January 2024

Targeted Software Profiling Based on Static Code Analysis to Detect Small Regressions

Shu-Wei Cheng

Kamakshi Kodur

Anne Stern

Yue Hu

Fei Wang

See next page for additional authors

Inventor(s)
Shu-Wei Cheng, Kamakshi Kodur, Anne Stern, Yue Hu, Fei Wang, Zening Li, Oscar Ding, Kevin Culberg, Eugene Krasichkov, Aleph Aseffa, and George Lu

Targeted Software Profiling Based on Static Code Analysis to Detect Small Regressions

ABSTRACT

Degradation of performance between different release versions of a software is termed as regression. Traditional reliability testing and benchmarking tools can detect regressions of large magnitudes much more easily compared to those with smaller regression effects. As code changes accumulate over time, the cumulative impact of undetected micro regressions can add up to noticeable negative impact on performance. This disclosure describes techniques for timely detection of micro regressions based on static analysis of code changes in a change request and by targeted dynamic benchmarking. The static analyses can be performed by comparing the AST and/or call graphs of the software before and after changes connected to a change request. The results of the comparison can be employed to detect any small or large regression resulting from the changes via bytecode injected binaries of the software in a laboratory testing environment. The approach can save substantial time and effort in detecting and addressing small regressions, thus helping speed up the application as well as the software development pipeline and avoiding negative impacts on user engagement.

KEYWORDS

- Regression testing
- Reliability testing
- Performance metrics
- Abstract Syntax Tree (AST)
- Call graph
- Change log
- Code injection
- Static analysis
- Micro regression
- Targeted code profiling

BACKGROUND

Software development involves work that is performed by teams of software professionals. Large development teams are often divided into sub-teams. Software releases are organized with individual software features serving as the base coding unit used for tracking various milestones, such as experiments, rollouts, launches, etc. Different software features are developed independently by assigning the responsibility for implementing individual features to specific developers or sub-teams. The individual or team in charge of developing a feature commits the corresponding code in the form of a change request to a code repository that stores the code for the overall software solution.

Software solutions typically evolve over time with each successive released version containing a number of additional features and/or refinements to existing features. In addition, it is generally expected that performance metrics for a software release be better than, or at least equal to, those of the previous release. The performance metrics can include a number of measures and benchmarks such as user interface latency, memory use, computational resource use, etc.

Software performance can be examined by benchmarking and tracing tools. Benchmarking tools treat the software as a black box and run multiple A/B tests across multiple devices to detect performance changes. Tracing tools measure various parameters and tracing tools generate an end-to-end stack log of software operation containing various pieces of runtime information about invoked methods calls, such as name, start time, duration, call depth, threat number, etc. If the performance for a given metric for a software release degrades compared to the previous release, the situation is termed as “regression.”

Traditional reliability testing and benchmarking tools can detect regressions of large magnitudes much more easily compared to those with smaller regression effects. The challenges in detecting such micro regression effects can arise because of a number of factors, such as noise from the software and the device under test (DUT). As a result, current techniques either fail at detecting micro regressions or generate false positives that cannot be reliably reproduced. As code changes accumulate over time, the cumulative impact of any undetected micro regressions can add up to noticeable negative impact on performance. Moreover, even tiny negative changes in performance metrics such as startup latency can have a measurable negative impact on user engagement and user experience (UX).

Accurately and timely identification of features that cause small regressions is a challenging problem with a history of unsuccessful attempts. The lack of success mostly arises because of the large number of devices required to measure small performance changes in a meaningful way. Software that involves a number of different configurations can pose additional challenges for detecting small regressions because some regressions can be introduced in developer builds or be guarded by a testing experiment and surface only after the corresponding features are promoted to the production version.

DESCRIPTION

This disclosure describes techniques to detect micro regressions by focusing on the parts of the software changed by a specific code change request from a developer. Benchmarking can be applied in a manner that targets the changed parts to determine whether the changes caused regression along with the size and root cause(s) of the regression.

