How to Detect Performance Changes in Software History: Performance Analysis of Software System Versions

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

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

ABSTRACT

Source code changes can affect the performance of software. Structured knowledge about classes of those changes could guide software developers in avoiding negative changes and improving the performance by positive changes. Neither a comprehensive overview nor a mature method for structured detection of those changes exists for this purpose. We address this research challenge by presenting **Pe**rformance **A**nalysis of **S**oftware **S**ystems (PeASS). PeASS builds up a comprehensive knowledge base of changes affecting the performance of a software by analyzing the version history of a repository using its unit tests. It is based on a method for determining the significant performance changes between two unit tests by measurement and statistical analysis. Furthermore, PeASS uses regression test selection for saving measurement time and root cause isolation method for performance changes analysis. We demonstrate our methodology in the context of Java by analyzing the versions of Apache Commons IO.

KEYWORDS

Performance Testing, Mining Software Repositories

ACM Reference Format:
David Georg Reichelt and Stefan Kühne. 2018. How to Detect Performance Changes in Software History: Performance Analysis of Software System Versions. In *ICPE '18: ACM/SPEC International Conference on Performance Engineering Companion , April 9–13, 2018, Berlin, Germany.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3185768.3186404

1 INTRODUCTION

In commit 1d0c2d, error handling in Apache Commons $IOUtils.copy$ was changed from throwing an exception to return a value. If this value is -1, the execution of the method is considered as erroneous. Therefore the execution time in 1d0c2d of $IOUtilsCopyTestCase.testCopy\_inputStreamToOutputStream\_IO84$, which tests an erroneous case, is reduced by 50%. Creating an exception in Java introduces computation time cost, since the stack is saved into an exception. Therefore, when introducing and removing exception handling, performance aspects should be considered.

ICPE '18, April 9–13, 2018, Berlin, Germany

ACM ISBN 978-1-4503-5629-9/18/04…$15.00
https://doi.org/10.1145/3185768.3186404

Source code changes affecting performance, like in 1d0c2d, occur in software projects frequently [1]. Those performance changes at code level, i.e. performance changes which are measurable by unit test, might improve or degrade the performance. By elaborating a method capable of automatically finding performance changes at source level in the version history of a project, it is possible (1) to classify the performance changes at source level, (2) to find performance regressions in an existing software and, if they are still present, fix them and (3) to understand better, which performance changes occur and when they occur. Therefore, it is a research challenge to find a method for structured detection of performance changes in software repositories. We address this research challenge by presenting Performance Analysis of Software Systems (PeASS). PeASS is able to identify performance changes by measurement of the performance of every version.

This papers presents (1) a description of the method of PeASS, (2) a method for comparing the performance of two versions and (3) the results of a case study which analyzed Apache Commons IO.

The remainder of this paper is organized as follows: Section 2 gives an overview about the PeASS method. Section 3 describes the method for comparing the performance of two software versions. The results based on the described method are presented in section 4. Afterwards, section 5 describes related work. Finally, a summary and an outline of future work are given.

2 METHOD

In order to determine performance changes, the central step of PeASS is the measurement and comparison of the performance of the code. Since this takes much time, regression test selection determines which test needs to be run in which version. After changes are found, they need to be understood. Therefore, root cause isolation is executed after measurement. Finally, the found changes are classified. All in all, detection of performance changes is done by the steps indicated: (I) Regression test selection in order to determine tests that need to be run, (II) distinguishing performance by measurement, (III) performance change root cause isolation and (IV) performance change classification. These steps will be described in the remainder of this section. They are summarized in figure 1.

In order to consume as little test time as possible, *regression test selection* (I) selects tests with potentially changed performance for every version. Performance tests rely on execution of the same load more than once in order to produce statistical reliable measurements. Since a one time execution of a test is usually fast, it is feasible to execute every test once in order to determine whether the execution trace changed and therefore the performance of two versions may differ. By analyzing this single execution and the

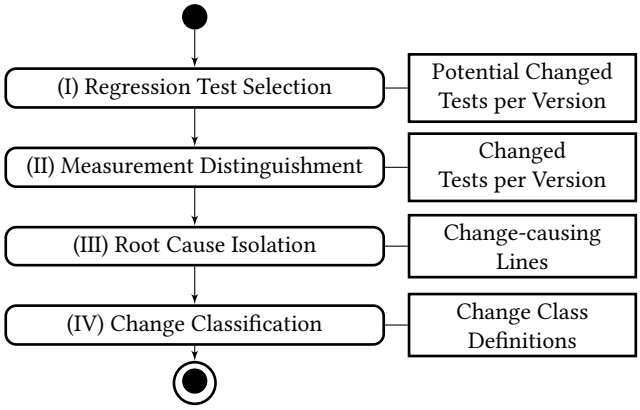

Figure 1: Steps of PeASS

diff of the version control system, it can be determined which test needs to be executed in which version. A regression test selection based on this idea is implemented [14]. Its result is a list of potential changed tests for every version.

