2016

Input-Sensitive Performance Testing

Qi Luo

Coll William & Mary, Dept Comp Sci, Williamsburg, VA 23185 USA

Recommended Citation

Luo, Qi, Input-Sensitive Performance Testing (2016). *Fse'16: Proceedings of the 2016 24th ACM Sigsoft International Symposium on Foundations of Software Engineering*.
10.1145/2950290.2983953

Input-Sensitive Performance Testing

Qi Luo
Department of Computer Science
College of William and Mary
Williamsburg, VA, USA
qluo@cs.wm.edu

ABSTRACT

One goal of performance testing is to find specific test input data for exposing performance bottlenecks and identify the methods responsible for these performance bottlenecks. A big and important challenges of performance testing is how to deeply understand the performance behaviors of a nontrivial software system in terms of test input data to properly select the specific test input values for finding the problematic methods. Thus, we propose this research program to automatically analyze performance behaviors in software and link these behaviors with test input data for selecting the specific ones that can expose performance bottlenecks. In addition, this research further examines the corresponding execution traces of selected inputs for targeting the problematic methods.

CCS Concepts

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

Keywords

Performance testing, machine learning algorithms, genetic algorithms, change impact analysis

1. INTRODUCTION

During software development and maintenance, a software system may exhibit worsening performance behaviors, such as longer elapsed execution time and/or lower throughput, for specific combinations of test input data [7, 23]. Performance testing is an important activity to target the specific combinations of test input data for detecting the performance bottlenecks. In a survey of 148 enterprises, 92% responses claim that improving software performance has a top priority [22]. During performance testing, software engineers commonly perform two actions: 1) run instrumented software systems with some test input data to expose worsening performance behaviors, and 2) analyze the execution traces to determine the methods responsible for performance degradation. Unfortunately, a nontrivial software system always has a large body of test input data as well as complex logics. It is challenging for engineers to effectively link test input data with software performance behaviors for finding the specific ones that trigger performance bottlenecks, and further locate the problematic methods.

A large body of research work has been dedicated to studying and improving performance testing [6, 7, 11, 18, 26]. Burnim *et al.* provided a testing algorithm to discover worst-case input sizes for exposing bottlenecks [1] whereas Coppa *et al.* proposed an approach to understand performance costs in terms of input size [6]. However, these papers only focus on the impact of input sizes on performance behaviors, failing to consider the potential impact of test input values. Some recent research work utilizes control charts and statistical analysis to target the specific test inputs triggering bottlenecks [16, 17]. However, they do not analyze the corresponding execution traces of the selected test inputs to locate the methods responsible for bottlenecks.

These issues make it clear that existing approaches do not well support engineers in finding specific test inputs to expose performance bottlenecks and locating the problematic methods. Thus, we propose this research program to 1) understand the performance behaviors in terms of selected input combinations for exposing performance bottlenecks, and 2) deeply analyze the traces of the selected inputs to locate the problematic methods. Specifically, we consider two different real-world scenarios of performance testing: 1) single-version scenario, in which targeting the performance bottlenecks in one released software version, and 2) two-version scenario (for an evolving system), in which targeting the problematic code changes responsible for performance regressions between two versions. In the rest of this paper, we introduce the proposed work and the experimental results, and outline the expected contributions of our research.

2. RESEARCH APPROACHES

In this section, we explain our approaches and briefly discuss the experimental results. The complete results can be found in our papers [8, 11, 12, 13, 14, 15, 23].

2.1 FOREPOST

We proposed FOREPOST and its alternative version, FOREPOST$_{RAND}$, which use Machine Learning algorithms (ML) to automatically extract rules for describing the relationship between inputs and performance behaviors, and use these rules to select specific inputs for finding bottlenecks in single-version scenario [13, 14, 8]. Specially, FOREPOST selects inputs based on rules, while FOREPOST$_{RAND}$ selects both of random inputs and the inputs chosen by rules.

