Empirical Analysis of Performance Problems at Code Level

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

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

ABSTRACT

Performance problems are well known on architecture level. On code level their occurrences have not been systematically researched so far. Since a lot of everyday work of software developers is done on code level, methods and tools with focus on frequent performance problems are relevant.
In the presented thesis, a method for systematically evaluating the occurrence and the frequency of performance problems on code level is presented and applied to repositories. The results of this empirical research will be a classification of performance problems and a quantification of their frequency. This will raise the awareness on certain problem classes for developers and will provide a basis for the development of new performance tools for preventing performance problems.

1. INTRODUCTION

Software developers aim to choose an implementation with as little performance problems as possible. Choosing the best solution for a given problem is a non-trivial task since performance requirements are manifold, e.g. low answering time versus low memory usage, and since these requirements compete with other requirements, e.g. concerning maintainability or re-usability. Developers rely on their own experiences, ad-hoc measurements or incomplete performance problem lists to avoid performance problems, i.e. implementations with possible improvements regarding performance. A systematic, empirically founded classification of re-occurring performance problems on code level is missing. Such a classification would help developers to avoid unperformant code like regular antipatterns and would help the developer to avoid hard maintainable code.

Therefore, the research questions of the sketched PhD thesis are: (1) Which performance problem classes exist at code level? (2) How often do instances of these problem classes occur? The thesis will provide methods and tools to identify performance problems on code level and create an empirically founded problem classification gained by those methods and tools. This will help developers to decide which code patterns could be avoided. Furthermore, this contribution could be used by researchers and practitioners to tackle performance problems directly when designing tools and methods for performance improvement.

The remainder of this paper is organized as follows: First, the approach for identifying performance problems is introduced. Section 3 describes the current state of the implementation. Section 4 presents related work and explains the novelty of the approach. Section 5 presents the research plan and time schedule. Finally, a summary of the paper is provided.

ICPE'16 March 12-18, 2016, Delft, Netherlands
© 2016 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-4080-9/16/03.
DOI: http://dx.doi.org/10.1145/2851553.2892038

2. APPROACH

The performance-optimal implementation of functional or non-functional requirements is not known a-priori. Therefore, performance problems cannot be detected from code directly. Existing code repositories contain a vast amount of performance problems which are introduced and reverted [1]. The introduction or fixture of a problem results in a measurable performance change. The set of performance changes in the code history of a software project therefore provides the basis to identify performance problems.

In the version history of a project, performance problems are present if a performance change occurs between two distinct versions and this change is not caused by a functional behaviour change[1]. If a performance change is a regression, i.e. a degradation of performance measurement values, the performance problem is located in the newer version. If the performance change is an improvement, the performance problem is located in the older version. By finding these performance problems in a sufficient number of projects and classifying them, an empirically founded classification of performance problems can be derived.

Performance changes, which are not caused by a functional behaviour change, may be due to other causes:
(1) Changes trying to fulfil other performance requirements, e.g. a higher answering time may be caused by a change which decreases memory usage. (2) Changes trying to fulfil other non-functional requirements, e.g. requirements considering maintainability. The first type of change is caused by a performance trade-off and therefore no performance problem. It will not be marked as performance problem. The second type of change is a performance problem, even

[1]Functional behaviour changes are not equivalent to functional requirement changes. If a bug is fixed, this is not a requirement change, but a behaviour change, and it may cause a performance change.

if is caused by a reasonable trade-off between performance requirements and other requirements. Therefore, it will be marked as performance problem. A consideration whether a performance problem is acceptable due to a trade-off with other requirements is beyond the scope of this thesis.

An ideal basis for the performance analysis of a software development project would be a sufficiently documented set of load tests. Since most publicly available repositories do not maintain load tests we make the following "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. This assumption does hold for frameworks, in which most methods could be called in different contexts and thus become performance relevant, and for isolated backend components, where the performance of the executions themselves mainly influences the performance of the program as a whole. It does not hold for components in enterprise applications that make heavy use of other services because the performance of these services may mainly drive performance of the component.

Based on the "unit test"-assumption it is possible to detect performance changes of a program by detecting performance changes of unit tests. Therefore, this thesis will detect performance problems by comparing performance measurements of unit tests and detect changes by manually inspecting code changes that cause measurable performance changes. This method is called Performance Analysis of Software System Versions (PeASS) [12] [11]. It is planned to save detected performance problems with metadata, e.g. revision, class and expected type, into a problem database. Based on an analysis of multiple projects, a quantification of the occurrence of those problems will also be performed. These can be done based on the performance problem database created before. Furthermore, other research questions, e.g. if performance problem introduction and solving is executed by one or many committers, can be answered by the performance problem database. In the next section, the current implementation of PeASS is described.