For detecting performance changes, it is necessary to *distinguish the performance* of two different software versions (II). This could be done by code analysis, e.g., by model based performance prediction [3] or by measurement. Measurement provides a reliable empirical basis to observe small changes. Since we focus on performance changes at code level and those may be small, we apply measurement. This results in a list of changes per version.

Software performance is the efficiency regarding time and resource consumption. Usually, it is measured by exposing a system to a load, which is specified in load tests or benchmarks. Since most repositories do not maintain a set of benchmarks or load tests, those cannot be used for measuring the performance of a software [20]. Therefore, 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, if the performance is not driven mainly by external factors. This holds for environments where the performance of one execution of the program units mainly drive the overall performance, like backend components or end user applications and for environments where every method could be called in different contexts and therefore become performance relevant, like in libraries or frameworks. The unit test assumption does not hold in environments where performance is mostly driven by calls to other services, like enterprise applications, or by parallel executions, like database systems. Furthermore, using unit tests has technical limitations: For comparison of performance by a unit test, the unit test needs to be unchanged or the API needs to be unchanged, so the old test can be used, and functional utilities for unit testing, like mocks, must not mainly drive the tests performance.

In some special cases, unit test performance might not correlate with application performance: Performance regressions at unit level are not always regressions for the application, i.e., introduction of a cache may slow down a unit test but improve the system performance, and performance improvements at unit level are not always improvements for the application, i.e., initialization of an *ArrayList* with smaller array size may improve the unit test performance but slow down the overall performance. Furthermore, functional behavior changes may cause performance changes.

Since unit tests usually have high code coverage, they mostly cover source paths where performance changes are caused. Therefore, they are able to detect a big part of all performance changes in environments where the unit test performance correlates to the use case performance.

In order to measure the performance of unit tests, they need to be transferred to performance unit tests. Since existing methods are missing the capability to precisely identify a performance change in an unit test, we develop an measurement and analysis method in order to examine whether a performance change exists between two test case versions. This method is presented in section 3.

After a performance change is identified, we need to *isolate the root cause* (III) of the change. Therefore, we get traces by instrumentation of multiple executions of the test case. In order to assign the methods of the old trace to the methods of the new trace, we use maximal bipartite matching. Afterwards, we compare the performance measurements by statistical tests. While first experiments show promising results, this is still an active research challenge.

Based on the performance change root causes, *performance change classification* (IV) is done. Therefore, the properties of change classes need to be developed. Classification might be done based on different properties, like type of changed code elements, magnitude of change, time of change in relation to the next release date or intention of change. While some of the properties can be detected automatically, others, like the intention, can only be detected automatically in some cases, e.g., if the intention is written in the commit comment. Currently, the whole process of classification is done manually. We plan to first gather a set of classes and afterwards automate the assignment of classes by analyzing the elements of the changed code.

3 DISTINGUISHING PERFORMANCE

In order to distinguish the performance of two unit tests by measurement, a measurement method and an analysis method need to be defined. We define an approach capable of assessment of the quality of the methods, i.e., capable of determining how good measurement method and analysis method are able to distinguish tests. This approach will be described in the first subsection. Afterwards, the creation of the measurement method will be described. Finally, the determination of measurement parameters and the selection of an analysis method will be described.

3.1 Approach

In order to assess the quality of a measurement and analysis method and their parameters, we (I) define artificial unit test pairs, where it is known whether their performance differs, (II) execute the measurement and analysis with those methods and (III) get precision and recall based on the measurement results. If precision and recall are below acceptable thresholds, we repeat measurement and analysis. This process is visualized in figure 2.

