Improving Performance Analysis of Software System Versions Using Change-Based Test Selection

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

Fabian Scheller
Universität Leipzig
scheller@wifa.uni-leipzig.de

ABSTRACT

The detection of performance bugs by measurements is very time-consuming since measurements must be repeated often in order to get reliable results. Especially in performance analysis of software system versions (PeASS), where performance bugs are detected by comparing the performance of unit tests of different revisions, the testing process takes much time. Using change-based test selection enables to execute only those performance unit tests which may have changed results. This leads to a significant reduction of test execution time.

1. INTRODUCTION

The first measurable performance change of Apache Commons IO[1] after moving to the current repository (in 560660) was in revision 584162. Since the option to decide wether file filter tests should be performed case sensitive is added, certain unit tests gets slower. Using classical PeASS, a method for finding performance changes in a projects version history introduced in [8], the performance change would have been found after 6320 test executions. 16 revisions would have been tested before. In these revisions mainly XML-files or source files that are not accessed from the present test are changed. Change-based test selection solves this problem by selecting only those tests where the called code is changed instead of re-running all tests in every revision. To detect the given performance change 428 test executions are needed. Change-based test selection accelerates PeASS and allows for producing results much quicker.

This paper describes the application of change-based test selection in PeASS. In section 2, the method of PeASS is introduced. In section 3, the change-based test selection is described and applied. In section 4 related work is presented. Finally, in section 5 the method is summarized and the forthcoming steps are outlined.

[1]Official website of Apache Commons IO: https://commons.apache.org

2. PERFORMANCE ANALYSIS OF SOFTWARE SYSTEM VERSIONS

Every modification of a software can lead to a change in its performance. Currently, there has been no empirical research on the extraction of performance changes that took place in the history of a project. A method to find these changes would both enable the extraction of changes for scientific use, e.g. classifying and quantifying the occurence of performance changes, and the extraction of performance changes in real software projects. This chapter describes the method of PeASS, first introduced in [8], which fills this gap.

The basic assumption of PeASS is, that the performance of unit tests corresponds to the performance of the program. Thus, there are performance bugs which cause performance changes of unit tests. Based on this assumption PeASS measures the performance of all unit tests and analyses in which revision performance bugs could occur. The novelty of PeASS is that a whole repository is examined by re-running each unit test of every revision as performance test. In other works less tests are executed, e.g. by running only a fixed set of tests [4] or analyzing only a fixed set of versions [2].

PeASS contains three steps: (1) measurement of performance of all unit tests of all software versions, (2) identification of performance changes and (3) identification of performance bugs. The remainder of this chapter describes these steps in more detail.

For the measurement of the performance of all unit tests of all software versions, every revision is checked out at the beginning. Therefore, the version control system, e.g. SVN or git, has to be used automatically. Subsequently, unit tests are converted into performance unit tests. The performance unit tests are executed several times with warm-up executions to avoid inaccuracy due to background processes and in a java-environment due to the optimizations of the JVM. This is done by annotating the unit tests with KoPeMe-framework annotations [7]. Furthermore, the build process needs to be instrumented, i.e. performance test dependencies need to be added and interfering plugins, e.g. Cobertura and Apache Rat, must be disabled. This is done by changing Ant- or maven-files. Afterwards the tests are executed.

After the results are collected, performance changes are identified. Since the measurement for every revision is very time-consuming, it is executed less often. Therefore changed performance measures do not necesserily correspond to real

changes. To overcome this, re-measurements are done for change candidates which are determined by heuristics. In sum the identification of performance changes is done in three steps: (1) Change candidates are identified by heuristically analysing the performance measurements. (2) Re-measurements are performed. As only a few tests are performance change candidates, this test remeasurement are executed more often. (3) Source code is manually checked.

Finally, it is checked whether performance changes are performance bugs. It is possible that the measurement changes are only caused by test changes or by functional requirement changes. If neither of both is the case, the performance change is a performance bug, that has been either introduced or fixed in the new commit.

