Automated Root Cause Isolation of Performance Regressions during Software Development

Christoph Heger
Karlsruhe Institute of Technology
Am Fasanengarten 5
76131 Karlsruhe, Germany
christoph.heger@kit.edu

Jens Happe
SAP Research
Vincenz-Priessnitz-Strasse 1
76131 Karlsruhe, Germany
jens.happe@sap.com

Roozbeh Farahbod
SAP Research
Vincenz-Priessnitz-Strasse 1
76131 Karlsruhe, Germany
roozbeh.farahbod@sap.com

ABSTRACT

Performance is crucial for the success of an application. To build responsive and cost efficient applications, software engineers must be able to detect and fix performance problems early in the development process. Existing approaches are either relying on a high level of abstraction such that critical problems cannot be detected or require high manual effort. In this paper, we present a novel approach that integrates performance regression root cause analysis into the existing development infrastructure using performance-aware unit tests and the revision history. Our approach is easy to use and provides software engineers immediate insights with automated root cause analysis. In a realistic case study based on the change history of Apache Commons Math, we demonstrate that our approach can automatically detect and identify the root cause of a major performance regression.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]; D.2.5 [**Software Engineering**]: Testing and Debugging; D.2.8 [**Software Engineering**]: Metrics—*performance measures*

General Terms

Measurement, Performance

Keywords

Performance Regression, Root Cause Isolation

1. INTRODUCTION

Despite a growing awareness for the importance of software quality, performance evaluation is still postponed to late development stages (the "fix-it-later" approach [32]). Performance evaluation is still a manual and time-consuming task. Software engineers need broad knowledge and expertise about the various tools and techniques to analyse the performance of their application. The conducted case studies of [35, 34] show that comparing to functional bugs, performance bugs are more challenging and require experienced software engineers and more code changes to fix.

Due to the high efforts required and the missing experience, software engineers defer performance tests as far as possible. But if a performance problem is found after deployment (usually by customers) it would be too late to fix it efficiently. Zimran and Butchart already showed that the integration of performance engineering into software development improves performance when applied early [36]. In contrast, the costs for fixing problems grow heavily the later they are discovered [10, 11].

Today in software development, many teams contribute to the code base of an application. Unit tests are written to validate the functionality of development artifacts. Revision control systems are employed to manage code artifacts and merge changes in order to enable the development teams to simultaneously contribute to the code base. Automated build infrastructure immediately notifies software engineers when a build is broken after changes have been applied. Performance regression testing is also an established discipline with many facets. Standard performance regression tests are specific for an early defined scenario. Software engineers create tests for the scenario which are then executed continuously. On the unit test level, software engineers are provided with tools like ContiPerf [2] or JUnitPerf [8]. Both tools build on existing JUnit [7] tests to evaluate the performance whenever a build is performed. However, both tools lack support for root cause analysis. In consequence, software engineers still have to identify the responsible revision which means, depending on the revision history, one or more revisions have to be tested for the occurrence of the performance regression. Thereafter, software engineers often create and run problem specific performance tests and manually investigate the functionality in the identified revision in order to find the root cause.

Software Performance Engineering (SPE) provides guidelines and approaches for evaluating software performance throughout the development process. Approaches for model-driven performance analysis allow software architects to create architectures that can satisfy their performance requirements. Since many performance problems are caused by low-level implementation details, they cannot be detected in high-level architecture models which miss the necessary details. Continuous performance tests during development are needed to identify problems that are caused by low level

ICPE'13, April 21–24, 2013, Prague, Czech Republic.
Copyright 2013 ACM 978-1-4503-1636-1/13/04 ...$15.00.

details. Load tests validate the performance of the system as a whole. However, they are usually executed late (e.g. just before a product is shipped) and provide no support for tracking the performance throughout the development process. Specific regression benchmarks [13, 16, 15] built for a specific problem, are proven to be useful to identify performance problems. While this approach provides important insights, it cannot be easily applied to a broader range of products. Automatic detection of performance problems and root cause analysis in load tests using statistical process control charts for performance problem detection and root cause analysis have been presented in [25] but require further research. Nevertheless, the authors show that in most cases automation reduces the root cause analysis time by about 90 percent [25]. Identification of performance bugs (not only performance regression problems) by monitoring deployed applications and providing the information to software engineers helps in finding the root causes [23]. However, load testing cannot be applied during development when no running application is available. None of these approaches provide continuous development-time root cause analysis by making use of unit tests and the revision history of the system under development.

In this paper, we propose a novel approach called PRCA (Performance regression Root Cause Analysis) for automated root cause analysis of performance regressions. PRCA uses unit tests to continuously monitor the application during the development phase, detects performance regressions, and identifies the revision in the history in which the performance regression was introduced. Thereafter, it isolates the root cause with systematic performance measurements based on dynamically extracted call tree information of unit tests. Our approach aims at ease of use and immediate insight for software engineers. We build on existing and commonly used technologies, such as JUnit [7] for unit testing and Git [4] or Subversion [9] as revision control systems known to software engineers. The performance tests can be easily integrated in the development infrastructure and their execution requires no further effort from software engineers. This approach allows us to automatically isolate the root cause of performance regressions in many cases.

We apply our approach to the revision history of Apache Commons Math [1], an open source math library, with the goal of demonstrating that PRCA can isolate the root cause of a known performance regression that took more than 14 month to be resolved. As a result, PRCA correctly identified the root cause of the performance regression.

Overall, we make the following contributions:

- We introduce an algorithm called PRCA to identify the root cause of a performance regression. The algorithm combines performance-aware unit tests with the revision history of the revision control system to isolate the problem's root cause as far as possible.
- We demonstrate that PRCA can isolate the root cause of a performance regression successfully.

The paper is structured as follows: In Section 2, we give an overview of our approach. In Section 3 we discuss our employed performance regression detection approach. In Section 4 we present our performance regression root cause analysis approach including bisection, call tree analysis, change correlation and software engineer feedback. We demonstrate that our approach identifies the root cause of a major performance regression in Section 5. We discuss our approach in Section 6 with potential future work, and present the related work in Section 7. Section 8 concludes the paper.

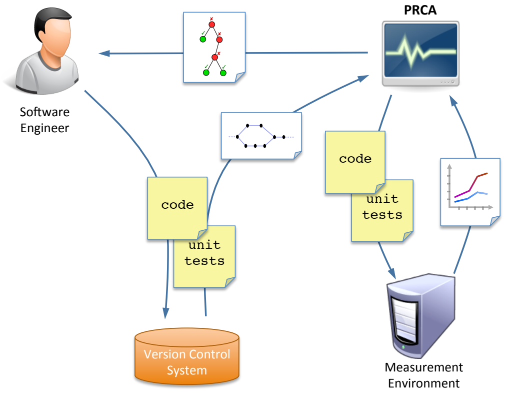


Figure 1: Overview of the PRCA approach

2. OVERVIEW

The main goal of our approach is twofold: to automatically identify performance problems as early as possible and to support software engineers with root cause analysis of the problems in order to facilitate a swift and efficient problem resolution.