In general, the artificial unit test pairs should cover all possible workload type and size combinations in order to be a proper test set for a generic measurement and analysis method. This could be done

by defining unit test pairs for all workload types and automatic combination of those types. To simplify matters, we chose to implement only five artificial unit test pairs[1]. If we recognize that the resulting measurement and analysis method and their parameters are not able to distinguish performance measurements correctly, we extend those tests.

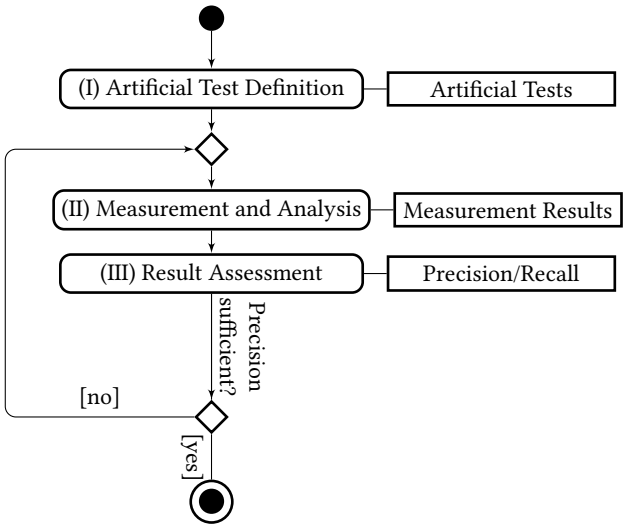

Figure 2: Steps of Approach for Distinguishing Performance

The artificial test pairs are: (1) Addition of 10 and 11 random numbers, (2) printing 10 and 11 random numbers to system out, (3) addition of 10 random int and 10 random long numbers, (4) adding 3 ints and 4 ints to a long and (5) entering a try-catch block with throwing an exception 10 and 11 times. Given each pair of tests, a measurement configuration and a statistical method need to be able to correctly identify that a performance change happened, e.g., that adding 10 random numbers is faster than adding 11 random numbers. Furthermore, they need to determine that executions of the same test case have no change in execution time. Therefore, the statistical method needs to identify two subsets of the measurements of one equal test as equal.

In order to implement the performance unit tests, we use the KoPeMe framework [17]. That framework for measuring the performance allows to extend a JUnit-test in order to specify a count of warmup iterations and executions which are executed. The results including all individual measurements are saved afterwards. Since this fulfills our requirements, we chose KoPeMe for measuring the performance.

3.2 Measurement Method

Performance could be represented by different measurements, e.g., time, CPU, memory or energy consumption. We concentrate on time consumption, since time consumption is the most noticeable performance property to the end users. Since the characteristic of other performance measurements, e.g., memory, is different, the measurement and analysis methods can not be re-used.

[1]These can be found in the repository precision-experiments, available in https://github.com/DaGeRe/precision-experiments

Influences like parallel processes or memory partitioning result in different execution time for every execution. Since we cannot reliably predict the effects of those effects, we consider time consumption a random variable characterized by a statistical distribution. The duration of a function in a managed environment like Java is influenced additionally by parallel processes inside the Java Virtual Machine, garbage collections, compilations and optimizations. They result in different measurements and optima of the same use case [6]. It is possible to tune parameters in order to avoid or force garbage collection, compilation and optimization events. Since every test case has different memory reservations and thus different garbage collection behaviour, the configuration of these parameters needs to be different for every test case. If we tuned those parameters, the performance may differ from real-world scenarios, where tuning might be done differently. Therefore, we consider the unit test as a black-box and omit tuning of those parameters.

Comparing the performance of two unknown unit tests is difficult, since (1) the kind of workload is unknown, (2) the duration of unit tests is low compared to the inaccuracy of the time function and (3) the variation of time measurements due to measurement errors is relatively high compared to the variation of performance measurements due to source code changes. Since the time consumption is a random variable, we use the measurements as input for a statistical test which determines whether a performance change happened.

When measuring performance in managed environments, the startup performance, the performance during optimization and compilation, or steady state performance, the performance after those processes, could be measured [6]. Since performance changes, that are only measurable during warmup, disappear after the warmup, we decide to measure steady state performance.

The usual approach is to sequentially start the virtual machine (VM) vm times, execute the use case w times for the warmup and m times for the measurement [6]. Since it is unclear, how many executions are needed, it is recommended to choose a count of iterations k, measure the coefficient of variation v, i.e. the standard deviation of k measurements divided by its mean and finish execution iff v drops below a threshold which should be 0.01 or 0.02 [6].