3. STATE OF IMPLEMENTATION

Performance measurements are very time-consuming [3]. Since performance changes can only take place when called source code changes[2], measurements need only to be executed if a class called from a test case is changed. Therefore, PeASS contains the following steps: (1) determination of the tests that need to run in each version, (2) measurement of the performance of selected unit tests of selected software versions and (3) identification of performance bugs.

To determine which tests need to be run in each version, we apply change-based test selection. In the beginning, for each test, a list of classes which are called by a test is determined (I). Afterwards, for every version, it is determined based on the version control system diff whether a test has to be executed in this version (II). The test has to be executed iff the test itself or a class called from the test is changed in the current version. If the test has to be executed in the current version, the called classes of this test are re-determined since changes may introduce new called classes (IV). The construction of dependencies is done using Kieker [15], its AspectJ instrumentation and trace analysis. This process is displayed in 1.

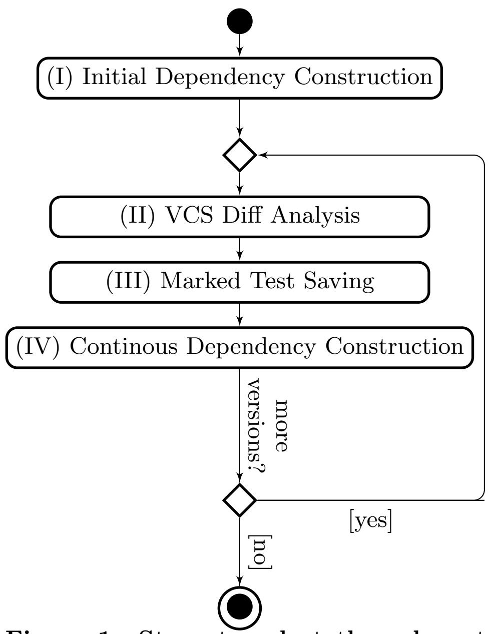
(I) Initial Dependency Construction
(II) VCS Diff Analysis
(III) Marked Test Saving
(IV) Continous Dependency Construction
more versions?
[yes]
[no]

Figure 1: Steps to select the relevant tests, from: [11]

Afterwards, the performance measurement is executed. Since performance measurements are imprecise due to background processes, JIT compilation, thread scheduling and garbage collection [3], a test harness, which produces replicable results, has to be established. While jmh[3] provides such functionality, adapting performance tests for running in jmh implies considerable effort for transforming the Java source code and setting up the run environment with classpath and variable for the surefire-run without using surefire itself. Therefore, the test process with multiple sequential vm runs, test repetition for a warmup phase until a steady state is reached and the warmup executions themselves has been implemented. Currently experiments are carried out to determine the count of vm runs, warmup executions and measurement executions in order to find performance changes as fast as possible. After executing the tests, which are determined by the change-based test selection, performance changes are detected. This is done by comparison of the confidence intervals of the average values of the vm runs.

After performance changes have been identified, they are manually inspected. Every change is marked according to whether the change is necessary due to a change of a functional requirement. If it is a pure performance change, it will be classified manually. Currently, performance problem class candidates such as inefficient exception handling, inefficient exit conditions and inefficient concurrent processing have been identified based on a first run of PeASS on Apache Commons IO. With the execution of experiments with more executions and on more projects a more fine-grained classification will be gained.

[2]The fraction of performance problems due to other causes, e.g. changes of configuration files, is ignored. It is assumed that these changes are rare incidents.

[3]http://openjdk.java.net/projects/code-tools/jmh/

4. RELATED WORK

Work related to the PhD thesis can be found in three research fields: (1) empirical analysis of repositories for performance properties, (2) empirical analysis of documentation artifacts for performance problems and (3) work on performance antipatterns.

In order to analyse performance properties of repositories (1), several work exists analysing the performance of a version history directly [1] [5] and analysing the performance of the version history considering special properties of the software [6] [4] [10].

[1] aims at analysing the evolution of performance during software development. This goal is pursued by analysing performance changes of predefined benchmarks for projects for Pharo[4]. These benchmarks are adjusted in order to skip the call of functionalities which are not present in some of the analysed versions. The existence of performance changes during the lifecycle of a software is proven. Furthermore, they classify the changes in negative changes, like *Composing Collection Operations*, and positive changes, like *Deleting Redundant Method call*. Goal and method slightly differ from the approach of this thesis: Performance changes and not performance problems should be detected in [1], and constant benchmarks are used instead of unit tests. Since in [1] the bugs are detected in functionalities that stay stable over time, a bias may be introduced, as developers might consider performance of stable functions more relevant than performance of new functions. [5] describes a method for determining the history of energy consumption of an application over the last revisions. Therefore an hard- and software environment capable of measuring the performance of an application over several versions is defined. Only the method is presented, performance problem classes are not described.

