Better Early Than Never: Performance Test Acceleration by Regression Test Selection

David Georg Reichelt
Universität Leipzig
Leipzig, Germany
davidgeorg_reichelt@dagere.de

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

ABSTRACT

Currently, performance tests take much time and are therefore not able to provide fast feedback. Fast feedback on performance tests would support finding performance problems. In order to accelerate performance tests we provide a regression test selection method for performance tests. It is based on test selection by (1) code analysis and (2) trace analysis. We show the efficiency of our approach by comparison with the test selection tools EKSTAZI and Infinitest.

CCS CONCEPTS

• **Software and its engineering → Software performance**; **Software maintenance tools**; **Software testing and debugging**; *Empirical software validation*;

KEYWORDS

Regression Test Selection, Performance Testing, Benchmarking

ACM Reference Format:
David Georg Reichelt and Stefan Kühne. 2018. Better Early Than Never: Performance Test Acceleration by Regression Test Selection. In *ICPE '18: ACM/SPEC International Conference on Performance Engineering Companion , April 9–13, 2018, Berlin, Germany.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3185768.3186289

1 INTRODUCTION

Currently, only 3,4% of all GitHub projects execute performance tests [7]. Assuming a test coverage of performance tests equal to the test coverage of unit tests and 10 minutes test duration, the currently 6582 tests of the application server Jetty would need 45 hours to build per commit. Only 62% of all GitHub projects containing performance tests get results within 4 hours [7]. Since software developers have to wait so long for performance result, it gets harder for them to fix possibly created performance problems. In order to reduce the test time and give rapid feedback, we provide PRONTO (Performance RegressiON Test chOosing), a method for reducing the amount of experiments. The usage of PRONTO facilitates fast feedback on the performance impact of changes and could be used in continuous integration.

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

ACM ISBN 978-1-4503-5629-9/18/04…$15.00
https://doi.org/10.1145/3185768.3186289

We can prune the measurement of tests where the performance stays the same. The performance stays the same if the called code and the execution environment, i.e., used operating system, managed environment etc. do not change. Assuming equal execution environment during benchmarking, only the measurement of tests with changes in the test itself or the called code is necessary. PRONTO determines the performance tests that need to be run in a new version. They will be called *selected tests*. We select them with two techniques: (1) Static Selection Rules, defining when to select a test with static code analysis and (2) Trace Analysis, where two traces, i.e., the sorted list of all method executions of a program, are compared. While the first is faster, the latter is more efficient in selecting performance changes.

We assume that (1) the same workload is executed in every version, (2) some or all performance tests examine only parts of a software and (3) test traces do not change due to load size. Therefore, PRONTO is not able to select tests (1) if the test itself changes, (2) if there exists only one big load test which examines every part of the software or (3) if the system behavior changes with load size, e.g. in stress tests.

The remainder of this paper is organized as follows: Section 2 describes the method of PRONTO, including the test selection by code analysis with static selection rules and the test selection by trace analysis. Section 3 evaluates this work by comparing the selected tests to the tests which are selected by the functional test selection tools EKSTAZI and Infinitest. Finally, section 4 gives a summary and an outlook.

2 METHOD

In order to determine the selected tests of a new version, the old source, the new source and the diff of the sources produced by the version control system can be used. If a test uses any part of the source code that has been changed, this test needs to be re-executed and therefore is selected. Since it is in general undecidable whether a part of the code is called, a static analysis can solve this problem only for special cases. Therefore, a combination of static and dynamic code analysis is applied.

A performance test P usually consists of equal executions p_0, p_1, .., p_n in order to produce statistical reliable results [1]. If the behaviour of p_0 is not changed in a new version, the performance of P cannot change since we assume that all executions are equal. Since the amount of executions n is usually high, a one time execution of p_0 is not consuming too much time for test selection.

We present two steps for determining whether a test needs to be called: (1) Static Selection Rules maintain a set of called code parts and determine whether a test needs to be executed by static code analysis and (2) Trace Analysis runs both versions of a test

and determines which test needs to be run by trace analysis. The following subsections will describe both steps.

The method of PRONTO is implemented for the usage of Java JUnit tests which are transformed to performance unit tests using KoPeMe[1]. A prototype of the described method is available online in the dependency module in the PeASS repository[2]. The prototype determines performance change candidates for Java projects in Git repositories with a Maven build process and JUnit 3 or 4 tests in the version history.

2.1 Static Selection Rules

For fast determination of selected tests, we first select those tests where we can deduce that a change has happened based on static code analysis. We cannot in general determine by static analysis which methods are called. Therefore, we maintain a map from each test case to its called methods, which we refer to as *dependencies*. We will first describe how to determine changed tests based on static code analysis and afterwards how to maintain dependencies.