We could reuse the method if it would be able to identify all changes correctly. We evaluated this method using the artificial unit tests. In those tests, the coefficient of variation was not reaching the recommended threshold nor was it a proper indicator for reaching the steady state. Figure 3 shows the average mean and average coefficient of variation for both alternatives of test case (1) for 30 VM executions for every iteration.[2] The average mean stays at the same level after about 5000 iterations, but the average coefficient of variation is changing and not falling below 0.3 during the measurement. This example shows that the coefficient of variation is not a good indicator for reaching the steady state: While the measurement values nearly stay equal and indicate that the steady state is reached, the coefficient of variation still stays high.[3]

[2]We always summarize 100 measurement iterations in one display point for display purposes.

[3]This measurement can be reproduced by the repository precision-experiments, see section *Comparing Coefficient of Variation* in README.

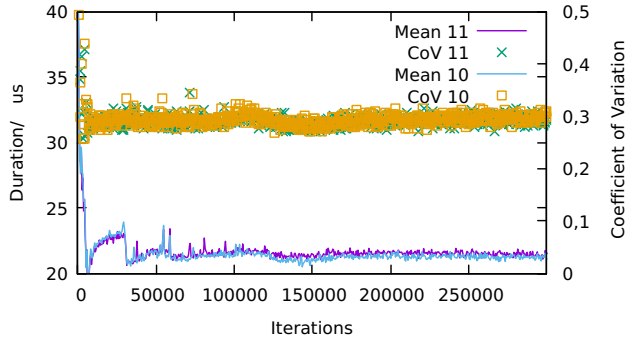

Figure 3: Graph of Average Mean and Average Coefficient of Deviation for Adding Numbers

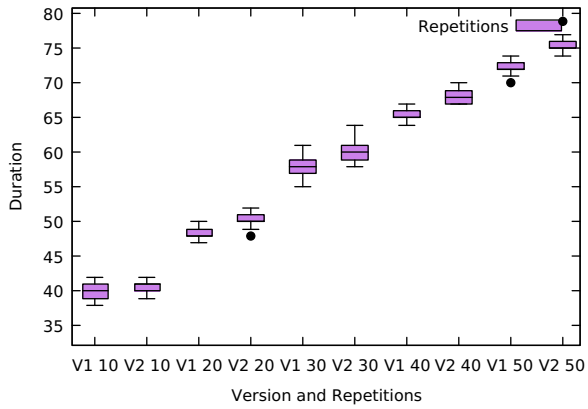

Figure 4: Confidence Intervals of Duration of Test Execution with Different Repetition Count

The performance of the two (1)-test cases is nearly equal, as shown in figure 3. This is still the case if we execute the measurements further. We assume that this happens because the measurement inaccuracy is too big compared to the workload itself. Therefore, we chose to measure after we have repeated the benchmark multiple times. Figure 4 shows the evolution of the average difference of the measurements for 10.000 iterations with no warmup. This shows that, if we repeat two different benchmarks, the difference gets bigger and therefore analysable. Unfortunately, we lose the ability to measure performance changes which only happen when executing a small count of repetitions, e.g., because the garbage collector is triggered earlier in the measurements, but not more often.[4]

3.3 Measurement Parameters and Analysis Method

With a sufficient high amount of repetitions, warmup executions, measurement executions and VMs, it is possible to decide whether two measurements are equal. Since we want to execute performance

[4]This measurement can be reproduced by the repository precision-experiments, see section *Comparing Different Repetition Counts* in README.

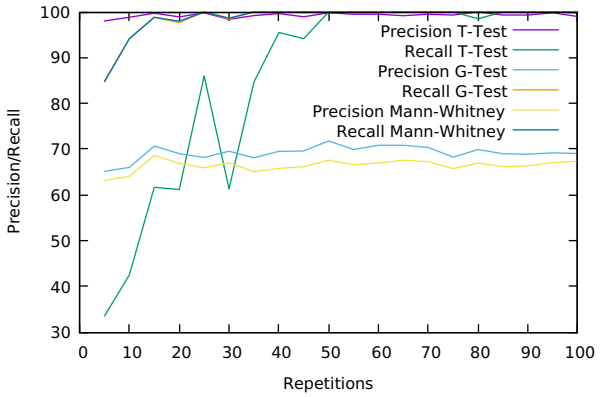

Figure 5: Precision of T-Test, G-Test and Mann-Whitney-Test

comparisons as fast as possible, we search for the parameters and statistical method with the lowest measurement time and precision. We assume that the performance comparison is correct with a sufficient high count of iterations and warmup executions. Therefore, we chose 50.000 iterations and 60 VMs. We determine after the measurement how many iterations and VMs could be skipped without changing the result. In order to determine the amount of repetitions, we tested different repetition counts from 1 to 100.

