Generic Performance Measurement in CI: The GeoMap Case Study

David Georg Reichelt
d.g.reichelt@lancaster.ac.uk
Lancaster University Leipzig, Leipzig, Germany

Hannes Krauß
hannes.krauss@evermind.de
evermind GmbH, Leipzig, Germany

Stefan Kühne
stefan.kuehne@uni-leipzig.de
Universität Leipzig, Leipzig, Germany

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

Abstract

Continuously developed industry projects with medium budget often focus on functional correctness, but not on optimal performance. This is partially caused by the lack of easily available approaches and tooling that check for performance changes. The tool Peass-CI examines performance changes by measurement and analysis of the duration of unit tests. We present a case study of establishing a continuous performance engineering process in GeoMap, a Spring-based tool that allows to analyze the real estate market for real estate service and market experts.

In the continuous performance engineering process, we monitored performance changes that happened during six months and derived performance improvements by code reviews and load test execution. We found that (1) continuous performance measurement gives detailed insights into performance changes and (2) performance improvements by source code changes are reproducible using performance measurement of unit tests in Peass-CI.

1 Introduction

It is possible to detect software performance changes, including regressions that require fixes, by performance measurement of microbenchmarks or existing unit tests [5, 6, 7]. In addition to those case studies, that build on application-specific tooling, a more generic process for performance measurement of unit tests would allow broader use. In our case study, we research how a continuous performance engineering process–including usage of the tool Peass-CI[1]–affects the performance of the real estate tool GeoMap.

In modern execution environments like the JVM, performance measurement is hard since non-deterministic effects such as just-in-time compilation and garbage collection influence measured performance values. To make substantial claims about performance changes, the repetition of a workload inside a JVM, the repetition of starts of the JVM and a statistically rigorous analysis are necessary. This process is time-consuming. Using the tool Peass-CI, the process can be sped up using a regression test selection for performance measurement of unit tests.

[1]`https://github.com/jenkinsci/peass-ci-plugin`

We conducted a case study on the tool GeoMap, where we installed the tool Peass-CI in a Jenkins server and measured the performance during the development. 21 commits out of all 240 commits cause performance changes. Acting on them is rarely useful, because no commit detoriated the performance in an unacceptable size. Furthermore, we implemented performance improvements based on code reviews and load test executions. Thereby, we improved the performance of the load tests significantly.

The remainder of this paper is structured as follows: First, we introduce the tool GeoMap. Afterwards, we describe the adaptions required to make Peass-CI usable on GeoMap. In the following section, we describe our measurement results on regular performance measurements and our performance improvements. Our measurement results are subsequentially compared to related work. Finally, we give a summary and an outlook.

2 GeoMap

GeoMap is a tool for real estate data analysis.[2] It supports real estate experts with data of offers, statistical and historical data of prices and sociodemographic and socioeconomic data that are the fundation of real estate investment decisions. Since GeoMap analyzes a big amount of data and experts need to work with the tool day-to-day, fast responses are necessary.

The tool consists of three components: The frontend, which is used by real estate experts to make their analysis, the backend, which provides data to the frontend via a REST interface and an API for external usage, and the crawler, which crawls various sources for real estate data. Since the performance visible to the end user is mostly driven by the backend, we focus on it. The backend is built using a microservice architecture, were each microservice provides individual functionalities e. g. for data management of real estate objects. The data is persisted in a NoSQL database.

[2]`https://geomap.immo/`

The backend uses three main libraries, which provide data transfer objects, commonly used functions, e.g., for spatial data mangement, and an abstraction layer for the database access.

GeoMap uses three kinds of test: Regular unit tests, docker-based tests that use locally started docker instances and integration tests that use remote services. The docker-based tests are implemented using `SpringJUnit4ClassRunner` and use an ad-hoc database instance and mailhog for mocking email sending and receiving functionality. We only consider the regular unit tests and the docker-based tests, since calling external services would increase the deviation of our measurements heavily.