To achieve this goal, PRCA combines the concepts of regression testing, bisection and call tree analysis to detect and provide performance regression root cause analysis as early as possible. The idea is to have the approach tightly integrated into the existing development infrastructure for continuous integration and regression testing. The continuity of performance tests allows identifying problems early. The close integration with the development infrastructure simplifies the usage of performance detection method.

The approach assumes the existence of two essential elements in the development environment: (a) unit tests as important artifacts of the target system that indirectly identify the main functionalities of the system, and (b) a code repository that offers change graphs of the artifacts of the system. Hence, the approach is applicable to any Java-based software system that has a proper set of unit tests and its source code is maintained on a revision control system.

Figures 1 and 2 provide an overview of PRCA. At regular intervals, PRCA fetches the latest revision of the software system from the repository, instruments the tests in unit test classes, runs all tests (with repetition) and gathers measurements on performance metrics of interest (step 1). For every test in unit test classes, it then compares the measurement results with the previous measurements done at the last performance test (step 2). If a performance regression is detected for any test, PRCA retrieves the change history graph from the revision control system and, performing a bisect algorithm, identifies one of the (potentially many) changes that introduced the performance regression (step 3). In the next step, the methods called by the affected test are instrumented, the test is executed and a call tree annotated with performance data is produced. The annotated call tree is then analyzed to identify the methods that con-

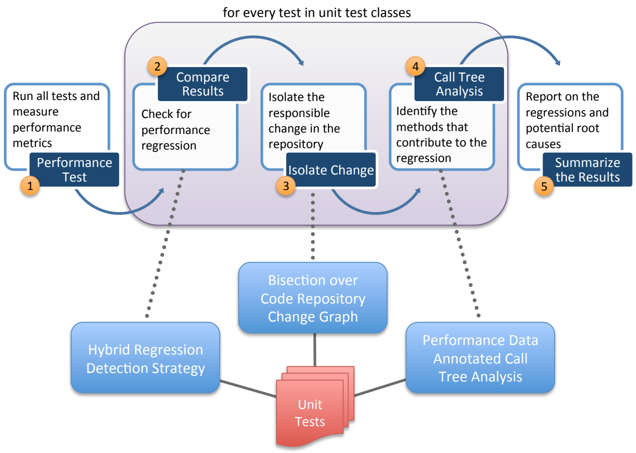

for every test in unit test classes
Run all tests and measure performance metrics
Performance Test
1
2
Compare Results
Check for performance regression
Isolate the responsible change in the repository
3
Isolate Change
4
Call Tree Analysis
Identify the methods that contribute to the regression
Report on the regressions and potential root causes
5
Summarize the Results
Bisection over Code Repository Change Graph
Hybrid Regression Detection Strategy
Performance Data Annotated Call Tree Analysis
Unit Tests

Figure 2: The PRCA process

tribute to the performance regression (step 4). The result of the analysis is then reported to the software engineer (step 5).

In order to achieve the best results, we assume that for the components that have performance requirements, the unit tests accompanying the source code are *performance-aware unit tests*. We define performance-aware unit tests as unit tests that are developed with performance requirements in mind. For example, a unit test for a component that computes the critical path of a graph, may test the component with graph sizes of 1, 5, and 10, focusing only on the graph structure. If the component is expected to be used, in the target application, with graphs of size 50 to 100 nodes, a performance-aware unit test will either test the component for larger graph sizes (such as 30, 60, and 120), or will have an explicit test for at least one large graph size (e.g., size 50) in order to test the component under a “typical” load.

For regression detection (step 2 of Figure 2), we need to properly define what is considered as a performance regression. In Section 3, we briefly discuss various methods of regression detection considered for this approach.

In order to isolate the change in the repository (step 3), we need an efficient strategy to identify the change that introduced the regression. Running performance tests is a time and resource consuming task. Each performance test may require a warm up time and many repetitions in order to achieve stable and reproducible results. As a consequence, performance tests cannot be executed with every build. For the scope of this paper, we assume that performance tests are run in longer but regular intervals (e.g., weekly). If a regression is detected, there may be a long history of changes made to the application. In this step, we apply a variation of the Git bisection algorithm [3] to identify the change(s) that introduced the regression. We discuss this topic in Section 4.

For the call tree analysis (step 4), our approach automatically instruments the methods that are called by the test (and that are part of the application or component interface) and systematically searches for increases in response times (Section 4). It then performs a breath first search that investigates the methods with the largest increase in response time. The measurements are done on both revisions the new and the changed one. If the implementation of a method has been changed compared to the original revision, we consider this method as the root cause of the regression problem.

3. REGRESSION DETECTION

In this work, we consider a performance regression as a significant increase of the response time of a tested method. In detecting regressions based on measurements, we are facing the challenge to report regressions that are relevant in the application context and valuable for the software engineers in improving the performance of the software. Therefore, we have to avoid reporting irrelevant or insignificant regressions (false positives) or miss a critical regression (false negatives). The balance here is to apply a meaningful regression strategy that deals with noise and also distinguishes between minor and major regressions. Existing approaches employ either some kind of a threshold [18, 22, 25] (derived or specified) or historic data [13, 28] each can be considered as a baseline acting as comparison counterpart. Lee et al. [25] state that the main issues in regression detection include the determination of the baseline of performance of the system under test and the deviation from the baseline which is considered as performance regression. PRCA includes no contribution in the field of performance regression testing. We consider the regression detection part as an extension point for future research. Our approach can be enhanced with automated generation of parameterized unit tests [19], test input selection which maximizes resource consumption in order to find performance problems [21] and Stochastic Performance Logic formulas [12] to enable software engineers expressing performance requirements. This could reduce the burden of providing performance-aware unit tests for software engineers.

In PRCA, we currently employ statistically rigorous performance evaluation after the work of Georges et al. [20] for performance regression detection. Since errors in experimental measurements affect the accuracy of the results, we employ a stable measurement environment to avoid that systematic errors introduce a bias into the measurements. Our stable measurement environment is a hardware platform and Java Virtual Machine (JVM) implementation that remains unchanged and is dedicated to performance testing. We run a unit test multiple times obtaining stable measurements in the measurement environment for a certain revision. We do not exclude JIT (re)compilation overhead from our performance measurements because we observed that the first run does the compilation and the performance of subsequent runs suffer less from variability due to JIT (re)compilation. We use the Analysis of Variance (ANOVA) to compare the response time distributions of two revisions. Our decision to use ANOVA is based on the suggestion of Georges et al., who propose ANOVA as a statically rigorous method to compare alternatives wrt. performance: “Also, ANOVA assumes that the errors in the measurements for the different alternatives are independent and Gaussian distributed. ANOVA is fairly robust towards non-normality, especially in case there is a balanced number of measurements for each of the alternatives.” [20] After applying an ANOVA test PRCA can conclude if there is a statistically significant difference between the two samples that is a performance regression. Based on a series of experiments, we choose a confidence level of 99% as suitable to deal with false positives and false negatives in most cases. The termination criteria of the unit test run repetition can either be a fixed number of repetitions or a

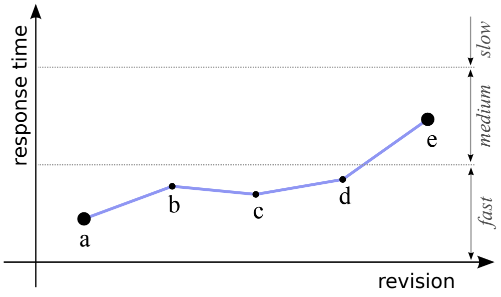


Figure 3: Regression detection strategy example. Category separation is indicated by the horizontal dashed line.

certain confidence level. While confidence levels are more appropriate, it is unclear how many repetitions are actually necessary. In our case studies, we repeated each test run 50 times in order to achieve stable results based on our observation that 50 repetitions with a confidence level of 0.95 can result in a relative measurement error of less than 10% for larger response times (>100 ms). However, this setting is very case specific and needs to be considered carefully.

