Vision of Continuously Assuring Performance

David Georg Reichelt
Universität Leipzig
dg.reichelt@uni-leipzig.de

Stefan Kühne
Universität Leipzig
kuehne@uni-leipzig.de

Wilhelm Hasselbring
Universität Kiel
hasselbring@email.uni-kiel.de

Abstract

When assuring performance, small performance differences at code level are difficult to measure at application level. Current approaches aimed at performance assurance are capable of identifying hotspots and major performance bugs. Apart from their inability to detect small regressions, they require manual effort for specification and execution. In this paper, we present the vision of continuously assuring performance by using functional unit tests. Utilizing small tests allows developers to detect small performance differences. Additionally, they do not have to define workloads manually if they use functional unit tests, since these are present in most projects. To achieve this, we propose integrating performance measurements in the continuous integration (CI) process, accelerating root cause analysis and creating parallel tests capable of identifying regressions that arise with parallel use.

1 Introduction

Preventing performance regressions is a challenge for developers and operators. Performance regressions can arise at architecture, deployment or code level. Various tools such as profiling, benchmarking, load testing and monitoring are specialized to identify performance problems at one or more of these levels.

While these tools provide useful information, they suffer from two main problems, namely (1) They cannot efficiently detect small performance differences, since performance measurements are nondeterministic and measuring performance differences that are small in relation to the usual standard deviation of measurements is difficult and (2) they require manual effort for configuration and execution (profiling and monitoring) or for specification of execution scripts (benchmarking and load testing). Given these problems, small performance regressions are not found and may add up over time. Additionally, only highly relevant parts of programs are covered with benchmarks and load tests due to costs; therefore, issues in other parts of a program are not detected.

Our vision is that performance regressions, which are currently too small to be examined, can be found and understood by measuring transformed functional unit tests. To evaluate this vision, a Java-prototype of this vision will be implemented in the project *PermanEnt* (Performance assurance Efficiently integrated).

The remainder of this paper is structured as follows: first, the approach pursued by *PermanEnt* is described. Next, its components–the CI-integration of the measurement, the accelerated root cause analysis, and the test generation–are sketched. Finally, related work is described and a summary is given.

2 Approach

To identify small regressions, small performance benchmarks that measure a large share of the code base of a software are needed. Developing those benchmarks is not feasible from a cost view. Therefore, we rely on the unit-test-assumption: *"The performance of relevant use cases of a program correlates with the performance of at least a part of its unit tests, if the performance is not driven mainly by external factors."* [16]

Reusing unit tests for performance regression testing suffers from three main problems, namely (1) In contrast to functional correctness, the fulfillment of performance requirements cannot be assured by single executions since performance behavior is nondeterministic [2]. To overcome this, the same workload needs to be repeated, which is time-consuming. (2) While assertions allow developers to trace the reason of failures in functional unit tests, identifying root causes of performance regressions often requires manual experimentation and is therefore not feasible for small and frequent regressions. (3) Since the workloads of unit tests are non-parallel, the parallel performance of the program cannot be examined.

The research prototype PeASS (**Pe**rformance **A**nalysis of **S**oftware **S**ystem Versions) [16] enables performance measurement of unit tests by transforming them to Performance Unit Tests (**PUT**s). Since the measurement process is time-consuming, PeASS relies on regression test selection [12].

While PeASS automates the performance measurement, the daily usage of unit tests for performance measurement is limited by: (1) the lack of an integration of performance measurement in CI processes, (2) the lack of a root cause analysis, which detects performance changes in a time which is reasonable for a reaction of the developer and (3) the lack of a method for identifying performance issues with parallel test execution. *PermanEnt* strives for solving these issues

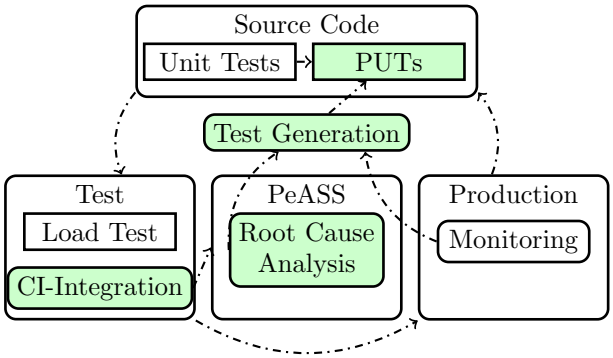

Figure 1: Approach of *PermanEnt*

by: (1) **integration** of performance unit test measurement into **CI**, (2) researching methods to **speed up root cause analysis** and (3) automatic **generation of tests** from existing test data and monitoring data. This approach is summarized in Figure 1. We address the solution of each issue in one of the following sections. Prototypes of this project will be created in Java and released under an open source license.

