Characterizing Performance Regression Introducing Code Changes

Deema Adeeb ALShoaibi
Software Engineering Department, Rochester Institute of Technology, NY, USA
da3352@rit.edu
Advisor: Mohamed Wiem Mkaouer
Software Engineering Department, Rochester Institute of Technology, NY, USA
mwmvse@rit.edu

Abstract—**Performance regression testing is highly expensive as it delays system development when optimally conducted after each code change. As a result, performance regression testing should be devoted to code changes highly probably encountering regression. In this context, recent studies focus on the early identification of potentially problematic code changes through characterizing them using static and dynamic metrics. The aim of my research thesis is to support performance regression by better identifying and characterizing performance regression introducing code changes. Our first contribution has tackled the detection of these changes as an optimization problem. Our proposed approach used a combination of static and dynamic metrics and built using evolutionary computation, a detection rule, which was shown to outperform recent state-of-the-art studies. To extend our research, we are planning to increase metrics used, to better profile problematic code changes. We also plan on reducing the identification cost by searching for a traedeoff that reduces the use of dynamic metrics, while maintaining the detection performance. In addition, we would like to prioritize test case based on code changes characteristics to be conducted when regression predicted.**

I. Introduction

Regression testing is a technique to ensure code modification or addition is not impacting software quality over time. Performance regression testing, as well, monitors quality but specifically its execution time to avoid any degradation during evolution. Due to the growth in the number of committed changes on a regular basis, it is hard for developers to manually identify Performance Regression Introducing Code changes (PRICEs). Also, due to the dynamic nature of the problem, code reviews also cannot easily detect them without proper testing process. So, the ideal solution to detect it is by running performance tests, also known as benchmarks, after each code change. Nevertheless, executing performance testing after each commit is an expensive and a long process that has overhead of resources and delays programmers from further development until the results of testing have been gathered [2]. Therefore, in a real-world setting, software testers are constantly challenged to find the right trade-off between optimally performance testing newly introduced changes, and increasing the development overall productivity [1]. To overcome this problem, testers conduct performance regression testing periodically. This heuristic approach may be successful in detecting any performance issues but it remains expensive as developers should untangle all code changes conducted during that period and test them separately to distinguish the problematic change. For example, if performance tests are postponed by the end of a sprint, then developers need to commit their code throughout the cycle and hope that no performance test would fail by the end, otherwise, they have to rewind all previously committed changes to debug them. In this context, various research has been analyzing performance regression inducing code changes to characterize them and allow their early detection to support the prioritization of performance regression i.e., for upcoming changes to commit, if any of them exhibits characteristics that are similar to these known to have induced performance regression, then this may be a trigger for software testers to schedule their performance tests.

In this proposal, we are outlining our goal in supporting performance testing through these two research thrusts:

PRICE detection. Our plan is to better characterize code changes introducing performance regression to identify them as start then move to prioritizing test cases to detect regression based on code change characteristics. As part of achieving this first research thrust, we show, in this paper, results of our first contribution, in which we tackled the detection of these changes as an optimization problem. Our multi-objective optimizer used a combination of static and dynamic metrics to build a detection rule. Experiments show that our generated rules outperform recent state-of-the-art studies. Our first contribution was recently accepted in the 11th Symposium on Search-Based Software Engineering [16].

Performance test case prioritization. Test Case Prioritization is a critical regression testing practice for the early prevention of anomalies by reordering test cases according to a given strategy. In particular, we plan on avoiding performance regression by combining our detection model as past of the prioritization strategy i.e., we prioritize the testing of PRICEs. We plan on co-evolving both strategies i.e., PRICE detection and test case prioritization, as both algorithms can complement each other: the detection technique flags PRICEs as potential files to test, and so their corresponding test cases are prioritized. Also, the testing results can be used as input to improve the detection, i.e., all false positives and true-negatives can be used to tune the detection model.

II. Related Work

Previous studies focuses on detecting performance regression introducing code changes using code characteristics induced from static and dynamic analysis. Yet, no study investigates how code change characteristics could be used to prioritize test cases when regression detected. Huang et al. [2] devoted regression testing to commits encounter performance regression only. To achieve that they rank commits based on the probability of encountering performance regression based on a static performance risk analysis (PRA). This analysis focuses on how the change is expensive and frequent. After ranking commits, based on the analysis, a comprehensive testing is conducted on risky commits while light testing conducted on the rest. PRA uses code characteristics in finding problematic code changes to apply comprehensive testing. Perphecy [14] consist with PRA [2] that applying comprehensive performing testing on each commit is expensive. Perphecy checks the probability each commit change may introduce performance regression based on a rule generated from history commits data. History commits data are a set of dynamic and static metrics calculated from static and dynamic analysis. Rule generation is deterministic which require trying all possible metric combination. New commit metrics are compared with generated rule to classify it to be introducing performance regression or not. Both Perphecy and PRA are lightweight and programming language dependent approaches since they do not require extensive white-box program analyses. Rather than finding test suite detect performance regression or commits encounter regression, PerfImpact recommendation system [15] recommends subset of inputs and code changes that might introduce performance regression by combining search-based input profiling and change impact analysis.