Furthermore, not every performance degradation is a major performance regression that must be corrected. To support software engineers in understanding the importance of a regression and minimize false positives, we employ a regression detection strategy that classifies performance degradation into minor and major regressions. The classification uses categories (thresholds) as baseline for major performance regression detection and previous measurements as baseline for minor performance regression detection. Categories span a range defining an interval for response time; for example, performance categories "fast", "medium" and "slow" can be defined (cf. Figure 3). Methods are automatically classified based on the measured response time during the first test execution. PRCA reports a warning if it is not the fastest category from a performance perspective. Software engineers can change the assignments as needed. The defined categories act as baselines indicated by the dashed horizontal line in Figure 3. With evolving revisions our approach then reports a major performance regression if the requirements of the assigned category are not satisfied. For example, the changes applied in revision e in Figure 3 change the category of the tested method from "fast" to "medium" leading to a major regression. In contrast, if the category's requirements are satisfied but historic data comparison shows a significant performance regression (cf. revisions a and b in Figure 3), PRCA reports a minor performance regression. The strategy enables software engineers to easily distinguish between minor and major performance regressions and to select only crucial performance regressions for further problem diagnosis. The advantage is that a manually specified threshold is not needed and the risk of detecting a performance regression late is reduced. Furthermore, it overcomes the lack of knowledge software engineers may have particularly with regard to the acceptable response time.

4. ROOT CAUSE ANALYSIS

Our problem diagnosis approach is partitioned into two steps: bisection and call tree analysis. Within the bisection step, we systematically evaluate the performance of revisions in the repository in order to identify the change that leads to the performance regression. The call tree analysis step is then applied to find the root cause of the problem. Therefore, we extract a performance annotated call tree (serving as a performance behaviour model) out of the tested method and apply our analysis to systematically investigate the tested methods call tree.

4.1 Bisection

Detecting the presence of a performance regression alone within a performance test period is not sufficient, especially when there are many revisions in the given period. When a performance regression is detected, it would be desirable to know which change introduced the regression. Hence, identifying the responsible change of a performance regression is the first step of our proposed problem diagnosis approach. It supports and limits the problem diagnosis effort by enabling the software engineer(s) to focus on the responsible change rather than the whole history of the changes within the test period.

In this section, we describe our approach of identifying the responsible change in the revision control system that lead to a performance regression. The approach is based on the Git bisect functionality [3]. This feature of Git offers a binary search over the change history and enables software engineers to identify the change that introduced a problem. The bisection process can also be automated by configuring Git bisect to launch a script or command at each bisection step checking if the current change is already affected by the regression.

We adapt the bisection algorithm [3] of Torvalds and Hamano to find the best intersection point in each step. Change histories in general can be described as directed acyclic graphs (DAG). When a performance regression is detected between a pair of changes (c_s, c_e), where $time(c_s) < time(c_e)$, the actions performed by the algorithm can be summarized as follows:[1]

1. Filter the DAG for relevant revisions and keep only the nodes that are ancestors of the change c_e, excluding c_s and ancestors of c_s.

2. Compute and associate the number of ancestors a_i to each node c_i starting with the leaves of the DAG. This is the first step to compute a weight for each node in the DAG.

3. Compute the corresponding weight

$$w_i = min(a_i + 1, N - (a_i + 1)) \tag{1}$$

   and associate it with node c_i. Here, N is the number of nodes in the filtered DAG.

4. The next intersection point is the change c_i with the highest weight w_i. We call this change c_c.

We then apply our regression detection strategy of Section 3 to examine the triple of changes (c_s, c_c, c_e). We first check the pair (c_s, c_c) to see if c_c includes a performance

[1] We assume that there are no skipped changes.

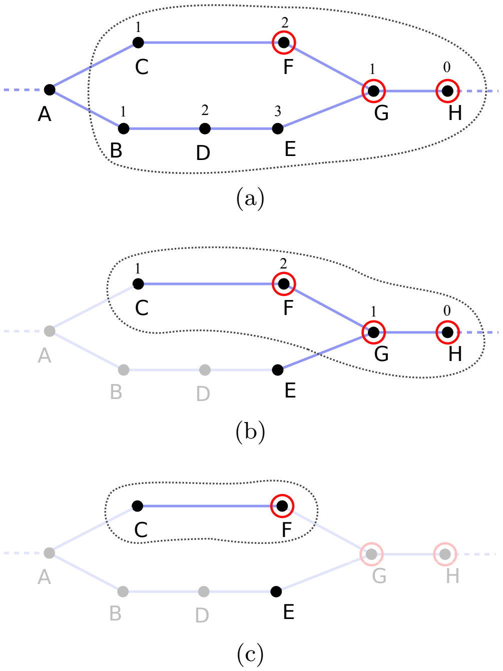


Figure 4: Bisection steps on a change graph

regression. To do so, we test the hypothesis that the mean of the response times μ_{c_s} is equal to that of μ_{c_c}. Therefore, we check if the null hypothesis

$$H_0 : \mu_{c_s} = \mu_{c_c} \tag{2}$$

cannot be rejected. If the response times do not differ significantly or if they differ but $\mu_{c_s} > \mu_{c_c}$, we assume that c_c is not affected by a performance regression and continue the bisection method with the pair (c_c, c_e). Otherwise, c_c includes a performance regression and we continue the bisection method with the pair (c_s, c_c). The method continues until for a given input pair of (c_s, c_e), we have only the nodes c_s and c_e in the graph. We then report c_e as the change responsible for the performance regression.

Figure 4 shows an example of applying bisection to a change graph for the pair of changes (A, H). At each step, the dotted grey line indicates the filtered graph. Assuming that a detectable performance regression is introduced in change F, the regression is propagated into changes G and H (marked with red circles in Figure 4(a)). Comparing changes H and A, our method detects a performance regression at H. At the first step of bisection, node E has the highest weight of $3 = min(3, 4)$ in the filtered graph, so E is selected as the next intersection point. Since E does not introduce a performance regression compared to A, bisection is repeated for the pair of (E, H). The graph is filtered (Figure 4(b)) and F (with a weight of 2) is selected as the next intersection point. Node F shows a performance regression compared to E, hence bisection continues for the pair (E, F) with only the nodes C and F remaining in the filtered graph (Figure 4(c)). Node C does not show a regression compared to E, so the bisection continues with the pair (C, F) and since there would be no more node in the graph except C and F, node F is reported as the node responsible for the introduced regression.

