

[237] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15:1053–1058, 1972.

[238] Simon Parsons. Independent component analysis: A tutorial introduction. *Knowl. Eng. Rev.*, 20(2):198–199, 2005.

[239] PIT. http://pitest.org/.

[240] Denys Poshyvanyk, Malcom Gethers, and Andrian Marcus. Concept location using formal concept analysis and information retrieval. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 21(4):23, 2012.

[241] Denys Poshyvanyk and Andrian Marcus. Combining formal concept analysis with information retrieval for concept location in source code. In *Proceedings of 15th IEEE International Conference on Program Comprehension(ICPC)*, pages 37–48. IEEE, 2007.

[242] Michael Pradel and Thomas R Gross. Leveraging test generation and specification mining for automated bug detection without false positives. In *ICSE '12*, pages 288–298, 2012.

[243] Xiao Qu, Myra B Cohen, and Gregg Rothermel. Configuration-aware regression testing: an empirical study of sampling and prioritization. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 75–86. ACM, 2008.

[244] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia Chesley. Chianti: A tool for change impact analysis of java programs. In *OOPSLA '04*, pages 432–448.

[245] Google Report. http://google-engtools.blogspot.com/2011/06/testing-at-speed-and-scale-of-google.html.

[246] Meghan Revelle, Bogdan Dit, and Denys Poshyvanyk. Using data fusion and web mining to support feature location in software. In *Program Comprehension (ICPC), 2010 IEEE 18th International Conference on*, pages 14–23, 2010.

[247] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Test case prioritization: an empirical study. In *ICSM*, pages 179–188, 1999.

[248] G. Rothermel, R. J. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *TSE*, 27(10):929–948, 2001.

[249] G. Rothermel, R. J. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Trans. Software Eng.*, 27(10):929–948, 2001.

[250] Gregg Rothermel and Mary Jean Harrold. A safe, efficient regression test selection technique. *ACM Trans. Softw. Eng. Methodol.*, 6(2):173–210, 1997.

[251] Ripon K. Saha, Lingming Zhang, Sarfraz Khurshid, and Dewayne E. Perry. An information retrieval approach for regression test prioritization based on program changes. In *Proc. of the ACM/IEEE International Conference on Software Engineering, ICSE'15*, 2015.

[252] Carey Schwaber, Christopher Mines, and Lindsey Hogan. Performance-driven software development: How it shops can more efficiently meet performance requirements. *Forrester Research*, February 2006.

[253] Pranab Kumar Sen. Estimates of the regression coefficient based on kendall's tau. *Journal of the American Statistical Association*, 63(324):1379–1389, 1968.

[254] Sina Shamshiri, René Just, José Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 201–211. IEEE, 2015.

[255] Weiyi Shang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. Automated detection of performance regression using regression models on clustered performance counters. In *ICPE '15*.

[256] Ajeet Shankar, Matthew Arnold, and Rastislav Bodik. Jolt: Lightweight dynamic analysis and removal of object churn. In *OOPSLA '08*, pages 127–142, 2008.

[257] Samuel Sanford Shapiro and Martin B Wilk. An analysis of variance test for normality (complete samples). *Biometrika*, 52(3/4):591–611, 1965.

[258] Du Shen, Qi Luo, Denys Poshyvanyk, and Mark Grechanik. Automating performance bottleneck detection using search-based application profiling. In *ISSTA '15*.

[259] Adam M Smith and Gregory M Kapfhammer. An empirical study of incorporating cost into test suite reduction and prioritization. In *Proceedings of the 2009 ACM symposium on Applied Computing*, pages 461–467. ACM, 2009.

[260] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. In *ISSTA*, pages 97–106, 2002.

[261] Amitabh Srivastava and Jay Thiagarajan. Effectively prioritizing tests in development environment. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '02, pages 97–106, 2002.

[262] BM Subraya and SV Subrahmanya. Object driven performance testing of web applications. In *Quality Software, 2000. Proceedings. First Asia-Pacific Conference on*, pages 17–26. IEEE, 2000.

[263] Mark D Syer, Zhen Ming Jiang, Meiyappan Nagappan, Ahmed E Hassan, Mohamed Nasser, and Parminder Flora. Leveraging performance counters and execution logs to diagnose memory-related performance issues. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 110–119. IEEE, 2013.

