PeASS: A Tool for Identifying Performance Changes at Code Level

1st David Georg Reichelt
University Computing Centre
Universität Leipzig
Leipzig, Germany
reichelt@informatik.uni-leipzig.de

2nd Stefan Kühne
University Computing Centre
Universität Leipzig
Leipzig, Germany
stefan.kuehne@uni-leipzig.de

3rd Wilhelm Hasselbring
Software Engineering Group
Christian-Albrechts-Universität zu Kiel
Kiel, Germany
hasselbring@email.uni-kiel.de

***Abstract*—We present PeASS (Performance Analysis of Software System versions), a tool for detecting performance changes at source code level that occur between different code versions. By using PeASS, it is possible to identify performance regressions that happened in the past to fix them.**

PeASS measures the performance of unit tests in different source code versions. To achieve statistic rigor, measurements are repeated and analyzed using an agnostic t-test. To execute a minimal amount of tests, PeASS uses a regression test selection.

We evaluate PeASS on a selection of Apache Commons projects and show that 81% of all unit test covered performance changes can be found by PeASS. A video presentation is available at https://www.youtube.com/watch?v=RORFEGSCh6Y and PeASS can be downloaded from https://github.com/DaGeRe/peass.

I. Introduction

While 73% of all developers still rely on manual testing and reviews for detection of performance issues, only 17% use tools [11]. If more developers used tools automating the assurance of performance, manual efforts would be reduced.

To automatically examine and assure performance, performance can be measured. Performance measurement faces three main problems: (1) Performance measurements need to be repeated to produce statistically reliable results [7]. (2) Creation of performance benchmarks is time-consuming. (3) Performance measurements take, due to the high number of repetitions, long. Only 68.75% of all projects produce results in less than four hours [18].

We present the tool PeASS[1], which (1) repeats measurements to produce statistically reliable results, (2) uses transformed unit tests for measuring the performance and therefore does not require manual effort for performance benchmark definition and (3) provides a regression test selection and therefore does only execute performance tests if the executed source has changed. We present an evaluation of PeASS using Apache projects. We show that 81% of all existing and unit test covered performance changes are identified by PeASS.

The remainder of this paper is organized as follows: First, Section II describes the approach of PeASS. Afterwards, Section III describes a systematic evaluation on Apache Commons projects. Section IV compares the approach of PeASS to related work. Finally, Section V gives a summary.

Hanns-Seidel-Foundation http://www.hss.de

[1]Used version of PeASS: https://doi.org/10.5281/zenodo.3402855

II. Approach

In this section, we describe the approach of PeASS. We start with an overview and then describe the two steps of PeASS, the regression test selection and the measurement method.

A. Overview

The performance of software in production environments emerges from the interplay of its software architecture, implemented code and hardware. PeASS focuses on finding performance changes which are *observable at code level*. Performance changes due to inefficient architecture, e.g. excessive number of requests for serving one task (empty semi trucks [17]), or performance changes caused by too small or inefficiently configured hardware are beyond the scope of PeASS.

To find performance changes at code level, performance in different versions needs to be measured. This could ideally be done by code written especially for measuring the performance of the software under test, i.e. benchmarks or load tests. According to a case study of Stefan et al., only approximately 3.4% of all projects maintain benchmarks [18]. Unit tests in contrast are maintained in 33% of all projects. 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 is applicable if the performance of the software itself mainly impacts the overall performance, like in backend components, end user applications, or libraries. It is not applicable if the performance is affected by calls to external services, which may happen in microservice architectures. Furthermore, it is not applicable if the performance is determined by the efficiency of parallel executions, e.g. in database systems.

Based on the unit test assumption, we can compare the performance of two equal unit tests in two versions to identify whether the performance of the tested code units has changed. Furthermore, unit tests are usually not written for measuring performance. Therefore, (1) functional utilities mainly changing performance, like mocks, could make measurements unusable and (2) performance at unit test level may not always correlate with application performance. For example, the introduction of a cache may slow down a unit test but improve

978-1-7281-2508-4/19/$31.00 ©2019 IEEE
DOI 10.1109/ASE.2019.00123

the system performance [15]. However, using transformed unit tests for performance testing (1) enables to test a large share of a program, since usually many unit tests exist which cover a high share of code paths, (2) requires no manual effort and (3) requires no test generation.