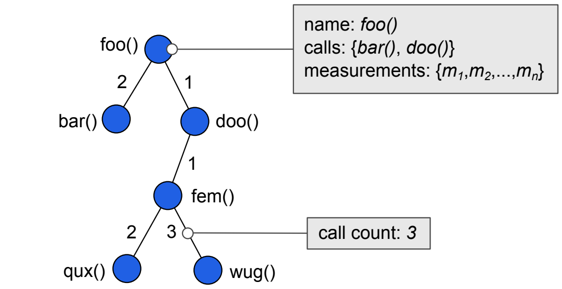


Figure 5: Call tree with performance annotations

Finding the next intersection point is the first step of the bisection process within our approach. The second step is to examine the change by running performance regression tests using our regression detection strategy described in Section 3. In each bisection step, the source code of the change is checked out and an automatic build is executed. In the resulting executable any tested method is instrumented for runtime monitoring. The corresponding unit tests are then executed to gather response time measurements.

4.2 Call Tree Analysis

In this section, we present PRCA's strategy for root cause analysis using call trees. PRCA identifies methods in the call tree and reports paths that are suspected to have caused the performance regressions to software engineers. For this purpose, PRCA examines the call trees of tested methods for which performance regressions have been detected. Furthermore, we explain PRCA's method for extracting call trees annotated with performance data gathered through systematic measurements.

For root cause analysis, we employ a call tree with performance annotations (cf. Figure 5). Methods are represented as nodes (e.g., *foo()*) in the call tree. Each node contains the signature of the method, a list of method calls and a series of response time measurements. Edges connecting two nodes represent method calls (e.g. *foo()* calls *bar()*). The edges are annotated with the number of calls (*call count*) performed during the execution of the corresponding tests. For example in Figure 5, *fem()* calls *wug()* 3 times.

PRCA extracts annotated call trees in two steps. In the first step, we use a full instrumentation of the test case to retrieve the structure of a call tree and the number of method calls. In the second step, we extract performance data of individual methods. This step requires many test executions and a very selective instrumentation in order to produce reliable performance data. In the following, we describe both steps in more detail.

To extract the structure of the call tree (i.e., its nodes and edges), we instrument all methods called during test execution. The instrumentation is fully automated and performed by PRCA requiring no manual intervention. For this purpose, PRCA employs byte code weaving based on Javassist [6, 14] and the monitoring possibilities of Kieker [31]. Java classes are loaded into a class pool, a copy of the byte code is stored in a byte code repository and the monitoring instructions are weaved into the byte code of the method(s) before they are used. The original class, if already loaded into the JVM, is replaced with the instrumented class using the HotSwap mechanism for class file replacement of the

Java SDK. The byte code stored in the repository is later used to undo the instrumentation. Based on the generated trace data, PRCA can extract the structure of the call tree including the number of calls from one method to another. Because the call tree and the call count are not affected by monitoring overhead, all data can be collected in a single test execution.

However, the same approach is not suitable for gathering the performance data. Response times can be strongly affected by measurement overhead introduced by a full instrumentation. The instrumentation of one method can affect the response time of another introducing systematic errors which causes a bias in measurements affecting the accuracy of the results [20]. For example, when *doo()* and *fem()* (cf. Figure 5) are instrumented and monitored in the same test run, the time required to execute the monitoring instructions in *fem()* are part of the response time of *doo()*. If this is not considered, the overall regression detection can easily be misleading. In addition, performance measurements have to be reproducible and minimize potential disturbances.

To address these issues, PRCA breaks down the extraction of response times in multiple steps and repeats measurements several times. In each step, PRCA instruments only the methods on one level of the call tree. With this strategy, PRCA minimizes disturbances as none of the methods called by the instrumented method are monitored. For the example in Figure 5, PRCA first instruments *foo()*, second *bar()* and *doo()*, third *fem()*, and finally *qux()* and *wug()*. For each step, PRCA executes the test multiple times which results in measurements $m_1, \ldots, m_n$ for each method (cf. Figure 5).

The goal of the call tree analysis is to identify paths in the call tree with methods that are suspected to cause the performance regression. For this purpose, the call trees with performance annotations are extracted for the revisions where a regression has been introduced. Figure 6 continues the example introduced in Section 4.1. In this example, the changes made from revision C to F introduced a regression. To identify the methods that potentially cause the performance regression (cf. *fem()* in Figure 6), PRCA compares the nodes in the call tree of revision C with its counterpart in revision F. For each pair of nodes, PRCA conducts an ANOVA test for the response time measurements. Using a breadth-first search, our approach looks for nodes with statistically significant differences between the measurements of both revisions (e.g. *foo()*, *doo()* and *fem()*). Such nodes are flagged and all methods called by them are evaluated in the next step of the breadth-first search. If the ANOVA test cannot conclude that there are statistically significant differences in the response times of a method between the two revisions, our algorithm does not further investigate it. For example, this is the case for *bar()*, *qux()* and *wug()* in Figure 6. Furthermore, if the call tree of a method changed from one revision to the next and its response time increased significantly, our call tree comparison will flag the method as a candidate for the root cause of the regression. Since the call trees of both revisions are different, our algorithm does not further investigate it. After the breath-first search is done, the flagged nodes point to the potential root cause in the call tree (*foo()*→*doo()*→*fem()* in Figure 6). PRCA uses the path and correlates it with the changes in the revision control system.

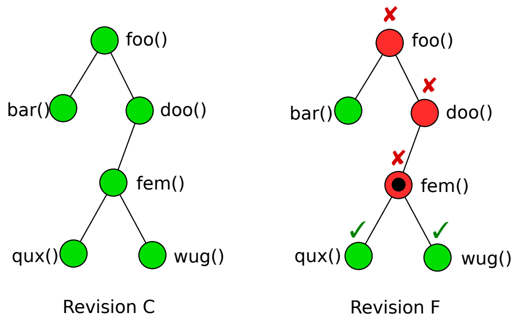


Figure 6: Example of call tree analysis. Methods that are suspected to cause a performance regression are marked with a cross.

4.3 Change Correlation

The goal of change correlation is to identify the methods in a path that are affected by changes that are suspected to cause the performance regression. The challenge here is to track down the changes included in a revision to the methods establishing a path. The changes are provided by and can be queried from the revision control system (e.g. Git). Without employing static code analysis, we are only able to track down changes to class level. Therefore, we extract the class name out of any node in a path (e.g. in Java from the full qualified method name *<packagename>.<classname>.foo()*). We then look up the changes in the revision control system for the responsible revision to check if there are any changes made to that class. Methods that might be affected by changes are marked (cf. *fem()* in Figure 6). This enables us to exclude methods of classes that are unchanged from the feedback we give to the software engineer.