In order to analyze the measurement results, we tested different methods for comparing the results. Confidence Interval Comparison, T-Test and Mann-Whitney-Test are recommended for comparing two performance measurements [4] [6]. We used those and additionally tested G-Test. These statistic tests assume the null hypothesis that two given distributions are equal. Every test states the conditions under which this null hypothesis can be disproved. For Confidence Interval Comparison, the α-confidence intervals with confidence level α of both distributions are determined and if they do not overlap, the distributions are considered different. The other tests use a significance level, which states the maximum probability of rejecting the null hypothesis when it is true. We use 0.05 as significance level. T-Test assumes normal distribution and equal means and Mann-Whitney-Test and G-Test assume independence of the measurement values. Since we do not know which assumption holds for our workload, we checked whether the tests are able to distinguish two measurement results correctly.

In order to determine which statistic test to use, we took i iterations and w warmup iterations from v VM executions from the artificial test results of both versions. Afterwards, we determined if the test identified them as performance change correctly. If so, it is considered a true positive, else a false negative. Afterwards, we took v VM executions from the same version and determined if the test correctly identified it as no performance change. If so, it is considered a true negative, else a false positive. We repeated this 1000 times for every test and every tested repetition count. Based on the values, we determined precision and recall of every method.

Figure 5 shows graphs of precision and recall. Confidence Interval Comparison only finds performance changes if the intervals do not overlap. Since the tests have a small difference, this happens rarely and Confidence Interval Comparison is therefore not capable

of finding changes at all. While T-Test has a high precision, G-Test has a high recall. Since the T-Test still gets a recall of more than 80%, we decided to use it for comparing performance results.

We use the same method for determining the count of VMs, iterations and warmup iterations: We execute the tests with more VMs, warmup and measurement iterations than needed and determined which parameters with minimal execution duration have sufficient precision and recall. We assume that the overall duration d of one measurement is $d = vm * (overhead + duration * (w + i))$ where vm is the count of VM executions, $overhead$ is the constant time overhead needed for checking out a version, compiling etc., $duration$ is the average execution time of one measurement, w is the count of warmup executions and m is the count of measurement iterations. The $duration$ is higher than the actual duration, since the time function call and the data processing needs more time than the unit test itself. We determined the average overhead time and the average execution time based on measurement timestamps, which were taken along the duration measurement.

We determined that the average overhead is 4018 *ms* and the execution duration is on average 11.5 *ms*. We search for a configuration with a precision above 99 %, a recall above 95 % and the lowest execution time. Therefore, we split the 50.000 iterations in up to 25.000 warmup and measurement iterations. We determined that 60 repetitions, 5 VMs, 1000 warmup and 4000 measurement executions are sufficient.[5] In summary, determining a change in the artificial test cases needs on average 49,5 minutes.

Furthermore, we researched whether an early stop of VM executions if the result is clear would decrease precision or recall. Therefore, we executed our analysis and stopped adding VMs if the t-value is above or below a theshold. We found that we could prune further VM executions without loss of precision and recall, if VM 10 executions took place and the t-value is above 10 or below 0.1.

4 CASE STUDY: APACHE COMMONS IO

In order to examine the usability of the PeASS approach, a mature project with a big count of commits, maven-managed build process and functional unit tests written in JUnit should by analyzed. We decided to analyze Apache Commons IO[6]. Apache Commons IO provides basic functionalities for input and output, has currently 2157 commits with 114 test cases in its current version. Since it is used widely[7], the performance of its functions has an impact on a wide range of other software.

We executed the tests on a cluster with 47 servers managed by slurm[8]. Every server had an Intel(R) Xeon(R) CPU E5-2620 processor with 24 cores á 2,4 GHz and 128 GB RAM. We analyzed 254 versions and found 93 changed test results. We classified them manually into 6 classes. These classes, their frequency and the average absolute change are summarized in table 1.