To get statistically reliable performance values, the measurements need to be repeated. Therefore, measuring performance has a high duration: Only 65% of all projects containing performance benchmarks get results in less than 4 hours [18]. To reduce execution times, we prune the set of tests which need to be measured by performance regression test selection (Step (1)). Afterwards, we execute the performance measurement and statistical analysis (Step (2)). The approach are summarized in Figure 1.

Fig. 1. Approach of PeASS

We record a performance change as the pair of version and affected test. If a performance change at code level happens, it is caused by a source change. This source change is unique for every version. However, the effect of the source change may differ depending on the test: A version may change class A and B, but tests T_A and T_B may be influenced in completely different ways by calling class A or B. For example, T_A may get slower and T_B may get faster, since the called methods of class A contain more operations and the called methods of class B contain less operations.

B. Regression Test Selection

A test needs to be executed if its outcome, here its performance, may have changed. PeASS' regression test selection identifies exactly those tests for which source code changes to previous versions can have an impact on performance. It relies on the idea, that the performance of a benchmark may only change iff the executed source code changes. Since PeASS assumes equal hardware and execution environments, this holds for the converted performance tests of PeASS.

To determine the selected tests, two steps are executed: (1) a static test selection determines which tests may have changed based on static code analysis and (2) a trace analysis determines whether the execution trace, i.e. the sorted list of executed methods and their source, has changed. Details can be found in [14].

To speed-up regression test selection, we parallelized the regression selection process. Therefore, we needed a parallelized application of the static selection rules and the trace analysis. The results of the static selection rules impact each other: If the dependencies of an prior version change, e.g. if a called method is added, this may influence the selected tests in a subsequent version, e.g. if the code of this called method is changed. Therefore, we cut the versions into slices. Each slice has the size of the overall version count divided by the available thread count. Each slice can be analyzed in parallel. Afterwards, the results are merged. Since the dependencies need to be initially constructed in each thread, this increases overall computation effort. Therefore, parallelization is only feasible until the overhead for initial version processing outweighs the benefit of parallelization. The parallelization of the trace analysis is straightforward: Each version can be run and analyzed in parallel, since they do not impact each other. Since most projects contain more versions than computers contain CPU cores, parallelization of the trace analysis is mostly only limited by the number of available CPUs.

C. Measurement

To compare the performance of two versions, the performance needs to be measured and the measurements need to be analyzed. Since response time is the most noticeable performance property of code units and more complex properties, e.g. database usage, service usage, are not generically derivable, PeASS focuses on response time.

The default approach for measuring performance in virtual machines [7] assumes that measuring the response time of a workload is non-deterministic since random effects, e.g. inaccuracy of time function, parallel processes, memory partitioning and in case of a managed environment like Java garbage collections, optimizations and just in time compilations, influence every measurement. Therefore, the measurements need to be repeated in a VM until a steady state is reached, i.e. until optimizations and just in time compilations are finished. This is assumed to happen when the *coefficient of variation*, i.e. variation of the last k values, falls below a threshold, e.g. 0.01 or 0.02. Since optimizations and compilations may lead to different optima and therefore different measurements, the VM needs to be started multiple times, usually in a fixed count of iterations. After the measurement, the distributions need to be compared. If the type of distribution is known, statistical tests suited for distributions could be used, like the two-sample t-test for Gaussian distributions. Otherwise, like for unknown workloads, a comparison of confidence intervals can be used [7]. Updates of this method aim for reducing the number of warmup iterations or VMs [1] [2] [10].

Our previous work [15] shows that using two-sided t-test is efficient for determining performance changes of unit test sized benchmarks. Unfortunately, the t-test only states that two performance measurements are unequal. Therefore, we use the agnostic t-test, which is able to determine whether two measurements are unequal, equal or whether with the given significance level a decision is not possible [6].

All parameters, VM starts, number of warmup iterations, measurement executions and significance level, may be parameterized when executing tests.

III. Validation

To validate PeASS, we research which share of existing changes can and cannot be found by PeASS, i.e. the amount

of true positives and false negatives. We do not research the amount of false positives or true negatives.

To find out which share of empirically existing changes are found by PeASS, we need a set of versions containing real performance changes. Unfortunately, such a set is not defined. Since developers usually document performance improvements in the commit comments, we build such a validation set by analyzing the commit comments of developers. Therefore, we identify commits with intended performance changes by automatically checking the commit comments for keywords, including "performance", "slow" and "fast", and manually remove the commits with no intended performance change (e.g. "Enable maven cache for in travis config").[2]

We use 8 Apache Commons projects, the library jackson-core and the App k-9 Mail for the validation, since they have a long version history and defined quality standards. We execute PeASS on the versions which we identified as performance change intending. We use 500 warmup iterations, 500 measurement iterations, 100 repetitions, at maximum 200 VM starts and a significance level of 99% on Intel Xeon E5-2620 v3 2,4 GHz processors, running CentOS 7.6.1810 and OpenJDK 1.8.0_112.