III. Research Plan

A. Approach Overview

The goal of our approach is to find the best rule that detects PRICE. The general structure is sketched in Figure 1.
Our approach is composed of three phases. Collection phase uses history performance tests data collected from previous commits to calculate metrics. Metrics represent collected data of each commit to the respect of the previous commit. Table 1 lists static and dynamic metrics used in this work. Metrics 1,2,6 are static where the rest are both static and dynamic. The tool used to collect static metrics is Lizard code complexity analyzer. Static data is afterward fed into dynamic analysis process to run benchmarks and calculate dynamic metrics.

The second phase after collecting metrics is generating a detection rule. Finding this rule is a multi-objective optimization problem. A detection rule should have the highest detection of problematic commits while minimizing the detection of benign commits. The search space contains solutions with different combination of metrics and a value for each metric.

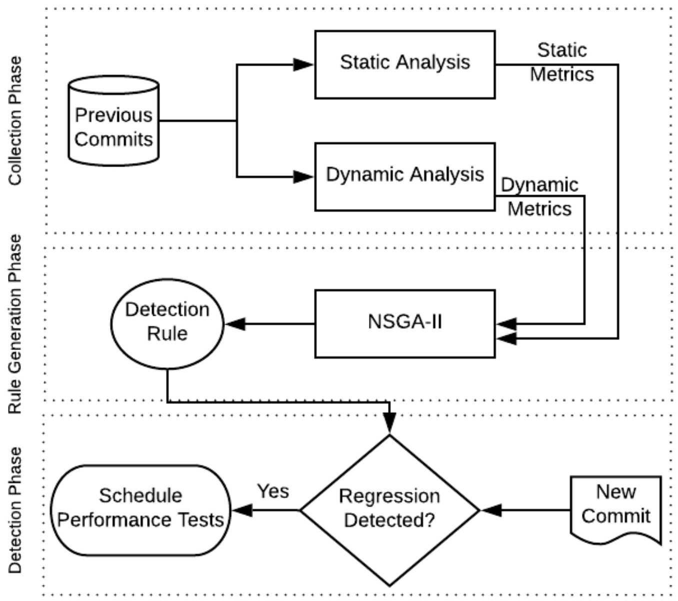

Fig. 1. Approach Overview.

In this paper, we considered seven metrics from a previous study [14], with which we will also compare our approach. Once a detection rule is generated, developers can apply it on each commit to detect regression and decide whether to run benchmark testing or not. In case benchmark testing is applied on a commit, dynamic metrics of that commit is stored on the database to help in updating detection rule in the future when rule is no longer providing good predictions.

IV. Preliminary Results

To evaluate the relevance of generated rules in detecting commits introducing regression, we compared NSGA-II regression detection with other techniques. The comparison held by applying 10-fold cross validation testing. Commits divided into ten equal partitions where one fold is for testing and the rest for training. Folds does not necessarily have same number of problematic commits. Each test is repeated five times to verify results. NSGA-II detection rule is generated by training from data of nine folds then tested on the tenth. Results are compared with two machine learning algorithms which are k-Nearest Neighbors algorithm (KNN) and Boosted Decision Tree by considering performance regression as a binary classification problem and metrics represent the feature space . KNN does not use training data to make any generalizations. Instead, it is based on features similarity that measures how close the new sample is to already defined samples. On the other hand, decision trees uses training data to generate generalized tree-like model of decisions from metrics and their values. Boosted decision tree enhance accuracy during learning process by fitting the residual of preceding trees. However this limits exploring the search space.
We also compared NSGA-II with a state-of-the-art approach called Perphecy [14]. Perphecy is an exhaustive deterministic

TABLE I
METRICS DESCRIPTIONS AND RATIONALES.

#	Description	Rationale
1	Number of deleted functions	Deleted functions indicate refactoring, which may lead to performance changes
2	Number of new functions	Added functions indicate new functionality, which may lead to performance changes
3	Number of deleted Functions reached by the benchmark	Deleting a function which was part of the benchmark execution could lead to a performance change
4	The percent overhead of the top most called function that was changed	Altering a function that takes up a large portion of the processing time of a benchmark has a high risk of causing a performance regression because it is such a large portion of the test
5	The percent overhead of the top most called function that was changed by more than 10% of its static instruction length	Similar to metric 4, however this takes into account that the change affects a reasonable portion of the function in question. Bigger changes may mean higher risk.
6	The highest percent static function length change	Large changes to functions are more likely to cause regressions than small ones
7	The highest percent static function length change that is called by the benchmark	The same as for metric 7, but here we guarantee that the functions are actually called by the benchmark in question.

approach that tries all possible rule combinations to find the best detection rule. Metrics threshold values are in advance determined individually by choosing the threshold that better optimize the hit and dismiss rate.