4.4 Software Engineer Feedback

The goal of our approach is to support software engineers in root cause analysis of performance regressions. The challenge of providing feedback on root cause analysis results is the amount and complexity of the presented information. Our goal is to provide self-explanatory feedback (cf. Figure 7) to software engineers that shows the starting point for problem resolution without requiring deep knowledge (e.g. in statistics). Following a lean approach, we hide the complexity as much as possible. For a detected performance regression, we show the name of the unit test and the test method as well as the name of the tested method. We employ two figures to communicate the results of the root cause analysis. We highlight the performance regression between the two revisions (e.g. C and F) as well as the categories. The results of the call tree analysis are shown in a figure that contains the call tree of the identified revision. The call tree is similar to the one for revision F in Figure 6. Methods that are suspected to cause the regression are highlighted. Nodes that are correlated with changes are flagged with a black dot (cf. *fem()* in Figure 7). The feedback includes all necessary information to provide software engineers with a well prepared starting point for problem resolution.

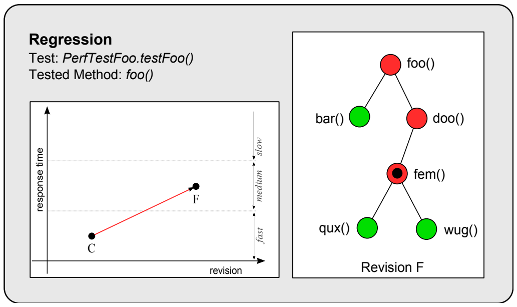


Figure 7: Feedback given to software engineers.

5. CASE STUDY

In this section we look into a number of case studies that validate the approach presented in this paper. We begin with four synthetic experiments that cover four cases of a performance regression (see Figure 9) and validate the regression detection approach for each case. We then provide a summary of applying our method to a real-world scenario, the Apache Commons Mathematics Library (Commons Math) [1]. There, we utilized our call tree analysis method to find the root cause of a performance regression that was introduced in a certain revision of Apache Commons Math.

5.1 Controlled Experiments

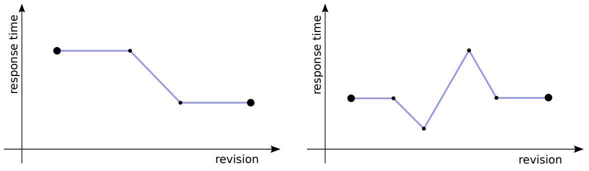
In order to demonstrate the validity of the regression detection method, we applied our method to four well-defined controlled experiments with artificial performance regressions. With these experiments, we address the overarching question:

Can PRCA identify the injected regressions as expected?

We defined each experiment around six consecutive revisions of a test system, namely revisions $[r_1 \dots r_6]$, where revision r_1 represents the last point in the revision history that performance regression testing was applied and revision r_6 is the latest revision in the repository when the current performance regression test is being performed. Since the goal is not to validate the already established bisection algorithm, we assume a linear order of the changes in each experiment. The system under test is an implementation of the Fibonacci function, which allows us to simply focus on and introduce changes in the response time. We assume that the configuration of the system under test provides the input value to the Fibonacci function, which can vary between revisions. The system comes with a unit test that reads the input value from the configuration file and runs the Fibonacci function for that particular value.

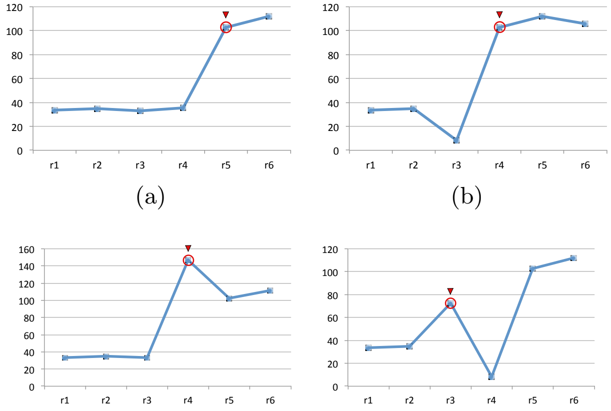
At every performance test, we are facing one of the following three cases: (i) performance is improved since the last tested revision (Figure 8(a)), (ii) performance is almost the same as the last tested revision with no significant change (Figure 8(b)), even though performance oscillation may have occurred between the two revisions which will not be detected, (iii) performance is worsened between the current revision and the last tested revision (examples in Figure 9). For these experiments, we focus only on the latter case where we have a significant performance regression between the two tested revisions.

In order to simulate the performance change in a revision, we change the system configuration and increase or decrease the input value to the Fibonacci function. The input value is changed such that it results to a significant performance change of the system. In the following sections, we describe each experiment in detail.



(a) (b)

Figure 8: (a) Performance improved; (b) No performance change detected



(a) (b) (c) (d)

Figure 9: Evaluation of the Bisection Method

Simple Regression

The purpose of this experiment is to validate detection of a simple performance regression introduced at some point between revisions 1 and 6 (see Figure 9(a)). We simulate the performance regression by increasing the input value in the configuration at the fifth revision (r_5). The null hypothesis $H_0 : \mu_{r_1} = \mu_{r_6}$ is rejected during the performance regression test and the bisection starts. In the first bisection step $H_0 : \mu_{r_1} = \mu_{r_3}$ is not rejected. The bisection continues until the test $H_0 : \mu_{r_4} = \mu_{r_5}$ is rejected as a result of performance regression. Hence, the bisection identifies r_5 as the responsible change (indicated by the arrow in Figure 9(a)).

Improvement and Regression

Next, we validate detection of a performance regression introduced after a performance improvement (see Figure 9(b)). We simulate a leading performance improvement in front of a performance regression. Therefore, we decrease the value of the input parameter at r_3 before we use fault injection to increase it in r_4. The performance regression detection rejects $H_0 : \mu_{r_1} = \mu_{r_6}$ and starts bisection. In the following bisection steps $H_0 : \mu_{r_1} = \mu_{r_3}$ is rejected due to a performance improvement, and $H_0 : \mu_{r_3} = \mu_{r_6}$ is rejected as a result of a performance regression, so bisection continues with r_3 and r_6. In the last bisection step, $H_0 : \mu_{r_3} = \mu_{r_4}$ is rejected due to performance regression and as a result, the bisection identifies r_4 as the source of performance regression.

The reverse of the story is also presented in Figure 9(c), where a performance improvement (r_5) follows a performance regression (r_4). The bisection method correctly detects r_4 as the source of regression.

Performance Oscillation

In the last experiment, we alternate performance regressions and improvements to simulate an oscillating performance (see Figure 9(d)). We use fault injection in revisions r_3 and r_5 to decrease the performance. We reverse the fault injection in r_4 to temporarily improve the performance. As a result, both revisions r_3 and r_5 are introducing performance regression. Through the bisection process $H_0 : \mu_{r_1} = \mu_{r_3}$ is rejected and the bisection identifies r_3 as responsible change and the bisection ends. We will further discuss, in Section 6, possible improvements to our method with respect to performance oscillation.