Methodology. FOREPOST and FOREPOST$_{RAND}$ are built on two key components: 1) extracting descriptive rules

FSE'16, November 13–18, 2016, Seattle, WA, USA
ACM. 978-1-4503-4218-6/16/11...$15.00
http://dx.doi.org/10.1145/2950290.2983953

for selecting inputs and 2) identifying bottlenecks with the selected inputs. Initially, the system is instrumented and run with random inputs. Collected execution trace for each input is clustered into two groups, *good* and *bad*, based on its execution time. The *good* traces require longer time to execute, which are "good" to expose bottlenecks, while, the *bad* traces need shorter time to execute, which are "bad" to expose bottlenecks. Each trace is represented in a vector-based format, linking test inputs with different performance behaviors (i.e., *good* and *bad*). A ML algorithm, RIPPER [2], is applied on these vectors to extract rules mapping inputs to their performance. FOREPOST selects inputs based on rules while FOREPOST$_{RAND}$ involves both random inputs and the inputs chosen by rules. These inputs are sent back to the system and this feedback loop will be continued until no new rule is generated. At this point, we perform independent component analysis on the traces of selected inputs to analyze methods' contributions to performance. The ones having significant contributions to *good* traces but little contributions to *bad* traces are considered as bottlenecks.

Results. The experiments were done on one commercial system, Renter, and three web-based applications, JPetStore, Dell DVD Store and Agilefant. The results show that FOREPOST is more effective in finding inputs increasing execution time by 78.2% to 333.3% and targeting bottlenecks as compared to random inputs. Some performance bottlenecks of Renter have been confirmed by the software engineers. FOREPOST$_{RAND}$ is more effective in detecting bottlenecks as compared to FOREPOST. FOREPOST has been deployed in cloud to help engineers improve provisioning strategies that guide cloud to (de)allocate resources [8].

2.2 GA-Prof

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. Inspired by effectiveness of GAs in selecting the optimal solutions as a whole in testing domain [9, 24, 25], we proposed an approach, GA-Prof, which uses GAs to search input space for finding the inputs that trigger more performance bottlenecks [23].

Methodology. The key idea is to map determining what combinations of input data expose performance bottlenecks to a search and optimization problem. Initially, we run the instrumented systems with random inputs, and utilize a fitness function to evaluate the collected execution traces. The fitness function maps inputs to their corresponding execution times. The ones requiring longer execution times are selected to create the input data for the next generation via GA operators. This GA process will be continued until pre-defined termination criteria are satisfied. At this point, GA-Prof is able to find the specific input values with longer execution time, likely to expose performance bottlenecks.

Results. We conducted experiments on three open-source applications to compare GA-Prof with FOREPOST. The results show that GA-Prof is able to find the inputs with longer execution times and capture more bottlenecks as compared to FOREPOST across all subjects. For example, GA-Prof captures 5.6 bottlenecks in JPetStore, while FOREPOST only captures approximately two bottlenecks.

2.3 PerfImpact

While FOREPOST and GA-Prof are powerful in detecting performance bottlenecks in a single-version scenario, they are not applicable for the two-version scenario, where the goals of performance testing are finding performance regressions between two versions (software performance degrades in the newly released version as compared to the old version with the same input data), and targeting the code changes responsible for the performance regressions. Thus, we proposed an approach, namely PerfImpact, using GAs to search the input data triggering performance regressions, and utilizing path-based dynamic Change Impact Analysis (CIA) [10] to target the problematic code changes [15].

Methodology. Initially, we run two software versions with the same random inputs, and collect the execution times in two versions for each input. A pre-defined fitness function is used to evaluate inputs for selection. That is, the inputs with longer execution times in new version but shorter execution times in old version are selected as the ones likely to expose regressions. These selected inputs are used to create new ones for the next generation via GA operators. After GA process is terminated, PerfImpact analyzes the corresponding execution traces of selected inputs to extract execution times in two versions for each method. The methods having increased execution time in new version are marked as "problematic" ones. Then, PerfImpact uses CIA to build a impact set for each code change, which contains the methods potentially impacted by this code change. The changes having more problematic methods in their impact sets are considered as the ones likely responsible for regressions.