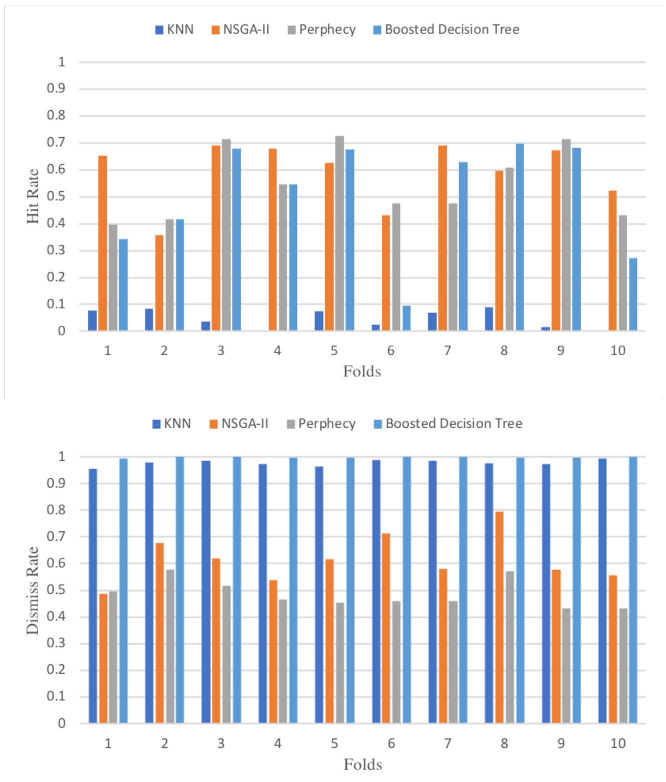

Fig. 2. Hit Rate, and Dismiss Rate of KNN, Boosted decision tree, Perphecy and NSGA-II, on 10-folds.

NSGA-II uses hit and dismiss rates to evaluate population during evolution. In addition we are using these rates to show the difference between the four techniques. Hit rate indicates the number of correctly detected commits to total number of commits encountering regression while dismiss rate is the number of commits classified not to be introducing regression to the total actual number of stable, not problematic, commits.

According to Figure 4 results, KNN's hit rate is very low, and only reached 10% at most, so it highly missclassifies commits with regression in contrast with a more successful dismiss rate where more than 95% of benign commits have been correctly classified. This is due to the imbalance between the two class representations: commits encounter regression are only about 4% of the overall commits. Although, this imbalanced setting represents a challenge for machine learning algorithms, it mimics naturally the real setting for typical software projects, where performance regression tends to be less frequent but critical to software health [2].
Perphecy provided significantly better results than KNN since its hit rate, across folds, varies between 39% and 72%, as for the dismiss rate, it ranges between 42% and 58%. Perphecy is independent of the naive aggregation of all values, and so it clearly outperforms KNN, since its F-Measure goes up to 68% while KNN achieved an F-Measure of 17% at best.

Boosted decision trees shows competing results on detecting problematic code changes with NSGA-II and Perphecy on 7 folds out of 10. However, hit rate difference on the rest 3 folds is ranging between 25% to 40%. This drop on detecting code changes introducing regression is not acceptable. Boosted decision trees provides high detection to non problematic code changes as KNN did.

NSGA-II's performance was competitive to Perphecy, since its hit rate is between 35%, and 69%, which is slightly below perphecy's hit rate, and for the dismiss rate, it ranges between 48% and 79%, which was slightly above perphecy's dismiss rate. As for the F-Measure, as shown in Figure 3, NSGA-II 's values are between 47%, and 68%, and it also outperforms perphecy, in all folds, expect for the second one. The main

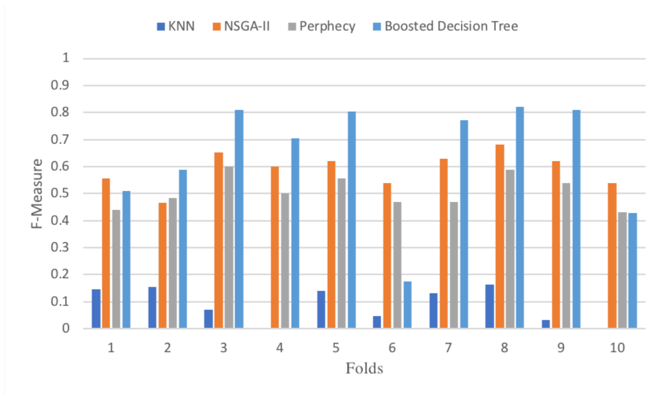

Fig. 3. F-measure of KNN, Boosted decision tree, Perphecy and NSGA-II, on 10-folds.

difference between NSGA-II and Perphecy is the ability of the latter to change the threshold values while composing the decision tree, besides the global exploration of NSGA-II for many possible competing rules during its evolutionary process.