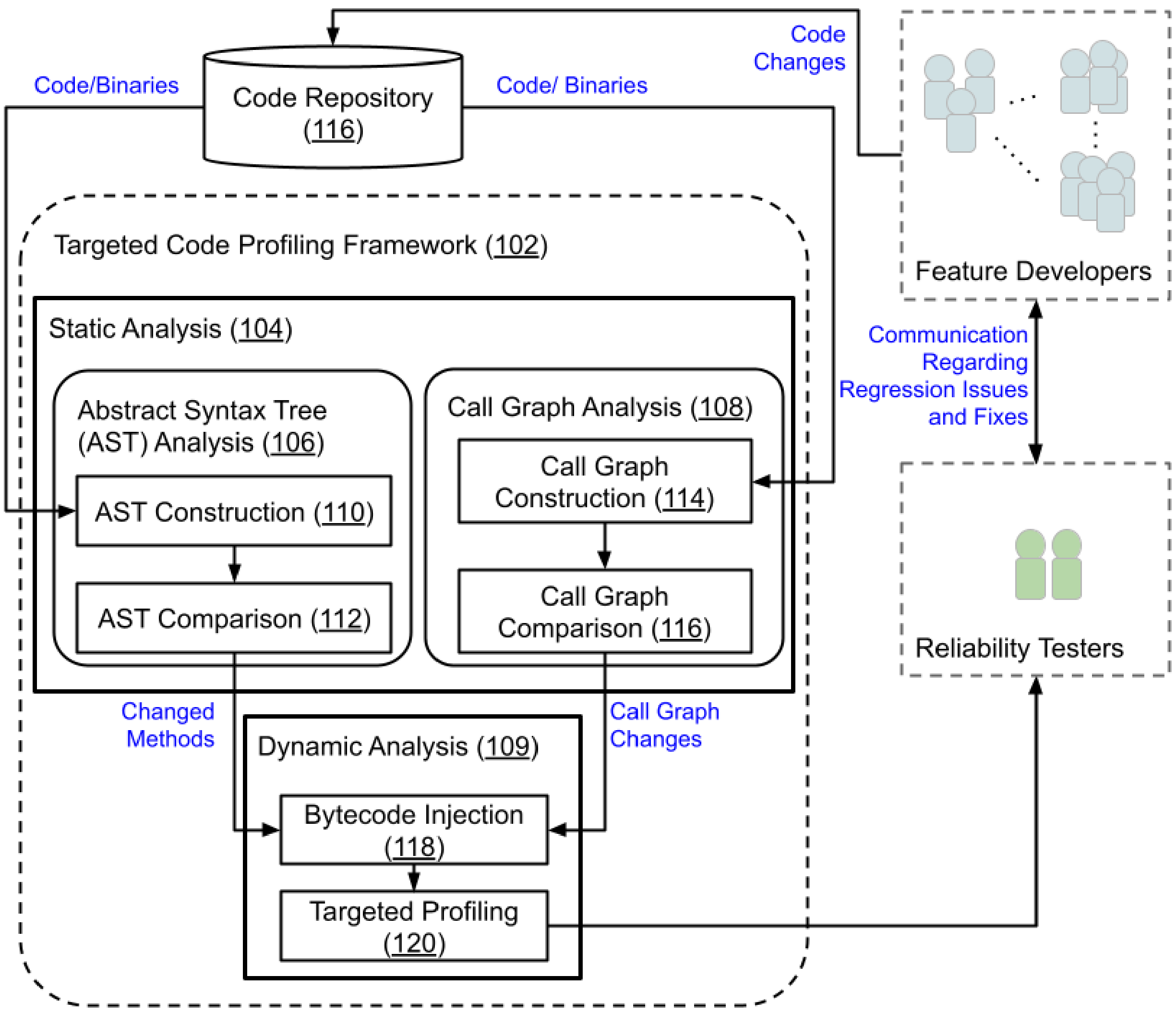

Fig. 1: Targeted code profiling to detect micro regressions

Fig. 1 shows an example operational implementation of the techniques described in this disclosure. As shown in Fig. 1, the implementation of the targeted code profiling framework (102) can be achieved via two approaches involving initial static software analyses (104) followed by targeted dynamic analysis (109) of relevant bytecode:

1. **Abstract Syntax Tree (AST) analysis (106)**: The AST for each file modified in a given code change request can be constructed (110) and compared (112) with that for the previous version. The comparison can be used to generate a list of methods that changed because of

the given change request. The performance impact of the methods in the list can then be profiled in a targeted manner (120) using bytecode injected binaries (118) of the software in a laboratory testing environment.

2. **Call graph comparison (108)**: Inter-procedure static analysis can be employed to construct the call graphs of the entire application startup path for versions with and without the changes introduced in a given change request. The two call graphs can be compared to determine whether the change request results in any changes in the call graph. If any changes are uncovered, the changed parts can be profiled (120) to measure the performance impact of the changes using bytecode injected binaries (118) of the software in a laboratory testing environment.

The two approaches can be employed sequentially or in parallel as appropriate. Because of their targeted nature that focuses only on relevant parts of the code when benchmarking, the techniques described herein avoid unwanted noise from other parts of the application, thus facilitating detection of micro regressions that typically go undetected by traditional regression testing methods.

Reliability testers and feature developers can take appropriate mitigating actions based on any detected regressions. For instance, regressions that increase the latency in startup of the application can be addressed by preventing the execution of unnecessary regression-inducing methods on the main thread, thus ensuring that the application starts and responds to user interaction as quickly as possible.

The techniques described herein can be implemented to perform regression testing in any software to check one or more of any relevant performance metrics, such as latency, memory usage, computational speed, power drain, etc. Implementation of the techniques can enable the detection of small regressions in a timely and focused manner, thus saving time and effort in addressing the underlying issues. As a result, the techniques can help speed up the application as well as the software development pipeline and avoid the negative revenue impacts of undetected small regressions that degrade the UX and decrease user engagement.

CONCLUSION

This disclosure describes techniques for timely detection of micro regressions in software based on static analysis of code changes in a change request and by targeted dynamic benchmarking. The static analyses can be performed by comparing the AST and/or call graphs of the software before and after changes connected to a change request. The results of the comparison can be employed to detect any small or large regression resulting from the changes via bytecode injected binaries of the software in a laboratory testing environment. The approach can save substantial time and effort in detecting and addressing small regressions, thus helping speed up the application as well as the software development pipeline and avoiding negative impacts on user engagement.