[6] analyses the performance of a repository in order to show the applicability of Stochastic Performance Logic (SPL) for performance unit testing. Based on commit messages it is checked whether the intended performance change, i.e. an improvement of the performance, has really taken place because of a commit. This is tested by hand-written SPL unit tests. It is shown that performance does not always change as the developer beliefs. [4] shows a method to find the root cause of performance regressions in a defined range of revisions. This is a valuable part of a performance analysis of a repository. Nevertheless, an analysis itself is not carried out for all versions. [10] analyse the performance of repositories for performance issues of concurrent usage of classes. Concurrent performance tests are automatically generated using the interface of a class. Performance regressions related to behaviour under concurrent usage are detected.

Several work discusses the occurrence of performance bugs through analysis of documentation artifacts (1) [17] [7] [9] [8]. These works analyse bug tracker or commit messages in order to answer different research questions. [7] analyses 109 performance bugs and classifies them into *Uncoordinated Functions*, *Skippable Functions*, *Synchronization Issues* and *Others*. Furthermore, the introduction, exposure and fixture of performance bugs are discussed. [9] analyses the introduction and fixture of performance in repositories of established software projects. They find among others that performance bugs are found more often by reasoning on the source code than functional bugs. [17] addresses research questions comparing performance and security bugs by an analysis of Firefox. Similar work was also done for special systems, e.g. for android applications [8]. They find that performance bugs are harder to fix than non-performance bugs and that problem effects (e.g. *Energy Leak*) and bug patterns (e.g. *Lengthy operations in main threads*) differ from other contexts. This works provide valuable findings about performance bugs in bug trackers. They may find bugs on architecture level since these bugs are also likely to be reported. Their results may lack unreported bugs. This gap is filled by PeASS which is capable of finding all performance bugs at code level which are covered by a unit test.

The work on performance antipatterns on architecture level (3) describes antipatterns and provides methods to find them in existing models. [13] analysed performance in practical projects and found 14 antipatterns ranging from the concrete *Unnecessary Processing* to the rather abstract *Falling Dominoes* which occurs when failure of one component causes the failure of another component. This is the first comprehensive listing of performance anti-patterns. Nevertheless, the performance anti-patterns have not been empirically grounded. A quantification of their occurrence has not taken place. The antipatterns described in [13] are put in a hierarchy in [16]. Furthermore, the Performance Problem Diagnostics (PDD), a method for finding performance problems in a three-tier application, is presented. PDD finds those performance problems by executing systematic experiments. It pursues a complementary approach to this thesis: Based on a-priori known problem classes, it identifies their occurrence. The performance anti-patterns are also used in [2] and [14] to detect performance problems in existing projects. [2] defines OCL queries on UML models for certain rules which formalize performance antipatterns. [14] aims at finding performance problems in Palladio Component Models. In addition, actions providing a solution to the performance problem are defined. These works define and use performance antipatterns based on real world problems. An empirical research on their occurrence has not taken place yet.

5. RESEARCH PLAN AND TIME SCHEDULE

Currently, the main process of PeASS has been implemented, including the transformation of unit tests to performance benchmarks and the statistical evaluation of their results. Furthermore, PeASS has been enriched by change-based test-selection. The remaining tasks of the thesis are: (1) The PeASS process will be finalized by determining optimal parameters for running performance experiments. Furthermore root cause analysis of performance regressions [4] will be adopted. If possible the environment for systematic experiments from DynamicSpotter[5] will be used. (2) Experiments with big repositories should be carried out. For this, open source libraries will be evaluated. It is planned to evaluate Apache Commons IO, BCEL, BeanUtils and Collections. Furthermore isolated backend components, where performance is mainly driven by unit performance, should be evaluated, if suitable software is provided. The resulting performance changes should be tagged as performance problem or as performance changes caused e.g. by functional changes. Furthermore, the resulting performance problems

[4]http://pharo.org/

[5]http://sopeco.github.io/DynamicSpotter/

will be classified by manual inspection. The classes should be compared to the classes created by [1] and [13]. This will be done while the experiments run. (3) Research questions arising additionally, which could be answered by the created performance problem database, e.g. about frequency and timeliness of performance problem occurrence, will be answered. (4) In the end, the thesis will be written.

The presented tasks are summarized in table 5 together with their planned time schedule.