[264] Peri L. Tarr, Harold Ossher, William H. Harrison, and Stanley M. Sutton Jr. degrees of separation: Multi-dimensional separation of concerns. In *ICSE*, pages 107–119, 1999.

[265] Tukey HSD test. https://en.wikipedia.org/wiki/Tukey

[266] Stephen W. Thomas, Hadi Hemmati, Ahmed E. Hassan, and Dorothea Blostein. Static test case prioritization using topic models. *EMSE*, 19(1):182–212, 2014.

[267] Paolo Tonella, Paolo Avesani, and Angelo Susi. Using the case-based ranking methodology for test case prioritization. In *IEEE International Conference on Software Maintenance, ICSM 2009*, pages 123–133, 2006.

[268] Vipindeep Vangala, Jacek Czerwonka, and Phani Talluri. Test case comparison and clustering using program profiles and static execution. In *FSE '09*, pages 293–294, 2009.

[269] Sujata Varun Kumar and Mohit Kumar. Test case prioritization using fault severity. *IJCST*, 1(1), 2010.

[270] WALA. https://github.com/wala/WALA.

[271] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos. Time aware test suite prioritization. In *ISSTA*, pages 1–11, 2006.

[272] Kristen R. Walcott, Mary Lou Soffa, Gregory M. Kapfhammer, and Robert S. Roos. Timeaware test suite prioritization. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, ISSTA '06, pages 1–12, New York, NY, USA, 2006. ACM.

[273] Shuai Wang, David Buchmann, Shaukat Ali, Arnaud Gotlieb, Dipesh Pradhan, and Marius Liaaen. Multi-objective test prioritization in software product line testing: an industrial case study. In *Proceedings of the 18th International Software Product Line Conference-Volume 1*, pages 32–41. ACM, 2014.

[274] Shuai Wang, David Buchmann, Shaukat Ali, Arnaud Gotlieb, Dipesh Pradhan, and Marius Liaaen. Multi-objective test prioritization in software product line testing: An industrial case study. In *Proceedings of the 18th International Software Product Line Conference - Volume 1*, SPLC '14, pages 32–41, New York, NY, USA, 2014. ACM.

[275] Song Wang, Jaechang Nam, and Lin Tan. QTEP: quality-aware test case prioritization. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, pages 523–534, 2017.

[276] Joachim Wegener, Klaus Grimm, Matthias Grochtmann, Harmen Sthamer, and Bryan Jones. Systematic testing of real-time systems. In *EuroSTAR '96*, 1996.

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

[278] Joachim Wegener, Harmen Sthamer, Bryan F Jones, and David E Eyres. Testing real-time systems using genetic algorithms. *Software Quality Journal*, 6(2):127–135, 1997.

[279] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *ICSE '09*, pages 364–374, 2009.

[280] Malcolm R. Westcott. *Toward a contemporary psychology of intuition. A historical and empirical inquiry.* Holt, Rinehart and Winston, 1968.

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

[282] Frank Wilcoxon. Individual comparisons by ranking methods. *Biometrics bulletin*, 1(6):80–83, 1945.

[283] Jonathan Wildstrom, Peter Stone, Emmett Witchel, and Mike Dahlin. Machine learning for on-line hardware reconfiguration. In *IJCAI'07*, pages 1113–1118, 2007.

[284] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques.* Morgan Kaufmann, 2005.

[285] Xusheng Xiao, Shi Han, Dongmei Zhang, and Tao Xie. Context-sensitive delta inference for identifying workload-dependent performance bottlenecks. In *ISSTA '13*, pages 90–100, 2013.

[286] Dianxiang Xu and Junhua Ding. Prioritizing state-based aspect tests. In *Proc. ICST*, pages 265–274, 2010.

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

[288] Zhihong Xu, Myra B. Cohen, and Gregg Rothermel. Factors affecting the use of genetic algorithms in test suite augmentation. In *GECCO '10*, pages 1365–1372.

[289] Zhihong Xu, Myra B. Cohen, and Gregg Rothermel. Factors affecting the use of genetic algorithms in test suite augmentation. In *GECCO '10*, pages 1365–1372, 2010.

[290] Zhihong Xu, Yunho Kim, Moonzoo Kim, and Gregg Rothermel. A hybrid directed test suite augmentation technique. In *ISSRE '11*, pages 150–159.