V. Conclusion

In this paper, we tackle performance regression research problem by characterizing code changes inducing it and propose test cases to detect it. Characterizing code changes is done by analyzing and investigating history code changes already known to be problematic and try to find patterns to identify them in the future. Code changes data are a set of metrics measurements before and after the change. Characterizing code changes from collected metrics can be attained by finding what metrics strongly reflects the regression and under which circumstances. Rule characteristics can be controlled during evaluation by managing metrics types and number. Metrics could be static, dynamic or both. Dynamic metrics are costly when considering calculating it for every code change. Additional objective could be considered on generated rule beside maximizing detection is to minimize dynamic metrics included in the rule. Quality metrics as coupling and cohesion may affect software performance. We are planning to include them as detection metrics and find their relation with performance regression if any.

The second step on addressing performance regression problem is by prioritizing test cases. we are planing to use collected code changes characteristics on ranking test cases. This will reduce testing time when regression identified on detection stage.

As we defined performance regression as a Search-Based Software Engineering (SBSE) problem, we will apply different search based optimization algorithms to find the best among them.

References

[1] Ghaith, S., Wang, M., Perry, P., Murphy, J. (2013, March). Profile-based, load-independent anomaly detection and analysis in performance regression testing of software systems. In 2013 17th European Conference on Software Maintenance and Reengineering (pp. 379-383). IEEE.

[2] Huang, P., Ma, X., Shen, D., Zhou, Y. (2014, May). Performance regression testing target prioritization via performance risk analysis. In Proceedings of the 36th International Conference on Software Engineering (pp. 60-71). ACM.

[3] Deb, K., Agrawal, S., Pratap, A., Meyarivan, T. (2000, September). A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II. In International conference on parallel problem solving from nature (pp. 849-858). Springer, Berlin, Heidelberg.

[4] Agrawal, R. B., Deb, K., Agrawal, R. B. (1995). Simulated binary crossover for continuous search space. Complex systems, 9(2), 115-148.

[5] Foo, K. C., Jiang, Z. M., Adams, B., Hassan, A. E., Zou, Y., Flora, P. (2010, July). Mining performance regression testing repositories for automated performance analysis. In 2010 10th International Conference on Quality Software (pp. 32-41). IEEE.

[6] Chen, J., Shang, W. (2017, September). An exploratory study of performance regression introducing code changes. In 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME) (pp. 341-352). IEEE.

[7] Ostermueller, E. (2017). Troubleshooting Java Performance: Detecting Anti-Patterns with Open Source Tools. Apress.

[8] Khatibsyarbini, M., Isa, M. A., Jawawi, D. N., Tumeng, R. (2018). Test case prioritization approaches in regression testing: A systematic literature review. Information and Software Technology, 93, 74-93.

[9] Tallam, S., Gupta, N. (2006). A concept analysis inspired greedy algorithm for test suite minimization. ACM SIGSOFT Software Engineering Notes, 31(1), 35-42.

[10] Hsu, H. Y., Orso, A. (2009, May). MINTS: A general framework and tool for supporting test-suite minimization. In Proceedings of the 31st International Conference on Software Engineering (pp. 419-429). IEEE Computer Society.

[11] Wang, S., Ali, S., Gotlieb, A. (2015). Cost-effective test suite minimization in product lines using search techniques. Journal of Systems and Software, 103, 370-391.

[12] Shi, A., Yung, T., Gyori, A., Marinov, D. (2015, August). Comparing and combining test-suite reduction and regression test selection. In Proceedings of the 2015 10th joint meeting on foundations of software engineering (pp. 237-247). ACM.

[13] De Melo, A. C. (2010, September). The new linuxperftools. In Slides from Linux Kongress (Vol. 18).

[14] De Oliveira, A. B., Fischmeister, S., Diwan, A., Hauswirth, M., Sweeney, P. F. (2017, March). Perphecy: performance regression test selection made simple but effective. In 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST) (pp. 103-113). IEEE.

[15] Luo, Q., Poshyvanyk, D., Grechanik, M. (2016, May). Mining performance regression inducing code changes in evolving software. In 2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR) (pp. 25-36). IEEE.

[16] AlShoaibi, D., Hannigan, K., Gupta H., Mkaouer, M.W. (2019, August). PRICE: Detection of Performance Regression Introducing Code Changes Using Static and Dynamic Metrics. In 2019 11th Symposium on Search-Based Software Engineering (to appear). Springer.