Results. The experiments were conducted on three versions of Agilefant and two versions of JPetStore. The results show that PerfImpact is significantly more effective in finding inputs exposing performance regressions (162.4% - 288.7% time increase) and locating problematic code changes as compared to random inputs. We further checked the performance behaviors impacted by these identified problematic code changes, and found that the performance of their impact sets exhibits performance degradation in a new version.

3. CONTRIBUTIONS

The goal of our research is to support software engineers to select specific test input data for exposing performance problems and target performance bottlenecks in two testing scenarios. The proposed work on FOREPOST (or FOREPOST-$_{RAND}$) and GA-Prof has already contributed to finding inputs for detecting performance bottlenecks in the single-version scenario. In addition, for the two-version scenario, we proposed PerfImpact to select inputs for exposing performance regressions between two versions, and identifying the problematic code changes likely responsible for the exposed performance regressions. 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 [3, 4, 5, 19, 20, 21]) 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. ACKNOWLEDGMENTS

I would like to thank my advisor Denys Pohsyvanyk, and my collaborators Mark Grechanik, Kevin Moran and Aswathy Nair for their support. This work is supported in part by the NSF CNS-1510239 and CCF-1253837 grants. Any opinions, findings, and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

5. REFERENCES

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

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

[3] B. Dit, E. Moritz, and D. 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.

[4] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk. Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process*, 25(1):53–95, 2013.

[5] B. Dit, M. Revelle, and D. Poshyvanyk. Integrating information retrieval, execution and link analysis algorithms to improve feature location in software. *Empirical Software Engineering*, 18(2):277–309, 2013.

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

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

[8] M. Grechanik, Q. Luo, D. Poshyvanyk, and A. 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, 2016.

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

[10] J. Law and G. Rothermel. Whole program path-based dynamic impact analysis. In *ICSE '03*, pages 308–318.

[11] M. Linares-Vásquez, C. Vendome, Q. Luo, and D. Poshyvanyk. How developers detect and fix performance bottlenecks in android apps. In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, pages 352–361. IEEE, 2015.

[12] Q. Luo, K. Moran, and D. Poshyvanyk. A large-scale empirical comparison of static and dynamic test case prioritization techniques. In *Proceedings of The 24th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, 2016.

[13] Q. Luo, A. Nair, M. Grechanik, and D. Poshyvanyk. Forepost: A tool for detecting performance problems with feedback-driven learning software testing. *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, pages 593–596, 2016.

[14] Q. Luo, A. Nair, M. Grechanik, and D. Poshyvanyk. Forepost: Finding performance problems automatically with feedback-directed learning software testing. *Empirical Software Engineering(EMSE)*, pages 1–51, 2016.

[15] Q. Luo, D. Poshyvanyk, and M. Grechanik. Mining performance regression inducing code changes in evolving software. In *Proceedings of the 13th International Workshop on Mining Software Repositories (MSR)*, pages 25–36. ACM, 2016.

[16] T. Nguyen, B. Adams, Z. M. Jiang, A. Hassan, M. Nasser, and P. Flora. Automated verification of load tests using control charts. In *APSEC '11*, pages 282–289.

[17] T. H. Nguyen, B. Adams, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora. Automated detection of performance regressions using statistical process control techniques. In *ICPE '12*, pages 299–310.

[18] S. Park, B. M. M. Hossain, I. Hussain, C. Csallner, M. Grechanik, K. Taneja, C. Fu, and Q. Xie. Carfast: Achieving higher statement coverage faster. In *FSE '12*, pages 35:1–35:11.

[19] D. Poshyvanyk, M. Gethers, and A. Marcus. Concept location using formal concept analysis and information retrieval. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 21(4):23, 2012.

[20] D. Poshyvanyk and A. 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.

[21] M. Revelle, B. Dit, and D. 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.

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

[23] D. Shen, Q. Luo, D. Poshyvanyk, and M. Grechanik. Automating performance bottleneck detection using search-based application profiling. In *ISSTA'15*, pages 270–281.

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

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

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