The classes are defined based on intention of the change and used code elements. *Exception Handling*, *Synchronisation* and *Library* are based on code elements. They could be detected automatically by checking whether a change contains a try, catch throw statement, a synchronized statement or an import statement. An example is the initially presented change in 1d0c2d, where in $IOUtils.copy$, a return code marks the result instead of an exception. This reduces the execution time of $IOUtilsCopyTestCase.testCopy\_input - StreamToOutputStream\_IO84$ by 50%.

Name	Count	Increase	Decrease	|Change|
Functionality	46	12	34	2018,6 %
Condition Checking	15	15	0	217,4 %
Optimizations	12	7	5	3512,1 %
Exception Handling	7	4	3	2903,7 %
Synchronization	3	3	0	1097,6%
Library	1	1	0	89,7%
Test	9			

Table 1: Performance Changes by Classes

The *Functionality*, *Condition Checking* and *Optimizations*-classes are based on intentions. They could be detected in some cases by analysis of the code commit. An Optimization happens in 09a6cb in $FilenameUtilsWildcardTestCase.testMatch2$, while calling $toArray$. The method $toArray$ of a $List$ takes an Array as input. If this array has the size of the List, the entries of the list are copied to the array; otherwise, a new array is initialized. Using an array that has exactly the size of the list therefore speeds up the test case. In this case, the commit comment contains "optimisation" and might therefore be automatically detected.

While the mentioned test cases are corner tests, the found changes might also change the performance of live systems using Apache Commons IO: Both, copying files and searching for files by Wildcard, could be used in practice and be sped up by the particular change. Therefore, we expect to find relevant change classes by analysis of more projects.

5 RELATED WORK

PeASS analyzes repositories of a project in order to detect and classify performance changes. We presented the first step of PeASS, the distinguishing of the performance of two versions, in detail. In the following, we discuss work that addresses the (I) search and classification of performance changes in software repositories and (II) the measurement of performance changes.

Analysing repositories in order to detect performance changes (I) can use (Ia) the documentation, i.e., commit comments, code documentation or issue tracker information, or (Ib) the code itself.

Analysis of the documentation (Ia) has been done for different kinds of desktop applications [9] [19] [15] [22] and Android applications [12]. Work analyzing performance bugs is also able to classify root causes of performance problems, e.g., Jin et al. [9] identify uncoordinated functions, skippable functions and synchronizations issues as reasons of performance changes. While those works are able to identify and classify performance problems, the completeness of the problems is unclear since only performance regressions actively recognized by users or developers will be listed.

Analysing code repositories (Ib) could aim for detecting performance changes or performance regressions. Alcocer et al. [1]

[5] This measurement can be reproduced by the repository precision-experiments, see section *Determining Parameters* in README.
[6] Github Page of Apache Commons IO: https://github.com/apache/commons-io
[7] Apache Commons IO is used by 11 445 artifacts in maven central according to https://mvnrepository.com/artifact/commons-io/commons-io/2.5
[8] Official site of slurm scheduler: https://slurm.schedmd.com/ The script for executing slurm can be found in misc/scripts/slurm/ in the PeASS-Repository

[18] identify and classify performance changes based on the measurement of long running benchmarks provided by projects of the Pharo framework. They find five classes of changes that decrease performance and four classes of changes that improve performance.

There exists work which executes existing load test or benchmarks [5], own generated parallel tests [16] or own generated load [13] in order to find performance regressions between versions. Heger et al. [7] identify root causes of performance regressions by git bisect and trace analysis. Furthermore, the tool hopper [11] analysis the change of performance of Java applications and the tool GreenMiner [8] provides a hardware framework for determining energy consumption changes during software versions. While those works analyze performance issues by version history, this work transforms existing unit test to performance tests and provides a statical rigorous technique for distinguishing the measurement results. This makes it applicable to more projects, since many projects contain unit tests, but the used tests may be less suitable.

Work providing methods for measuring performance (II) mostly contains a measurement and an analysis part. Georges et al. [6] summarizes methods for measuring the performance and analyzing the results. Furthermore, they provide a own measurement method which we used as base for definition of our measurement method. Kalibera et al. [10] provide statistical rigor by manually determining how many executions are needed in order to reach the steady state. Barrett et al. [2] extend this work by automatically defining when a steady state is reached by change point analysis. They find that only 43,5 % of all benchmarks and VMs reach the steady state.