Test Selection We define that *a statically selected change* (SSC) happens when the source of a method or a called method may have changed. We create static selection rules that define which code change could change the executed code and therefore the performance of a method.[3]

If we have a look at a call to method M in class C, there are three cases that could influence the performance of the method itself: (1) the code of method M itself is changed, (2) there happened a code change in the class C, but outside of their methods or (3) there happened a code change in a sub or super class of C. (1) follows directly from the definition. (2) may influence the performance, since the class code may change the behaviour of its methods, e.g. if a method is overridden that was not overridden in the previous version or if a static value is changed that may change execution paths in the method. (3) could introduce a performance change for the same reasons: A method or a value that is used may be changed. Listing 1 demonstrates this: If the class is changed (red part), every test calling any method of the class needs to be re-evaluated. If only something inside the method changes (green part) only tests calling the method need to be re-evaluated.

Listing 1: Example Source for SSC

```
class MyClass extends MySuper {
  static int var = 3;
  public void methodA() {
     var++;
  }
}
```

The implementation uses javaparser[4] to parse the code. All comments are cleared in the beginning. Afterwards, PRONTO identifies whether there is a change in the class, which indicates a change in all methods, or a change in certain methods, which indicates that those methods are marked as changed. A method change only occurs if the method itself is changed (case 1). A class change happens if a change outside of the method occurs in its class or its sub or super classes, e.g. if a variable is changed or an initialization block is added (cases 2 and 3).

Dependency Maintenance In order to maintain the dependencies, we initially execute every performance test once with instrumentation by the performance monitoring and software analysis framework Kieker [8]. Every method that is called by a test is added to the dependencies of this test. Tests that are changed may introduce dependencies or remove dependencies, e.g. when a tested method calls a new class or stops calling a class. Therefore, the dependencies of each selected test need to be updated after a new version v_i was committed. This is done using the mechanism described above for the changed tests of v_i. Since updated dependencies are only needed for test selection in version v_{i+1}, the tests of v_i may be selected before the dependency update.

Shortcomings Static selection rules also selects changes which are not relevant, for example if a method is added to a class that is not called. Furthermore, performance changes caused by non-source files, e.g. changed libraries, database configuration or VM configurations are not found.

2.2 Trace Analysis

The SSCs indicate where a performance change may have happened. The SSC contains changes caused by (1) newly declared methods, (2) changed method signatures, (3) added variable declarations and (4) initialization blocks. In most cases, these changes do not cause performance changes, since (1) new methods may not be called, (2) changed method signatures may change only visibility, (3) new variables may not be used and make no heavy initialization operations and (4) initialization blocks may only change unused variables. Listing 2 shows this. The method declaration (1), signature (2), variable declaration (3) and variable initialization (4) changes (green part) do not affect the performance of the test *testMe* or any other test existing before the changes. Nevertheless, they are identified as changes by static selection rules. Changes to called methods or the test itself (red part) may influence the performance.

In most cases, performance only changes if the trace changes, i.e. if the source of a called method or the order of called methods is changed. Therefore we determine which test in which version is likely to contain a performance change based on the trace. We call this trace selected changes (TSC).[5]

Determination of TSC is done by (I) instrumented run of both versions, (II) trace annotation with (III) trace reduction and (IV) diff analysis.

In step (I), the test case and its predecessor are run with instrumentation. By using the instrumentation output, traces, i.e. the methods with their signature and order, are created for both versions. Since changes may also happen inside a method, in step (II), these traces are annotated with sources. Therefore, the code of both versions is parsed and the corresponding method sources are derived. In step (III), the traces are reduced. This has two advantages:

[1] GitHub Repository of KoPeMe: https://github.com/dagere/kopeme
[2] Repository of PeASS: https://github.com/DaGeRe/peass
[3] This can be started by using the DependencyReadingStarter from the PeASS-Repository.
[4] Source of javaparser: https://github.com/javaparser/javaparser
[5] This can be started by using the ViewPrintStarter from the PeASS-Repository. It additionally creates diffs of traces with method, which supports manual inspection of a change.

It speeds up automatic analysis and it facilitates manual analysis of the trace differences. In order to compress the trace, it is represented as a context-free grammar with Sequitur and afterwards it is compressed using run-length encoding as described by Reiss and Renieris [5]. A TSC happens iff the method-annotated traces of two versions differ, i.e. if a method signature, the method order or the implementation of at least two methods differs. This is determined in step (IV).