The main shortcoming of this process is that it is very time-consuming or the results have low precision. The presented change-based test selection overcomes this problem.

3. CHANGE-BASED TEST SELECTION

Performance measurements stay the same if called code and the execution environment are not changed. Therefore in PeASS remeasurement of tests with non-changed source code in the test itself and called code are not necessary. In order to skip unnecessary test executions, the process follows the steps indicated: (1) application of change-based test selection in order to determine 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. The remainder of this chapter describes these steps.

In step 1 for every revision the test classes that need to be called are identified. These are saved to a Revision Test File (RTF). The generation of the RTF is done by (i) Initial Dependency Construction and for every version: (ii) VCS Diff Analysis (iii) Marked Test Saving and (iv) Continous Dependency Construction. Figure 1 displays this process.

Before the tests can be selected, in *step i* the classes that are called from the test classes (from now on called dependencies) need to be initially identified. This is done by an initial run of all tests instrumented with Kieker [9]. For the current implementation this is done by enabling Kieker measurement in KoPeMe and adding the AspectJ instrumentation. Afterwards the traces constructed by Kieker are analysed using *ExecutionRecordTransformationFilter*, *TraceReconstructionFilter* and *KoPeMeFilter*. The *KoPeMeFilter* extracts the called classes for each test case. For the initial revision all tests are marked as changed in the RTF.

In *step ii* for each software version the tests that should be re-run are determined. In order to do this the changed classes in the new software version are read from the VCS diff. This is currently only possible for SVN and Maven projects. The SVN diff is read via a command-line call and afterwards, assuming standard maven file structures, the full qualified class names of the changed classes are infered. Every test class is marked as class that needs to be run where a dependent class changed. In *step iii* the tests that need to be run in the current revision because of an code update are saved in the RTF.

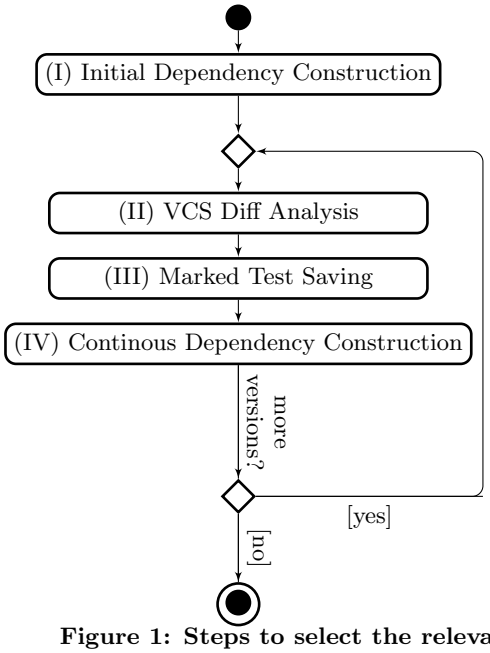

Figure 1: Steps to select the relevant tests

Project	Normal Tests	Reduced Tests
Commons BCEL	100521	12109
Commons BeanUtils	97178	8261
Commons Collections	44058	842
Commons IO	659190	77259
Commons Jelly	298	13

Table 1: Counts of normal and reduced tests for chosen Apache Commons projects

Since the changed classes could introduce or remove dependencies, the dependencies for tests that are marked as changed need to be re-determined. This is done in *step iv* by calling the tests with the same Kieker instrumentation and analysis as during the initial trace reconstruction.

Table 1 shows how many tests are executed in a normal execution and with test selection. It shows that change-based test selection saves a considerable amount of test calls. Currently, classes are also marked as changed if non-code parts of classfiles, e.g. documentation or whitespaces, are changed. This could be solved if the code is parsed and those changes are ignored. A realisation is planned in order to further decrease the amount of reduced tests.