Furthermore, Stochastic Performance Logic [4] defines a language and a tool for specifying a formula that defines whether a performance change is detected. For measuring the performance in a test environment, tools like jmh[9] and JUnitBench[10] exist. Tools for monitoring the performance of systems, like Kieker[21], Caliper[11], AppDynamics[12] and Dynatrace[13] are less related since they focus on measuring live systems.

6 SUMMARY AND FUTURE WORK

We presented the challenge of defining a method for structured detection of performance changes in software repositories. We adress this challenge by presenting the method PeASS, which is able to identify performance changes if the performance of unit tests correlates to the performance of relevant use cases of the application. In order to distinguish the performance of unit tests, we created a method which is capable of measuring the performance of unit tests and determining whether a performance change has taken place. Performance measurement of unit tests is done by repeating the workload until the duration of the test itself overweights the duration and variation of the time measurement. The applicability has been demonstrated in terms of detection of performance changes in Apache Commons IO.

The next step is to facilitate analysis of results by root cause isolation of performance changes and providing a unified format for definition of performance change classes.

[9]Website of JMH: http://openjdk.java.net/projects/code-tools/jmh/
[10]Website of JUnitBench: https://github.com/tpounds/junitbench
[11]Repository of Caliper: https://github.com/llnl/Caliper
[12]Website of AppDynamics: https://www.appdynamics.com
[13]Website of Dynatrace: https://www.dynatrace.com

ACKNOWLEDGEMENTS

This work was funded by the German Federal Ministry of Education and Research within a PhD scholarship of Hanns Seidel Foundation and within the project Competence Center for Scalable Data Services and Solutions Dresden/Leipzig (ScaDS, BMBF 01IS14014B). Computations for this work were done with resources of Leipzig University Computing Centre.

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] E. Barrett, C. F. Bolz-Tereick, R. Killick, S. Mount, and L. Tratt. Virtual machine warmup blows hot and cold. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):52, 2017.
[3] S. Becker, H. Koziolek, and R. Reussner. The palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82(1):3–22, 2009.
[4] L. Bulej, T. Bureš, V. Horký, J. Kotrč, L. Marek, T. Trojánek, and P. Tůma. Unit testing performance with stochastic performance logic. *Automated Software Engineering*, 24(1):139–187, Mar 2017.
[5] J. Chen and W. Shang. An exploratory study of performance regression introducing code changes. In *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*, pages 341–352. IEEE, 2017.
[6] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous java performance evaluation. *ACM SIGPLAN Notices*, 42(10):57–76, 2007.
[7] 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.
[8] 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.
[9] 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.
[10] T. Kalibera and R. Jones. Rigorous benchmarking in reasonable time. In *ACM SIGPLAN Notices*, volume 48, pages 63–74. ACM, 2013.
[11] C. Laaber and P. Leitner. (h, g)opper: Performance history mining and analysis. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ICPE '17, pages 167–168, New York, NY, USA, 2017. ACM.
[12] 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.
[13] Q. Luo, D. Poshyvanyk, and M. Grechanik. Mining performance regression inducing code changes in evolving software. In *Proceedings of the 13th International Conference on Mining Software Repositories*, pages 25–36. ACM, 2016.
[14] T. H. Nguyen, B. Adams, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora. Automated detection of performance regressions using statistical process control techniques. In *ICPE*, pages 299–310, New York, USA, 2012. ACM.
[15] A. Nistor, T. Jiang, and L. Tan. Discovering, reporting, and fixing performance bugs. In *MSR 2013*, pages 237–246. IEEE Press, 2013.
[16] 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.
[17] D. G. Reichelt and L. Braubach. Sicherstellung von performanzeigenschaften durch kontinuierliche performanztests mit dem kopeme framework. In *Software Engineering*, pages 119–124, 2014.
[18] J. P. Sandoval Alcocer, A. Bergel, and M. T. Valente. Learning from source code history to identify performance failures. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, ICPE '16, pages 37–48, New York, NY, USA, 2016. ACM.
[19] M. Selakovic and M. Pradel. Performance issues and optimizations in javascript: an empirical study. In *Proceedings of the 38th International Conference on Software Engineering*, pages 61–72. ACM, 2016.
[20] P. Stefan, V. Horky, L. Bulej, and P. Tuma. Unit testing performance in java projects: Are we there yet? In *Proceedings of ACM/SPEC ICPE 2017*, pages 401–412. ACM, 2017.
[21] 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.
[22] 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.