Listing 2: Changes neither affecting trace nor performance.

```
class MyTest {
  int var = 3;
  int var2 = 5; // (iii)
  { var2 = 8; } // (iv)
  protected int methodA() {// (ii)
     return var++;
  }
  public void newMethod() {//(i)
     var++;
  }
  public void testMe() {
     int val = methodA();
     Assert.assertEquals(val, 4);
  }
}
```

Implementation The instrumentation and trace creation are done using KoPeMe and Kieker as described above. In order to parse the source, javaparser is used. The creation of the diff of the traces is done by the Unix *diff* tool. Determination of TSC takes more time than SSC, since two instrumented executions of the tests, the trace annotation and the trace minimization are needed. Therefore, we only determine whether an SSC is also a TSC.

Listing 3: Non Selected Change Affecting Performance

```
class MyTest {
  int var = 5;
  public void testMe() {
     int sum = 0;
     for (int i = 0; i < var; i++) sum+=i;
     Assert.assertEquals(sum, var*var/2);
  }
}
```

Shortcomings There are cases where we may miss performance changes with this method that are found by SSC, e.g. if a conditional execution changes because of a variable initialization change like in Listing 3: TSC would not select a new version is the value of *var* is changed, since the trace would not be changed. SSC would identify a variable initialization change as a code change outside of a method and would therefore select *testMe*. Since AspectJ, the framework Kieker uses for generating its traces, does not allow to instrument e.g., method-local field access or loop iterations, a full logging and therefore a full analysis of such operations is not possible. This could be fixed by an own weaving implementation. Furthermore, not all TSC are real changes - a method may be replaced by another method with the same performance characteristic.

Besides, the current implementation does not handle package name changes. Therefore, if package names are changed, the tests are recorded as new tests. Furthermore, the current implementation is not capable of handling multi-module projects and unit tests that use custom test runners.

3 RELATED WORK

There are methods choosing (I) potential performance changes and (II) potential functional regressions.

Methods choosing potential performance changes (I) currently work based on heuristics. Alcocer et al. [6] identify potential changes by assigning costs to source code changes. They identify 87 % of all performance regressions by benchmarking only 14 % of all versions. Mostafa et al. [4] prioritise performance tests by estimating the impact of a code change on the performance. While those methods speed up performance benchmarking, they do not find all performance changes. There exists work, e.g., [3] [9] which focuses on selection of test input variables, e.g., user count, in order to determine the performance model, e.g. the function of the response time. This works focus on benchmarks where the same test case is executed with different parameters. PRONTO on the other hand finds all potential performance changes when testcases are executed with constant configuration but introduces the cost of a one time instrumented execution.

Yoo et al. [10] give an overview about functional regression test selection and prioritization (II). While selection defines exactly which tests to run, prioritization defines an order of tests which have higher priority, e.g. in order to detect similar bugs again. Infinitest[6] and EKSTAZI [2] are tools for regression test selection. Infinitest determines which tests to execute when the developer changes code in the IDE. Base of the change detection is the change-timestamp of the .class files. Infinitest only takes into account class relations produced by package membership and import statements. Therefore, more tests than necessary are executed, e.g. when class A and B are in the same package, a change to class B would imply a re-execution of class A, even if there is no call from A to B. Since Infinitest executes tests in parallel to the development process, fast and non-resource-heavy unit tests are needed. The tool EKSTAZI [2] saves dependencies of all test classes with the checksums of the called class files and updates them in every new version. The update is done by instrumenting the classes in order to determine which classes are really called. Therefore, EKSTAZI avoids false positives if files are imported and not called. Changes to a method that is not called are counted as change. Because EKSTAZI compares the checksum of the bytecode, it is not possible to handle changes of non-called methods differently.

4 EVALUATION

In the following, we will evaluate the selection rate of PRONTO against those of Infinitest and EKSTAZI. Therefore, the unit tests are transformed to performance tests and the selected tests of PRONTO, Infinitest and EKSTAZI are determined. Due to the current implementation, only projects using single module Maven in a part of its versions were analysed from the first version where Maven 3 with OpenJDK 8 runs until they stop using Maven or move to a multi

[6] https://infinitest.github.io/

Project	Versions	Tests	Ekstazi	%	Infinitest	%	PRONTO	%
commons-compress	1 968	595 747	58 433	9.81%	0*	0%	7 504	1.24%
commons-csv	937	112 660	22 547	20 01%	12.757	11.32%	14 049	9.16%
commons-dbcp	119	13 927	1 527	10.96%	0*	0%	2 330	0.97%
commons-fileupload	656	28 474	2 844+	9.99%	91+	0.32%	309+	2.59%
commons-io	1 193	205 171	40 534	19.76%	4 899	2.39%	22 751	2.80%
commons-numbers	157	48 365	7 403+	15.31%	3 120+	6.45%	22+	0.04%
commons-text	325	142 487	9 061	6.36%	95 920	67.32%	230	0.22%
Average		1 146 831	142 349	12.41 %	116 787	10.18 %	47 195	4.11 %

Table 1: Change Candidates for Open Source Projects

module project. Table 1 shows the results. On average, 12,41 % of the tests have to be executed based on EKSTAZI and 10.18 % based on Infinitest. Based on TSC, 4.11 % of the tests have to be executed.