5.2 Apache Commons Math

Apache Commons Math (henceforth called *commons math*) is a lightweight mathematics and statistics library for Java that addresses the common mathematical problems for which a suitable solution is not available in the Java programming language [1]. It is a small set of utilities that address programming problems such as fitting a line to a set of data points, curve fitting, solving system of linear equations, random number generation, and other miscellaneous mathematical functions. With a focus on real-world applications, it is apparent that performance of the commons math library in computing mathematical functions is of high importance. Various sources maintain and publish performance comparison of commons math against other available libraries (see [5] for example).

On October 10th, 2010, a code modification submitted to the commons math code repository (revision 1006301) introduced a substantial performance regression in one of the functions. The problem stayed hidden for 7 months until it was first reported[2] on May 16th, 2011. The resolution of the problem took more than 14 months until it was finally resolved on July 22nd, 2012.

There are two interesting aspects in this story: (a) the performance regression introduced by the submitted code stayed unreported for more than half a year; (b) it took more than one additional year for the bug to be resolved. Our proposed approach in this paper targets both of these aspects. Continuous performance monitoring based on performance-aware unit tests reveals performance issues early in the development process, before allowing their causes to get lost in the history of changes or their effects be propagated further into other components. When a performance regression is detected, our proposed call tree analysis method investigates and reports the root cause of the problem, hence considerably facilitating the problem solving process.

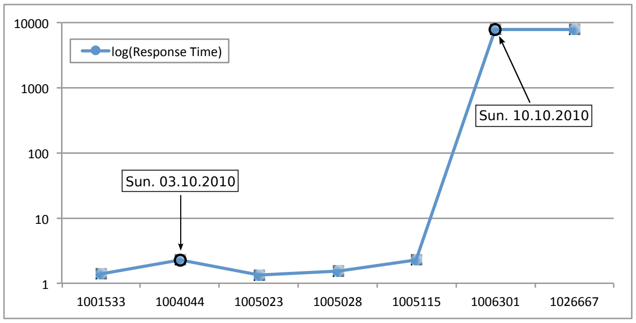


Figure 10: Performance Regression in Apache Commons Math (2010)

Assuming that performance-aware unit tests are available,[3] we applied our approach to the commons math code repository. We supposed a weekly performance regression routine performed every Monday morning at 00:00 that runs all unit tests and monitors their performance. Applying this method retrospectively to the commons math code repository revisions in October 2010 we observe the performance graph of Figure 10 for our performance-aware variation of the `DescriptiveStatisticsTest` unit test that uses the troubling function. The execution of all unit tests[4] (including repetition) took about 48 minutes for revision 1006301 and 30 minutes for revision 1004044. Our bisection method correctly detects a performance regression and identifies revision 1006301 as the cause of this regression. In this case, this is the right-most revision in testing interval. The residual distribution plots of the two revisions marked in the diagram of Figure 10 are provided in Figure 11. Despite a few outliers, the residuals in both plots are close to the straight line indicating that random errors have a close to normal distribution.

In the next step, PRCA applies the call tree analysis method to find the root cause of the problem in revision 1006301. It instruments the methods that are called in the test that introduced regression and identifies a regression in

$$\texttt{DescriptiveStatistics.getPercentile}(\texttt{double})$$

in revision 1006301 compared to 100515. A breadth-first analysis of the call tree (comparing the performance of the methods between the two revisions) is continued until the root cause of the regression is isolated to the call tree of Figure 12. The problem appears to be originated from a change in the following method:

$$\texttt{Percentile.evaluate}(\texttt{double}[], \texttt{int}, \texttt{int}, \texttt{double})$$

The call tree analysis of PRCA detects that (a) the call tree under the above method is changed in the problematic revision, and (b) the common parts of that call tree do not show a performance regression. As a result, the call tree leading to that method is reported as a potential cause of the introduced performance regression. This analysis also matches what happened in reality. The revision 1006301 introduced a change in this method that caused the performance of the method to decrease seriously for certain boundary cases.

6. DISCUSSION

There is no need to argue that the proposed combination of automated detection and root cause analysis of perfor-

[2] `https://issues.apache.org/jira/browse/MATH-578`

[3] We manually extended the unit tests with one that tests the affected functionality of commons math.

[4] Unit tests in folder `src/test/java` (excluding 3 tests that we were not able to run properly with our tooling).

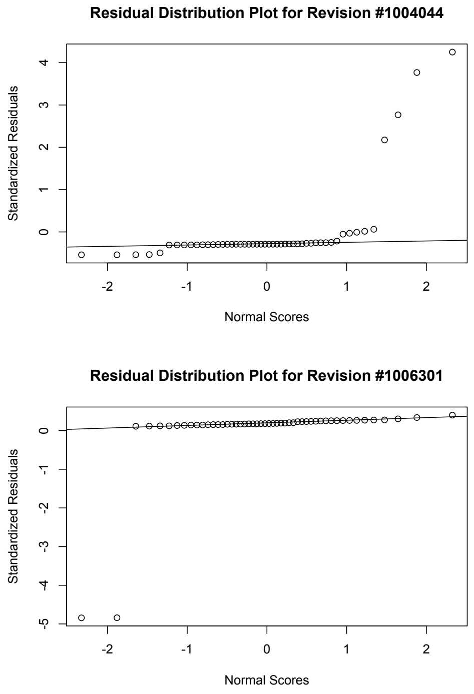


Figure 11: Residual Distribution Plots

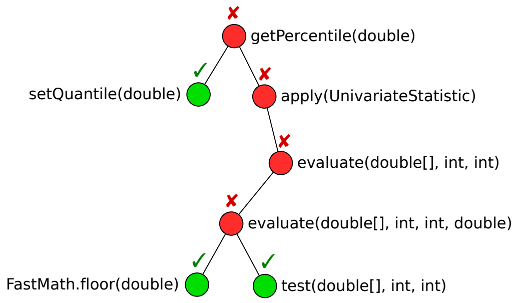


Figure 12: The Problematic Call Tree in Apache Commons Math Revision 1006301

mance regressions early in the software development process can significantly reduce the costs of fixing performance issues and maintaining the quality of the software. Automated regression detection identifies performance issues as soon as possible before they are hidden in the future changes to the software. When a regression is detected, the proposed root cause analysis then helps software engineers to focus on the potential sources of the problem.

We have already applied the idea of continuous performance regression testing and root cause analysis internally at SAP. Over the course of one month, we detected a crucial performance regression which would otherwise have stayed hidden. Despite the encouraging results of applying the method to internal projects and case studies, there are still limitations and issues that require improvement. In the following, we briefly point out these limitations and discuss potential solutions and future improvements.