For the measurement run itself in step 2, the test process needs to be changed. This is currently implemented for maven shurefire tests. It is done by running the tests from the RTF for each test method with -Dtest=Class1#MethodA. The tests are run as usual in PeASS. To avoid overhead, Kieker-AspectJ instrumentation is excluded in this step.

In step 3, performance changes with variation over a boundary value are identified. It is investigated by manual inspection whether they are performance bugs. In the future this step will be supported by additional tools.

Even if change-based test selection is an elaborate task, it

accelerates the process. Examining 428 tests with 4000 iterations each took 8 h 4 min on a Thinkpad T400. Executing 6320 tests would have taken approximately 119 h assuming equal duration per test.

The change-based test selection could be implemented by static code analysis. For this, imports and package-calls would be included in the dependent classes. With this approach it would be hard to determine which parts of the code are really executed and which are only imported. Therefore this implementation alternative was refused.

A shortcoming of this method is that the performance could change even if the change-based test selection detects no change. This could happen when a configuration file with performance-influencing flags is changed. In the experiments on Commons IO, no such influence could be identified. Nevertheless this might be the case in other environments.

4. RELATED WORK

Related work exist in two fields: (1) mining software repositories (MSR) analyses the data available in software repositories and (2) regression test selection (RTS) tries to minimise test time.

In MSR green-mining tries to analyse the development of energy consumption during versions, e.g. [3] analyses how energy consumption of software in Android-applications changed. The difference between this approach and PeASS is that energy consumption is measured instead of other performance properties and that there is no method for test selection. Furthermore [6] and [5] analyse performance bugs in public available bug tracker. They analyse the development of performance properties by their documentation in bug tracker instead of code measurement. Therefore their information about performance bugs are gained more indirectly.

In RTS [10] it is detected which functional unit tests need to be executed again if software changes. In this context a *regression* is a functional requirement which is not fullfilled anymore, which not necessesarily corresponds to a performance change. The RTS techniques detect the test cases which use modified code like in this work. There is also work which detects redundant tests, test suite minimization, and work which prioritises tests in order to detect errors more quickly. A use for performance-related issues has not been done yet according to [10]. There are implementations which run regression tests every time code changes, e.g. infinitest.[2]

5. SUMMARY AND FUTURE WORK

Performance bugs in repositories are efficiently detectable by PeASS with change-based test selection. For every revision of a repository, the performance of all tests with changes in itself or changes in called classes is evaluated for every revision. In order to do so the call hierarchy of all tests is reconstructed and maintained by Kieker.

The main shortcoming of PeASS is that the identification of performance bugs is time-consuming and error-prone. To overcome this, root cause isolation of performance regressions [2] will be used to support performance bug detection.

[2]http://infinitest.github.io/

The change-based test selection method could be enhanced to be usable in the performance test process. This could be done for load tests, performance tests using stochastic performance logic [1] and regression performances tests written by KoPeMe. Three adjustments are necessary if change-based test selection method is to be enhanced: (1) The test measurement process needs to be changed. In maven this could be done either by enhancing surefire or by writing another plugin that writes the includes into the pom.xml. (2) The system under test needs to be instrumented with Kieker. For load tests on separate machines, the results had to be collected afterwards. (3) An adapter for the used VCS has to be written in order to extract the changed classes from the diffs.

6. REFERENCES

[1] L. Bulej, T. Bureš, J. Keznikl, A. Koubková, A. Podzimek, and P. Tůma. Capturing performance assumptions using stochastic performance logic. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, ICPE '12, pages 311–322, New York, NY, USA, 2012. ACM.

[2] 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.

[3] 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.

[4] V. Horkỳ, 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.

[5] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and detecting real-world performance bugs. *ACM SIGPLAN Notices*, 47:77–88, 2012.

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

[7] D. G. Reichelt and L. Braubach. Sicherstellung von performanzeigenschaften durch kontinuierliche performanztests mit dem kopeme framework. In *Software Engineering*, pages 119–124, 2014.

[8] 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.

[9] 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.

[10] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120, 2012.