3 Integration of Performance Measurement in CI

To measure the performance of an applications unit tests in CI, the measurement needs to be technically integrated into the CI infrastructure used. Afterwards, the continuous performance measurement needs to be able to identify performance changes fast and with reasonable precision. Therefore, measurements need to be configured individually. Since performance measurements may disturb each other, they are currently mostly run on stand-alone machines.

Therefore, two research questions arise: *(RQ I) Which configuration (number of VM executions, iterations, etc.) is suitable for identifying performance regressions? (RQ II) How can performance measurements be isolated from other processes on a machine?*

To answer (RQ I), we plan to research the size of unit tests in terms of method executions and durations. We plan to automatically generate a configuration which is suitable for identifying changes based on an unit tests duration and method call count, the minimal duration difference and the expected accuracy. To answer (RQ II), we will research whether isolation measures, e.g. activation of the linux kernel feature cgroups[1] and setting the JVM RAM limit, are able to isolate performance measurements from each other. Thereby, we expect to be able to predict the accuracy with which performance changes can be identified in parallel to each other and other processes.

4 Accelerate Root Cause Analysis

Once a performance regression is detected, the developer needs to know its root cause. Thereby, he can decide whether a performance regression is a problem which needs to be fixed or whether it is a necessary regression caused by, for example, a functional change. Root cause analysis (RCA) [4] identifies performance change causes by systematic measurement of the nodes of a call tree.

Figure 2 visualizes the following. First, the topmost level, here `foo`, is measured. If `foo()` gets slower, the child nodes in the next level, `bar()` and `doo()` are measured. Then, their child nodes are measured if each individual node got a regression until no changes are measured anymore. Then, the node(s) which contain a regression and have no child containing a regression are the root cause, here `fem()`. The measurement of the individual nodes can be done using Kieker application monitoring [17]. This process takes several hours per level. The measurement may add up to several days if the regression causing node is far from the root node, which is not feasible in everyday software development.

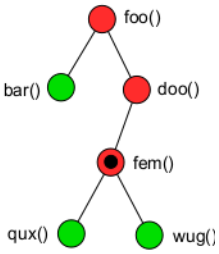

Figure 2: RCA from: [4]

To speed the measurement up, we plan to answer the following research questions: *(RQ III) Can the use of adaptive monitoring, i.e. reusing the same JVM and switching the monitored method, speed up the RCA? (RQ IV) Can the measurement of partial call trees, i.e. measuring more than one level at a time, speed up the RCA?*

Answering these questions using artificial workload is infeasible since different workloads use specific combinations of CPU, RAM and I/O access. Therefore, we will use known performance change causes from our industry partners. First, we will execute the regular RCA to determine the root causes and measurement duration. Second, we will attempt both approaches for speeding up RCA and check their duration and accuracy.

5 Test Generation

Using unit tests for performance measurement does not identify issues arising with parallel usage. Parallel unit test execution can detect some performance issues [9]. Therefore, we will research the question: *(RQ V) Is it possible to identify performance changes by parallel execution of the same unit tests?*

To research this question, we plan to use known performance regressions from industry partners which only occur during parallel usage. Then, we will generate parallel performance unit tests which reuse the same object instances. The generation will use call frequencies and orders from production monitoring data, and root cause analysis data to specifically reproduce behaviour that exposes performance problems. Based on the generated tests, we will research how the parallel execution of the same test methods can show these known performance regressions.

[1] https://en.wikipedia.org/wiki/Cgroups

6 Related Work

Related work (1) identifies performance changes by continuous assertion, (2) identifies performance problems or (3) generates performance tests. In the following, these groups of related work will be described.