(1)	Until 06 / 2016	Finalizing the PeASS-process
(2)	Until 03 / 2017	Running experiments, classification
(3)	Until 06 / 2017	Additional research questions
(4)	Until 09 / 2017	Finalizing thesis

Table 1: Planned time schedule of the PhD thesis

6. SUMMARY

This paper presents a thesis about the Empirical Analysis of Performance Problems at Code Level. Its main goal is to identify performance changes on code level by analysing the performance of unit tests in the version history of a project. These changes should be analysed in order to derive performance problem classes. The occurrence of those problem classes can be quantified afterwards based on measured data. Since the performance of unit tests only matters for programs where performance is mainly driven by them, this method is mainly applicable to frameworks and backend components with little outgoing calls.

The first implementation of the tool for PeASS is completed. The next steps in the PhD thesis are the completion of the tool, its application to repositories and the classification of performance problems. This classification of performance problems will assist developers to make better decisions on implementation versions regarding their performance.

Acknowledgement This work was funded by the German Federal Ministry of Education and Research within the project Competence Center for Scalable Data Services and Solutions (ScaDS) Dresden/Leipzig (BMBF 01IS14014B)

7. REFERENCES

[1] J. P. S. Alcocer and A. Bergel. Tracking down performance variation against source code evolution. In *Proceedings of the 11th Symposium on Dynamic Languages*, DLS 2015, pages 129–139, New York, NY, USA, 2015. ACM.

[2] V. Cortellessa, A. Di Marco, and C. Trubiani. Performance antipatterns as logical predicates. In *Engineering of Complex Computer Systems (ICECCS), 2010 15th IEEE International Conference on*, pages 146–156. IEEE, 2010.

[3] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous java performance evaluation. *ACM SIGPLAN Notices*, 42(10):57–76, 2007.

[4] C. Heger, J. Happe, and R. Farahbod. Automated root cause isolation of performance regressions during software development. In *ICPE 13*, pages 27–38, New York, USA, 2013. ACM.

[5] A. Hindle, A. Wilson, K. Rasmussen, E. J. Barlow, J. C. Campbell, and S. Romansky. Greenminer: A hardware based mining software repositories software energy consumption framework. In *MSR 2014*, pages 12–21, New York, USA, 2014. ACM.

[6] V. Horký, F. Haas, J. Kotrč, M. Lacina, and P. Tůma. Performance regression unit testing: a case study. In *Computer Performance Engineering*, pages 149–163. Springer, 2013.

[7] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and detecting real-world performance bugs. In *Proceedings of the 33rd ACM SIGPLAN PLDI*, PLDI '12, pages 77–88, New York, USA, 2012. ACM.

[8] Y. Liu, C. Xu, and S.-C. Cheung. Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of the 36th ICPE*, pages 1013–1024. ACM, 2014.

[9] A. Nistor, T. Jiang, and L. Tan. Discovering, reporting, and fixing performance bugs. In *MSR 2013*, pages 237–246. IEEE Press, 2013.

[10] M. Pradel, M. Huggler, and T. R. Gross. Performance regression testing of concurrent classes. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 13–25. ACM, 2014.

[11] D. G. Reichelt and F. Scheller. Improving performance analysis of software system versions using change-based test selection. In *Symposium on Software Performance*, 2015.

[12] D. G. Reichelt and J. Schmidt. Performanzanalyse von softwaresystemversionen: Methode und erste ergebnisse. In *Software Engineering & Management 2015, Multikonferenz der GI-Fachbereiche Softwaretechnik (SWT) und Wirtschaftsinformatik (WI), FA WI-MAW, 17. März - 20. März 2015, Dresden, Germany*, pages 153–158, 2015.

[13] C. U. Smith and L. G. Williams. More new software performance antipatterns: Even more ways to shoot yourself in the foot. In *CMG Conference*, pages 717–725. Citeseer, 2003.

[14] C. Trubiani and A. Koziolek. Detection and solution of software performance antipatterns in palladio architectural models. In *ACM SIGSOFT Software Engineering Notes*, volume 36, pages 19–30. ACM, 2011.

[15] A. van Hoorn, J. Waller, and W. Hasselbring. Kieker: A framework for application performance monitoring and dynamic software analysis. In *Proceedings of the 3rd joint ACM/SPEC International Conference on Performance Engineering (ICPE 2012)*, pages 247–248. ACM, April 2012.

[16] A. Wert, J. Happe, and L. Happe. Supporting swift reaction: Automatically uncovering performance problems by systematic experiments. In *Proceedings of the 2013 ICPE*, pages 552–561. IEEE Press, 2013.

[17] S. Zaman, B. Adams, and A. E. Hassan. Security versus performance bugs: a case study on firefox. In *MSR 2011*, pages 93–102. ACM, 2011.