Source changes, which are documented by the committer may still be not detected as changes or misinterpreted by PeASS for the following reasons: (I) Unit tests may contain the wrong *Workload* for finding performance changes, since they test inadequate input sizes or border cases. (II) The performance may remain *Unchanged* in current JVMs. (III) The change may be *Unselected* by the regression test selection. This may happen due to source not being covered by tests, non-source configuration files changing the performance or test trace size above the limit. We checked all of our filtered commits with performance change intention which case applies. Table I summarizes the results.

Project	compress	csv	dbcp	fileupload	io	imaging	pool	text	jackson	k-9	Sum
Selected	22	4	5	5	4	3	9	1	15	5	73
Correct	13	4	5	2	3	2	8	1	14	4	56
Workload	6	0	0	3	1	0	1	0	1	1	13
Unchanged	3	0	0	0	0	1	0	0	0	0	4
Unselected	4	0	0	0	1	2	3	1	6	15	32
All	26	4	5	5	5	5	12	2	21	20	105

TABLE I
CATEGORIES OF PERFORMANCE CHANGE RESULTS

105 changes are documented in the commit comments in the part of the history which can be currently built. 70% (73) of the changes are selected by PeASS. The others are not covered by tests. By a manual analysis of the changes, we found that 4 of these changes do not impact the performance, although intended by the committer. From the remaining 69, 81% (56) performance changes are found by PeASS and 19% (12) are not found due to the workload. Assuming that this is representative for commits impacting performance, PeASS is able to find 81% of all changes on code level which are impacting performance if the affected source section is covered by unit tests. While this is far from a full coverage, it still implies that without additional implementation effort, more than two out of three performance changes can be found. If we take into account performance problems not covered by tests, still 55 % (56 out of $105 - 4$) are found.

Performance changes which may not be found because the tests *Workload* is not suitable (I). This can happen if the test itself creates much noise in measurement: In commit `59ffca` in Commons IO, depicted in Listing 1, the `Tailers` read performance is enhanced by reading into a buffer instead of reading every sign alone. Unfortunately, the test contains multiple calls to `Thread.sleep` with 50 ms waiting time. Therefore, the performance cannot be measured by repetition of the unit test and the performance change is not found.

Listing 1. Unsuitable Workload

```
...
tailer = Tailer.create(file, listener,
    delay, false);
Thread.sleep(idle);
tailer.stop();
...
```

Furthermore, the workload may also be unsuitable because of its size: In commit `65128` in Commons Fileupload, depicted in Listing 2, the performance of `DiscFileItem` is improved because an `BufferedInputStream` is used on top of a normal `FileInputStream`. In real world use cases, this will often improve the performance because the input is read into a buffer, instead character by character. In the test case, only a few character are read. Therefore, this leads to a slowdown of the test case.

Listing 2. Unsuitable Workload

```
-FileInputStream fis = null;
+InputStream fis = null;

 try {
- fis=new FileInputStream(dfos.getFile());
+ fis=new BufferedInputStream(new
    FileInputStream(dfos.getFile()));
   fis.read(fileData);
```

Furthermore, the performance may remain *Unchanged*, even if the committer intended a performance improvement. An example for this is in commit 4e7672 in Apache Commons Compress, depicted in Listing 3. While replacing the call to `Math.pow` by a bitshift could reduce operations, we did not observe performance changes neither in the concrete transformed test case nor other benchmarks. As a side effect, this change may lead to a different functional behavior if the variable `bits` contains more than six lowest-order bits.

Listing 3. Unchanged Performance

```
- final long max=(long) Math.pow(2, bits);
+ final long max=1l << bits;
```

Furthermore, the source changes may not lead to a selected test (III). This happens e.g. in Apache Commons Imaging in commit `061b73`, since there are no tests covering the change which was intended to impact performance.

[2]All data from the evaluation can be found at https://doi.org/10.5281/zenodo.3252668

IV. Related Work

Related work tries to find either (I) performance changes or (II) performance problems. In the following, we discuss related work in both fields.