3 Continuous Performance Measurement in GeoMap

Regular unit tests can be measured directly by repeating the workload inside of one JVM and repeating the JVM start several times [9]. For the docker-based tests, the measurement is done using Kieker and the docker startup is configured.

Performance Measurement With Kieker – The measurement of plain JUnit tests can be done by repeating a JUnit `Statement`. `@Before`, `@After` and their class-level equivalents can be included or excluded by configuration, which enables skipping the setup for workload repetition or not measuring it as part of the workload duration. For custom test runners, this is impossible, since JUnit wraps the test runners workload inside the tested `Statement`. To still achieve fine control over what workload is measured, we use Kieker source instrumentation[3] directly in the test method.

Configuration of Docker Start – Since the start of docker containers is time-consuming, we decided to start the containers once per execution and measure only the tests that were compatible with this approach, i. e. tests that did clean up the database.

4 Results

In this section, we first give an overview over the regular commits that occured during our case study period and afterwards discuss how performance improvements could be reproduced using continuous performance measurement of unit tests.

4.1 Regular Commits

During our examination period of 6 month, 185 commits were analyzed in the backend and 55 commits were analyzed in the libraries. From the backend commits, 138 build successfully[4] and contain changed source code, and 89 of them contained unit tests that covered changed source code. From the library commits, 43 were analyzable and contained changed source code, 13 contained unit tests that covered the changed source code. The code in the libraries that is not tested in their unit tests is partially tested indirectly in the backend. Since we only measured the master branch, some of the commits were merge commits and therefore change huge parts of the source code. Of the overall 240 analyzed measured test cases, 21 contained performance changes. The root causes of these performance changes were grouped by manual analysis into three categories:

Functional Changes – Changed functional requirements cause added, removed or changed operations at method level. These cause both performance regressions and improvements. In our analyzed timeframe, 21 testcases (in 11 commits) had changed performance due to functional changes. Since the functional changes were necessary and were not causing huge slowdowns, no action was taken on them.

Testcase Changes – To improve test coverage or to fix flaky tests, the testcases themselves were changed. This usually causes performance changes, e. g. because added assertions consume time. This partially hides other performance changes in the same commit. This happened in 9 testcases (in 6 commits).

Optimizations – Due to reasoning on the code, developers detect performance optimization options, e. g. using `PoolingHttpClientConnectionManager` instead `BasicHttpClientConnectionManager` improved the performance. These improvements could be confirmed by the measurement in Peass-CI. It affected to 9 testcases (in 3 commits).

Version Updates – Updates of versions of third-party libraries like the database client both improved and detoriated the performance. Since checking the root causes in third party libraries would be very time-consuming, we accepted these performance changes. This affected 6 testcases in one commit.

Since the measured commits did not contain performance regressions caused by inefficient API usages or algorithms, no action was taken based directly on the measurements. Out of the 21 commits changing the performance, four performance changes were also measurable by GeoMaps load tests. One additional performance change was detected by the load tests. Therefore, while unit test measurement cannot replace load tests, it can be a proxy for performance changes. Accordingly, the developers conceived the overview of performance changes as an improvement for their insights into the performance evolution of GeoMap and continue to check for performance changes with Peass-CI.

4.2 Performance Improvements

By (partially automated) code review and manual experimentation, we found four performance improve-

[3]`https://github.com/kieker-monitoring/kieker-source-instrumentation`

[4]Between releases, the backend relies on Snapshot versions of the libraries, which led to incompatilities on our build server. We compared the last commit were the backend was compilable to the first commit were the backend was compilable again.

ment options. In the following, we describe the performance improvements.

Code Review – By code review together with the developers, we detected three types of performance improvements: (1) Inefficient `StringBuilder` usage: Instead of using `StringBuilder`, `StringBuffer` was used and the `append` calls were distributed over different statements[5] (2) Inefficient `Pattern` usage: Pattern were recreated on every method call with constant parameters. (3) Unnecessary call: It was checked whether a database entry exists and this information was never checked again. Based on this input, the developers created performance improvement commits. The improvements could be measured by Peass-CI.