Maddodi et al. [8] give an overview about *continuous performance assertion (1)*. This is done using models [10] or measurement of changes [7, 11, 15]. Domain-specific approaches like performance measurement scripts of the linux kernel [1], Raptor[2] for Firefox, and JMH[3] for OpenJDK exist. *Identifying performance problems (2)* happens at architecture or code level. At architecture level, antipatterns are identified for example by combining models and measurement results [5]. Furthermore, analysis of monitoring data detects performance anomalies [3]. At code level, problems can be found by correlation with code patterns [14]. The *generation of performance tests (3)* includes generating propabilistic workloads by Markov4JMeter [6] and generating realistic load instances by analysis of behaviour data [13]. None of the methods identifies small performance regressions at code level. With PermanEnt, we close this gap.

7 Summary

We presented our vision of continuous performance assurance in software development processes by the reuse of unit tests. Our approach integrates the measurements in CI on regular basis by automatically determining the configuration and isolating the process from other processes on the machine, accelerates root cause analysis by reusing the same JVM or measuring more than one level at a time, and generates parallel performance tests. By this approach, we aim for identifying performance changes at the code level. Thereby, regressions which are currently not detectable or too time-intense to fix can be avoided.

Acknowledgements This work is funded by the German Federal Ministry of Education and Research within the project "Performance Überwachung Effizient Integriert" (*PermanEnt*, BMBF 01IS20032D).

References

[1] T. Chen, L. I. Ananiev, and A. V. Tikhonov. "Keeping kernel performance from regressions". In: *Linux Symposium*. Vol. 1. 2007, pp. 93–102.

[2] A. Georges, D. Buytaert, and L. Eeckhout. "Statistically rigorous java performance evaluation". In: *ACM SIGPLAN Notices* 42.10 (2007), pp. 57–76.

[3] N. S. Marwede et al. "Automatic Failure Diagnosis in Distributed Large-Scale Software Systems based on Timing Behavior Anomaly Correlation". In: *ECSMR*. 2009.

[4] C. Heger, J. Happe, and R. Farahbod. "Automated Root Cause Isolation of Performance Regressions During Software Development". In: *ICPE 13*. Prague, Czech Republic: ACM, 2013.

[5] A. Wert, J. Happe, and L. Happe. "Supporting swift reaction: Automatically uncovering performance problems by systematic experiments". In: *ICSE*. 2013.

[6] A. van Hoorn et al. "Automatic extraction of probabilistic workload specifications for load testing session-based application systems". In: *ICPEMT (ValueTools)*. ICST. 2014.

[7] W. Shang et al. "Automated detection of performance regressions using regression models on clustered performance counters". In: *ICPE*. 2015.

[8] G. Maddodi et al. "The daily crash: a reflection on continuous performance testing". In: *ICSEA* (2016).

[9] M. Selakovic and M. Pradel. "Performance issues and optimizations in javascript: an empirical study". In: *ICSE*. ACM. 2016, pp. 61–72.

[10] A. Brunnert and H. Krcmar. "Continuous performance evaluation and capacity planning using resource profiles for enterprise applications". In: *JSS* 123 (2017).

[11] J. Chen and W. Shang. "An Exploratory Study of Performance Regression Introducing Code Changes". In: *IEEE ICSME 2017*. IEEE. 2017.

[12] D. G. Reichelt and S. Kühne. "Better Early Than Never: Performance Test Acceleration by Regression Test Selection". In: *Companion of the 2018 ACM/SPEC ICPE*. Berlin, Germany: ACM, 2018, pp. 127–130.

[13] C. Vögele et al. "WESSBAS: extraction of probabilistic workload specifications for load testing and performance prediction—a model-driven approach for session-based application systems". In: *SoSyM* 17.2 (2018), pp. 443–477.

[14] J. Chen et al. "Analyzing performance-aware code changes in software development process". In: *ICPC*. IEEE Press. 2019, pp. 300–310.

[15] C. Laaber. "Continuous software performance assessment: detecting performance problems of software libraries on every build". In: *ACM SIGSOFT ICSTA*. 2019, pp. 410–414.

[16] D. G. Reichelt, S. Kühne, and W. Hasselbring. "PeASS: A Tool for Identifying Performance Changes at Code Level". In: *ACM/IEEE ASE*. 2019.

[17] W. Hasselbring and A. van Hoorn. "Kieker: A monitoring framework for software engineering research". In: *Software Impacts* 5 (2020).

[2] https://wiki.mozilla.org/TestEngineering/Performance/Raptor
[3] https://openjdk.java.net/projects/code-tools/jmh/