Although one can apply the proposed method to any software under development that has a proper set of unit tests,

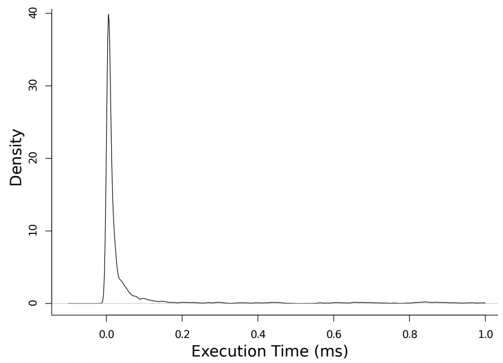


Figure 13: An example of unit test time distribution

regular unit tests are usually not enough to detect performance regressions. For example, Figure 13 shows the distribution of unit test execution times of an internal project at SAP. In this example, the majority of the tests are executed in less than 0.1 milliseconds. The execution times for these tests are too short to allow for a meaningful detection of regressions. In most cases such as this example, regular unit tests focus only on functional testing with usually the minimum set of test cases to test the functionality of the components. In order to detect performance regressions properly, our proposed method requires *performance-aware unit tests* that test the system units not only for functionality, but also for cases that can potentially be critical for the performance of the code in the target application context. This also implies that the performance unit tests have the right granularity and coverage to detect relevant performance regressions. Of course, considering that the goal of this approach is to mostly rely on the existing unit tests and *not* to build specifically designed unit tests that target all the performance-influencing parts of the code, there is no guarantee that all performance regressions will be detected.

The proper definition of *performance regression* and the development and evaluation of good heuristics for its detection are crucial for a broader applicability of performance regression testing. The heuristics for regression detection must minimize false positives (wrong alerts) and false negatives (regressions going undetected) in order to be useful. Software engineers quickly lose interest in performance regression tests if the results are not reliable or they have to provide too much additional information for the tests. In this work, we used a performance regression detection strategy based on historical data and categories. Although this approach worked well in our scenarios, it still requires further evaluation and refinements, such as conducting more experiments with performance regression issues and a thorough analysis of the results of fault positive and fault negative rates.

The current version of our framework has limitations in detecting and properly reporting three cases of performance regression:

1. Performance oscillations within a test period which lead to similar performance at both ends of the period are not detectable with our approach. If this is the result of a performance regression introduced and then resolved within the test period, it can be ignored.

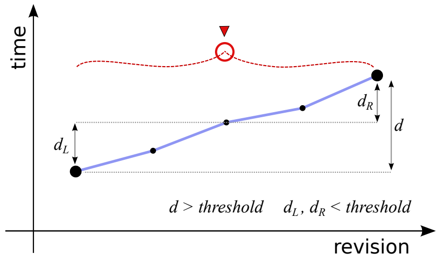


Figure 14: The case of slow performance decay

However, such an oscillation can be caused by a pair of unrelated performance improvement and regression. In such a case, it is desirable to be able to detect the regression introduced in the code. We can reduce the chance of such cases by having more frequent performance regression tests (hence shortening the test periods). However, unless there are enough resources to perform regression tests on every change submitted, we cannot detect the performance regressions introduced in such cases.

2. The second is the case of slow decay of performance within a test period. Using our regression detection strategy, we would ultimately detect such a decay, but the root cause analysis would be difficult. Take the example of Figure 14, in which the performance is decreasing slowly such that a performance regression is detected between the two points of the test interval, but the problem diagnosis method cannot identify a single revision as the cause of the regression since the mid point of the period does not introduce a significant change in performance. In such cases, we would like to report all the changes within the test period as potential sources of performance regression.

3. The third limitation occurs when there is more than one regression introduced within the test period. In this case, we only report one change as the cause of the overall performance regression. We would like to improve our bisection algorithm such that once a performance regression is detected for a testing period, the algorithm would recursively identify and report all the changes that introduce a performance regression within the test period.

We consider the issues and ideas for improvements discussed in this section as topics of future work.

7. RELATED WORK

The approach presented in this paper combines performance regression testing with automatic diagnosis of performance problems. In the following, we discuss existing work that addresses these aspects.

Only few approaches combine performance regression testing with problem diagnosis and root cause analysis. Lee et al. [25] present an approach for automatically detecting performance anomalies in database management systems and isolating their root cause. They developed a framework that significantly reduces the effort of detecting and isolating performance problems for database developers. The approach uses established database benchmarks and profiling methods to compare new commits to the most recent stable state. Lee et al. share our goal of detecting performance problems early and supporting software engineers in isolating the problem's root cause. However, their solution is highly specific to the domain of database management systems and cannot easily be generalized. Mostafa and Krintz [28] developed an approach to automatically identify differences in call trees between two subsequent code revisions. The call tree can also contain performance metrics and include them in the tree comparison. While performance is considered, the authors focus on tree comparison and try to identify changes in the call tree (like added, removed or modified methods). The most important challenges of performance regression testing are not addressed.

Performance regression testing itself has been subject to research for several years [13, 18, 29]. Nguyen et al. [29] use statistical process control charts to analyze performance counters across test runs to check for performance regressions. Bulej et al. [13] use specialized regression benchmarks to detect performance regressions in application middleware. Specific regression benchmarks provide important insights, but cannot be easily applied to a broader range of products. Foo et al. [18] present an automated approach for performance regression testing to uncover performance regressions in situations where software performance degrades compared to previous releases. Performance metrics of a software system are monitored and compared to a set of estimated correlated metrics extracted from previous performance regression testing results. None of the approaches supports the root cause isolation of performance regressions.

In addition, the performance of a software system depends on its input data [24]. Thus, it is important to choose the input data for test cases well. Existing approaches [19, 21] support software engineers in choosing good input data for functional and performance tests. Grechanik *et al.* [21] apply machine learning techniques to find the input set which maximizes resource consumption in order to increase the probability of finding performance problems. Fraser and Zeller [19] present an approach for automatic generation of parameterized functional unit tests. Such approaches complement our work. The combination of input data generation with our approach can increase the efficiency of performance problem detection and root cause analysis.

Existing approaches that focus on performance problem detection during runtime [17, 26, 27, 30, 33] are using monitoring techniques to track a software system's performance and to gather measurement data for performance analysis. Even though these approaches can detect performance problems in a productive system and sometimes diagnose their root causes, they can only be a last resort. Detecting performance problems during operation is way too late to solve them efficiently.

8. CONCLUSIONS

In this paper we presented PRCA, a novel approach that utilizes unit tests and revision history graphs for automatic detection and root cause analysis of performance problems throughout the development process. The approach presented here builds on (i) a hybrid regression detection strategy, (ii) bisection over revision change graphs, and (iii) analysis of performance annotated call trees, to provide a process of continuous performance regression root cause analysis. We evaluated different heuristics for regression detection in order to minimize false positives. We extended Git bisection algorithm to identify the changes that introduced a performance problem. Finally, we designed an approach to systematically analyze call trees to identify the methods and call graphs that most likely caused the performance regression.

Once our methods are integrated in the development infrastructure used at SAP, software engineers can use familiar tools and techniques to write performance-aware unit tests. The tests are automatically executed on a regular basis (e.g. during integration builds). If a regression is detected, PRCA automatically provides information on which change caused the regression and which methods are involved. Using performance regression testing, software engineers can detect performance problems early and fix them swiftly. We have validated and demonstrated the application of PRCA using control experiments and Apache Commons Math as a real-world example.