Tools finding performance changes (I) usually rely on measurements. Measurements can be done by load testing tools like JMeter or benchmarking tools like jmh[3] or SPL [3]. Benchmarks have been included into continous integration environments [18] [21]. Standardized integration of load tests into continous integration processes is provided by the MMS performance signature[4]. Furthermore, the production performance can be measured using monitoring tools like Icinga[5], dynatrace[6] or Kieker [20]. Profiling tools like VisualVM or Java Mission Control support analysis of change root causes.

Different authors use existing benchmarks for finding performance changes [4] [16] or generate benchmarks based on special criteria, e.g. to find concurrency issues [13] [8] or energy consumption issues [9].

Tools and methods searching specifically for problems (II) do this at (a) source code level or (b) architectural level.

Tools finding performance problems at source code level (IIa) usually rely on defined antipattern at code level. These known antipattern are common knowledge, but still may vary between different versions of underlying software layers. They are usually encoded in regular expressions, like in PMD[7]. Scientific works derive new patterns for special application types, like Android applications [12], and identify problems using these patterns. These tools are helpful for finding performance problems with well-defined source code patterns.

Performance problems at architecture level (IIb) are well-researched [17]. Methods for finding them include using systematic experiments [22] and model analysis [5] [19].

In summary, all these scientific works and tools offer various capabilities for finding performance changes and problems. None of them is able to measure performance at code level only for test cases selected by a regression test selection. This gap is closed by PeASS.

V. Summary

We presented PeASS, a tool for identification of performance changes at code level. We evaluated PeASS on a set of Apache libraries and showed that PeASS is able to detect 81% of all known and unit test covered performance changes.

Future work should research how many performance changes, which are irrelevant to the developer, are detected and how to filter them. This could be done by supporting root-cause analysis and examination of their results. Also, further extensions of PeASS could automate the inclusion in CI-systems. We plan to use PeASS to empirically research which code patterns change performance and based on which intentions performance-changing commits are made.

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

[4]https://github.com/T-Systems-MMS/perfsig-jenkins

[5]https://icinga.com/

[6]https://www.dynatrace.com/

[7]https://pmd.github.io/latest/pmd_rules_java_performance.html

Acknowledgement This work was funded by the German Federal Ministry of Education and Research within a PhD scholarship of Hanns Seidel Foundation. Computations were done with resources of Leipzig University Computing Centre.

References

[1] H. M. AlGhmadi, M. D. Syer, W. Shang, and A. E. Hassan. An automated approach for recommending when to stop performance tests. In *2016 IEEE ICSME*, pages 279–289. IEEE, 2016.

[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] 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.

[4] J. Chen and W. Shang. An exploratory study of performance regression introducing code changes. In *Proceedings of the 2017 IEEE ICSME*, pages 341–352. IEEE, 2017.

[5] V. Cortellessa, A. D. Marco, and P. Inverardi. *Model-Based Software Performance Analysis*. Springer, 2011.

[6] V. Coscrato, R. Izbicki, and R. B. Stern. Agnostic tests can control the type i and type ii errors simultaneously. *arXiv preprint arXiv:1805.04620*, 2018.

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

[8] R. Gu, G. Jin, L. Song, L. Zhu, and S. Lu. What change history tells us about thread synchronization. In *Proceedings of the Joint Meeting on Foundations of Software Engineering*, pages 426–438. ACM, 2015.

[9] 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.

[10] T. Kalibera and R. Jones. Rigorous benchmarking in reasonable time. In *ACM SIGPLAN Notices*, volume 48, pages 63–74. ACM, 2013.

[11] M. Linares-Vasquez, C. Vendome, Q. Luo, and D. Poshyvanyk. How developers detect and fix performance bottlenecks in android apps. In *2015 IEEE ICSME*, pages 352–361. IEEE, 2015.

[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] 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.

[14] 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*, pages 127–130, 2018.

[15] D. G. Reichelt and S. Kühne. How to detect performance changes in software history: Performance analysis of software system versions. In *Companion of the 2018 ACM/SPEC ICPE*, pages 183–188, 2018.

[16] 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 ICPE*, ICPE '16, pages 37–48, New York, NY, USA, 2016. ACM.

[17] 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.

[18] 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.

[19] 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.

[20] A. van Hoorn, J. Waller, and W. Hasselbring. Kieker: A framework for application performance monitoring and dynamic software analysis. In *Proceedings of the ACM/SPEC ICPE 2012*, pages 247–248, April 2012.

[21] J. Waller, N. C. Ehmke, and W. Hasselbring. Including performance benchmarks into continuous integration to enable DevOps. *ACM SIGSOFT Software Engineering Notes*, 40(2):1–4, 2015.

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