[291] Dacong Yan, Guoqing Xu, and Atanas Rountev. Uncovering performance problems in java applications with reference propagation profiling. In *ICSE '12*, pages 134–144, 2012.

[292] Cemal Yilmaz, Arvind S Krishna, Atif Memon, Adam Porter, Douglas C Schmidt, Aniruddha Gokhale, and Balachandran Natarajan. Main effects screening: A distributed continuous quality assurance process for monitoring performance degradation in evolving software systems. In *ICSE '05*, pages 293–302.

[293] Shin Yoo and Mark Harman. Regression testing minimization, selection and prioritization: a survey. *STVR*, 22(2):67–120, 2012.

[294] Dongjiang You, Zhenyu Chen, Baowen Xu, Bin Luo, and Chen Zhang. An empirical study on the effectiveness of time-aware test case prioritization techniques. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, pages 1451–1456. ACM, 2011.

[295] Tingting Yu, Xiao Qu, Mithun Acharya, and Gregg Rothermel. Oracle-based regression test selection. In *ICST '13*, pages 292–301.

[296] Tingting Yu, Witawas Srisa-an, and Gregg Rothermel. Simrt: An automated framework to support regression testing for data races. In *ICSE '14*, pages 48–59.

[297] Noel Yuhanna. Dbms selection: Look beyond basic functions. *Forrester Research*, June 2009.

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

[299] Shahed Zaman, Bram Adams, and Ahmed E Hassan. Security versus performance bugs: a case study on firefox. In *Proceedings of the 8th working conference on mining software repositories*, pages 93–102. ACM, 2011.

[300] Shahed Zaman, Bram Adams, and Ahmed E Hassan. A qualitative study on performance bugs. In *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*, pages 199–208, 2012.

[301] Shahed Zaman, Bram Adams, and Ahmed E Hassan. A qualitative study on performance bugs. In *MSR '12*, pages 199–208, 2012.

[302] Dmitrijs Zaparanuks and Matthias Hauswirth. Algorithmic profiling. In *PLDI '12*, pages 67–76, 2012.

[303] Dmitrijs Zaparanuks and Matthias Hauswirth. Algorithmic profiling. *ACM SIGPLAN Notices*, 47(6):67–76, 2012.

[304] Dongsong Zhang and Boonlit Adipat. Challenges, methodologies, and issues in the usability testing of mobile applications. *International Journal of Human-Computer Interaction*, 18(3):293–308, 2005.

[305] Lingming Zhang, Milos Gligoric, Darko Marinov, and Sarfraz Khurshid. Operator-based and random mutant selection: Better together. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 92–102. IEEE, 2013.

[306] Lingming Zhang, Dan Hao, Lu Zhang, Gregg Rothermel, and Hong Mei. Bridging the gap between the total and additional test-case prioritization strategies. In *ICSE'13*, pages 192–201.

[307] Lingming Zhang, Miryung Kim, and Sarfraz Khurshid. Localizing failure-inducing program edits based on spectrum information. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pages 23–32. IEEE, 2011.

[308] Lingming Zhang, Miryung Kim, and Sarfraz Khurshid. Faulttracer: a change impact and regression fault analysis tool for evolving java programs. In *FSE*, page 40, 2012.

[309] Lingming Zhang, Miryung Kim, and Sarfraz Khurshid. Faulttracer: a spectrum-based approach to localizing failure-inducing program edits. *JSEP*, 25(12):1357–1383, 2013.

[310] Lingming Zhang, Darko Marinov, and Sarfraz Khurshid. Faster mutation testing inspired by test prioritization and reduction. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 235–245, New York, NY, USA, 2013. ACM.

[311] Lingming Zhang, Ji Zhou, Dan Hao, Lu Zhang, and Hong Mei. Prioritizing JUnit test cases in absence of coverage information. In *ICSM*, pages 19–28, 2009.

[312] Pingyu Zhang, Sebastian Elbaum, and Matthew B Dwyer. Automatic generation of load tests. In *ASE '11*, pages 43–52, 2011.

[313] Pingyu Zhang, Sebastian G. Elbaum, and Matthew B. Dwyer. Automatic generation of load tests. In *ASE*, pages 43–52, 2011.

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