In order to evaluate the efficiency of our performance change candidate selection, a process for determining all tests in all versions that the regression test selection tool would run was implemented[7]. Therefore, the same process is executed for every version in a Git repository. In the end, the executed tests are saved into an evaluation file, which makes it possible to compare it to PRONTO. In order to automate EKSTAZI, its official Maven plugin is used and added to the plugins section of a pom. Afterwards, the Maven tests are executed and the output of the Maven process is parsed in order to determine the tests which have been run. In order to automate Infinitest, its methods for determining a change are used. Since they depend on the change date of its .class-files, the Maven compiler plugin is used in order to build only incrementally. Since the build of commons-compress and commons-dbcp assume the call of different lifecycle parts, their call could not be automated by Infinitest (*). Since commons-fileupload and commons-numbers move to multi-module builds, not all versions were analysed (+).

In order to test correctness of the selection, a process for determining all non-selected tests was implemented. The transformed performance tests of the first 100 runnable versions of Apache Commons IO were executed and no change was detected in the 37 146 non selected tests, but 7 in the 507 selected tests. We therefore assume that PRONTO is correctly selecting tests in relevant use cases. There were no additional executions found by PRONTO, therefore Infinitest and EKSTAZI are sufficient to determine the candidates for performance changes, but they produce more candidates.

Regression test selection for performance changes and for functional changes differs. For detecting a functional change, one run is sufficient, so the test selection has to be faster than one execution. To detect a performance change, the test case needs to be executed many times. Therefore, one run with instrumentation is a reasonable effort in order to identify the tests that have to be called. This makes more fine-grained performance regression test selection possible.

5 SUMMARY AND FUTURE WORK

This paper presented the novel method PRONTO for selection of potentially changed performance tests. It consists of two parts: statically selected changes, where a set of called methods of a test are maintained and tests are selected based on static code analysis, and trace selected changes, where traces with source codes are compared. We evaluated this approach against the functional regression test tools Infinitest and EKSTAZI and showed that we are able to select fewer tests which need to be re-run.

The current implementation will be extended to be usable with Maven multi-module projects and Gradle. This approach could speed up all kinds of performance benchmarks or tests consisting of equal executions, e.g. it could be extended to be usable with benchmarking tools like jmh, with load test tools like JMeter, for distributed applications by instrumenting all of them and for usage in continuous integration servers.

[7] The code is available in the evaluation-module in the PeASS-repo.

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] Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically Rigorous Java Performance Evaluation. *ACM SIGPLAN Notices* 42, 10 (2007), 57–76.
[2] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Ekstazi: Lightweight Test Selection. In *Proceedings of the 37th International Conference on Software Engineering*. IEEE Press, 713–716.
[3] Raoufehsadat Hashemian, Niklas Carlsson, Diwakar Krishnamurthy, and Martin Arlitt. 2017. IRIS: Iterative and Intelligent Experiment Selection. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*. ACM, 143–154.
[4] Shaikh Mostafa, Xiaoyin Wang, and Tao Xie. 2017. PerfRanker: Prioritization of Performance Regression Tests for Collection-Intensive Software. In *Proceedings of the 26th ACM SIGSOFT ISSTA*. ACM, 23–34.
[5] Steven P Reiss and Manos Renieris. 2001. Encoding Program Executions. In *Proceedings of the 23rd ICSE*. IEEE Computer Society, 221–230.
[6] Juan Pablo Sandoval Alcocer, Alexandre Bergel, and Marco Tulio Valente. 2016. Learning from Source Code History to Identify Performance Failures. In *Proceedings of ACM/SPEC on International Conference on Performance Engineering*. ACM, 37–48.
[7] Petr Stefan, Vojtech Horky, Lubomir Bulej, and Petr Tuma. 2017. Unit Testing Performance in Java Projects: Are We There Yet?. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*. ACM, 401–412.
[8] André Van Hoorn, Jan Waller, and Wilhelm Hasselbring. 2012. Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis. In *Proceedings of the 3rd ACM/SPEC ICPE*. ACM, 247–248.
[9] Dennis Westermann, Rouven Krebs, and Jens Happe. 2011. Efficient Experiment Selection in Automated Software Performance Evaluations. In *European Performance Engineering Workshop*. Springer, 325–339.
[10] S. Yoo and M. Harman. 2012. Regression Testing Minimization, Selection and Prioritization: a Survey. *Software Testing, Verification and Reliability* 22, 2 (2012), 67–120. https://doi.org/10.1002/stvr.430