Manual Experimentation – Since reviewing every class is time-consuming, the developers decided which part of the software is most relevant in terms of performance and implemented load tests for them. By review of the load test results, we detected an implementation error: A blacklist for data insertion was read in every call and appended to a not-resetted list. This caused increasing overhead during the operation of the system. The problem was measurable by a load test. Using Kieker source instrumentation and analysis of the measured data, the root cause could be determined. After fixing it, the performance improvement was directly measurable by a unit test.

5 Related Work

Different tooling exists to include performance benchmarking into CI, e.g., Stochastic Performance Logic [2] and JMH[6]. Only ≈0.4 % of open source projects use such tools [8]. There exist case studies of performance benchmarking in CI and of application of performance engineering methods on repository histories.

Benchmarking in CI – Waller et al. [5] describe how regression benchmarking was introduced to the Kieker CI process. Heger et al. [3] applied a root cause analysis approach at SAP and detected an unknown performance regression. In contrast to our work, these works use benchmarks written for performance measurement instead of unit tests.

Performance Engineering on Repository Histories – By a retrospective analysis of which changes or problems performance engineering methods would have identified, differents works prove the effectiveness of their approach. Chen et al. [7] show that performance changes can be detected by measurement of existing unit tests and microbenchmarks in Hadoop and RxJava. Pradel et al. [4] show that their performance testing of concurrent classes can identify performance problems on existing repositories. Foo et al. [1] show that by applying their regression testing measurement data analysis methods, performance problems can be detected that analysts overlooked before. In contrast to our work, these works do a restrospective analysis and identify performance problems, while we examined performance changes during the software development process.

[5]`append` calls should be written in one statement as described by `https://pmd.github.io/latest/pmd_rules_java_performance.html`.

[6]`https://github.com/openjdk/jmh`

6 Summary and Outlook

We examined the performance of the real-estate tool GeoMap using Peass-CI during the development. We were able to spot 21 performance differences and implemented four performance improvements. We plan to examine the usage of Peass-CI on other projects to get a more detailed view of how performance changes over time and how continuous performance measurement influences the software development process.

Acknowledgments 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] K. C. Foo et al. "Mining Performance Regression Testing Repositories for Automated Performance Analysis". In: *ICQS '10*. IEEE. 2010.

[2] L. Bulej et al. "Capturing Performance Assumptions Using Stochastic Performance Logic". In: ICPE '12. Boston, Massachusetts, USA: ACM, 2012, pp. 311–322.

[3] 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.

[4] M. Pradel, M. Huggler, and T. R. Gross. "Performance regression testing of concurrent classes". In: *ISSTA '14*. ACM. 2014, pp. 13–25.

[5] J. Waller, N. C. Ehmke, and W. Hasselbring. "Including Performance Benchmarks into Continuous Integration to Enable DevOps". In: *ACM SIGSOFT Software Engineering Notes* 40.2 (Mar. 2015), pp. 1–4.

[6] J. P. Sandoval Alcocer, A. Bergel, and M. T. Valente. "Learning from Source Code History to Identify Performance Failures". In: *ICPE '16*. Delft, The Netherlands: ACM, 2016, pp. 37–48.

[7] J. Chen and W. Shang. "An Exploratory Study of Performance Regression Introducing Code Changes". In: *Proceedings of the 2017 IEEE ICSME*. IEEE. 2017, pp. 341–352.

[8] P. Stefan et al. "Unit Testing Performance in Java Projects: Are We There Yet?" In: *ICPE '17*. ACM. 2017, pp. 401–412.

[9] D. G. Reichelt and S. Kühne. "How to Detect Performance Changes in Software History: Performance Analysis of Software System Versions". In: *Companion of the ICPE '18*. Berlin, Germany: ACM, 2018, pp. 183–188.