Based on the promising results of our case study, we plan to apply our approach inside SAP. To achieve this, we need to improve the implementation of PRCA toward offering it as a stand-alone tool and extending its supported performance metrics. In many cases, throughput and resource consumption are also needed to be monitored in addition to the response time.

9. ACKNOWLEDGMENTS

We like to thank Dennis Westermann and Alexander Wert for their constructive comments and feedback. This work is supported by the German Research Foundation (DFG), grant RE 1674/6-1.

10. REFERENCES

[1] Commons math: The apache commons mathematics library, 2012. http://commons.apache.org/math.

[2] Contiperf 2, 2012. http://databene.org/contiperf.html.

[3] Fighting regressions with git bisect manual page, 2012. www.git-scm.com/docs/git-bisect-lk2009.html.

[4] Git, 2012. http://git-scm.com.

[5] Java matrix benchmark, 2012. http://code.google.com/p/java-matrix-benchmark.

[6] Javassist, 2012. http://www.csg.is.titech.ac.jp/~chiba/javassist.

[7] Junit: A programmer-oriented testing framework for java, 2012. http://kentbeck.github.com/junit.

[8] Junitperf, 2012. www.clarkware.com/software/JUnitPerf.html.

[9] Subversion, 2012. http://subversion.apache.org.

[10] B. W. Boehm. *Software Engineering Economics.* Prentice-Hall Advances in Computing Science & Technology Series. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[11] B. W. Boehm. Software engineering economics. *Software Engineering, IEEE Transactions on*, SE-10(1):4 –21, jan. 1984.

[12] L. Bulej, T. Bureš, J. Keznikl, A. Koubková, A. Podzimek, and P. Tůma. Capturing performance assumptions using stochastic performance logic. In *Proceedings of the third joint WOSP/SIPEW international conference on Performance Engineering*, ICPE '12, pages 311–322, New York, NY, USA, 2012. ACM.

[13] L. Bulej, T. Kalibera, and P. Tma. Repeated results analysis for middleware regression benchmarking. *Perform. Eval.*, 60(1-4):345–358, May 2005.

[14] S. Chiba and M. Nishizawa. An easy-to-use toolkit for efficient java bytecode translators. In *Proceedings of the 2nd international conference on Generative programming and component engineering*, GPCE '03, pages 364–376, New York, NY, USA, 2003. Springer-Verlag New York, Inc.

[15] J. Davison de St. Germain, A. Morris, S. G. Parker, A. D. Malony, and S. Shende. Performance analysis integration in the uintah software development cycle. *International Journal of Parallel Programming*, 31:35–53, 2003.

[16] J. de St. Germain, A. Morris, S. Parker, A. Malony, and S. Shende. Integrating performance analysis in the uintah software development cycle. In H. Zima, K. Joe, M. Sato, Y. Seo, and M. Shimasaki, editors, *High Performance Computing*, volume 2327 of *Lecture Notes in Computer Science*, pages 305–308. Springer Berlin / Heidelberg, 2006.

[17] J. Ehlers, A. van Hoorn, J. Waller, and W. Hasselbring. Self-adaptive software system monitoring for performance anomaly localization. In *Proceedings of the 8th ACM international conference on Autonomic computing*, ICAC '11, pages 197–200, New York, NY, USA, 2011. ACM.

[18] K. C. Foo, Z. M. Jiang, B. Adams, A. E. Hassan, Y. Zou, and P. Flora. Mining performance regression testing repositories for automated performance analysis. In *Proceedings of the 2010 10th International Conference on Quality Software*, QSIC '10, pages 32–41, Washington, DC, USA, 2010. IEEE Computer Society.

[19] G. Fraser and A. Zeller. Generating parameterized unit tests. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 364–374, New York, NY, USA, 2011. ACM.

[20] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous java performance evaluation. *SIGPLAN Not.*, 42:57–76, October 2007.

[21] M. Grechanik, C. Fu, and Q. Xie. Automatically finding performance problems with feedback-directed learning software testing. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 156 –166, june 2012.

[22] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora. Automated performance analysis of load tests. *Software Maintenance, IEEE International Conference on*, 0:125–134, 2009.

[23] M. Jovic, A. Adamoli, and M. Hauswirth. Catch me if you can: performance bug detection in the wild. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 155–170, New York, NY, USA, 2011. ACM.

[24] H. Koziolek, S. Becker, and J. Happe. Predicting the performance of component-based software architectures with different usage profiles. In

S. Overhage, C. Szyperski, R. Reussner, and J. Stafford, editors, *Software Architectures, Components, and Applications*, volume 4880 of *Lecture Notes in Computer Science*, pages 145–163. Springer Berlin Heidelberg, 2007.

[25] D. Lee, S. Cha, and A. Lee. A performance anomaly detection and analysis framework for dbms development. *Knowledge and Data Engineering, IEEE Transactions on*, 24(8):1345 –1360, aug. 2012.

[26] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The paradyn parallel performance measurement tool. *Computer*, 28(11):37–46, Nov. 1995.

[27] A. V. Mirgorodskiy and B. P. Miller. Diagnosing distributed systems with self-propelled instrumentation. In *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '08, pages 82–103, New York, NY, USA, 2008. Springer-Verlag New York, Inc.

[28] N. Mostafa and C. Krintz. Tracking performance across software revisions. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, PPPJ '09, pages 162–171, New York, NY, USA, 2009. ACM.

[29] 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 *Proceedings of the third joint WOSP/SIPEW international conference on Performance Engineering*, ICPE '12, pages 299–310, New York, NY, USA, 2012. ACM.

[30] T. Parsons and J. Murphy. Detecting performance antipatterns in component based enterprise systems. *Journal of Object Technology*, 7(3):55–90, 4 2008.

[31] M. Rohr, A. van Hoorn, J. Matevska, N. Sommer, L. Stoever, S. Giesecke, and W. Hasselbring. Kieker: continuous monitoring and on demand visualization of java software behavior. In *Proceedings of the IASTED International Conference on Software Engineering*, SE '08, pages 80–85, Anaheim, CA, USA, 2008. ACTA Press.

[32] C. Smith. Software performance engineering. *Performance Evaluation of Computer and Communication Systems*, pages 509–536, 1993.

[33] D. Yan, G. Xu, and A. Rountev. Uncovering performance problems in java applications with reference propagation profiling. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 134–144, Piscataway, NJ, USA, 2012. IEEE Press.

[34] S. Zaman, B. Adams, and A. Hassan. A qualitative study on performance bugs. In *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*, pages 199 –208, june 2012.

[35] S. Zaman, B. Adams, and A. E. Hassan. Security versus performance bugs: a case study on firefox. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 93–102, New York, NY, USA, 2011. ACM.

[36] E. Zimran and D. Butchart. Performance engineering throughout the product life cycle. In *CompEuro '93. 'Computers in Design, Manufacturing, and Production', Proceedings.*, pages 344 –349, may 1993.