Automating the Performance Deviation Analysis for Multiple System Releases: An Evolutionary Study

Felipe Pinto[1] [2], Uirá Kulesza[1], Christoph Treude[1]
[1]Federal University of Rio Grande do Norte, Natal, Brazil
[2]Federal Institute of Education, Science and Technology of Rio Grande do Norte, São Gonçalo do Amarante, Brazil
felipe.pinto@ifrn.edu.br, uira@dimap.ufrn.br, ctreude@dimap.ufrn.br

***Abstract*—This paper presents a scenario-based approach for the evaluation of the quality attribute of performance, measured in terms of execution time (response time). The approach is implemented by a framework that uses dynamic analysis and repository mining techniques to provide an automated way for revealing potential sources of performance degradation of scenarios between releases of a software system. The approach defines four phases: (i) preparation – choosing the scenarios and preparing the target releases; (ii) dynamic analysis – determining the performance of scenarios and methods by calculating their execution time; (iii) degradation analysis – processing and comparing the results of the dynamic analysis for different releases; and (iv) repository mining – identifying development issues and commits associated with performance deviation. The paper also describes an evolutionary study of applying the approach to multiple releases of the Netty, Wicket and Jetty frameworks. The study analyzed seven releases of each system and addressed a total of 57 scenarios. Overall, we have found 14 scenarios with significant performance deviation for Netty, 13 for Wicket, and 9 for Jetty, almost all of which could be attributed to a source code change. We also discuss feedback obtained from eight developers of Netty, Wicket and Jetty as result of a questionnaire.**

***Index Terms*—Performance, execution time, scenario, dynamic analysis, repository mining.**

I. Introduction

The ability to understand and analyze how newly introduced changes impact the quality attributes of software systems when evolving them is an essential prerequisite for avoiding issues related to software erosion [1]. Degradation of quality attributes happens when a new release of a software system exhibits inferior measurements of quality attributes compared to previous releases. This work focuses on the quality attribute of performance, measured in terms of execution time (response time). We consider that performance refers to the responsiveness of the system, i.e., the time required to respond to events [2]. In order to measure performance, a set of different properties, including memory, disk and CPU usage, can be used for benchmark-based [3] or power consumption [4] [5] approaches. We chose execution time because it is a general and common property for the system responsiveness.

Our proposal consists of a scenario-based approach for evaluating software performance in terms of execution time. A scenario is a high-level action that represents the way in which the stakeholders expect the system to be used [2]. This is a common definition that is adopted by software architecture evaluation methods which explore architectural analysis of quality attributes based on scenarios [2] [6] [7]. However, they only perform early evaluation, which happens before system implementation and involves manual analysis and review of scenarios [2]. Other architectural approaches that perform code analysis focus mainly on the structural compliance of the systems [8] [9] [10] [11].

Some proposals use mathematical models for predicting quality attributes [12] [13], which represents an approximation of the real impact of the evolution. In addition, a well-known way to measure execution time and other properties is using performance benchmarks, which consist of monitoring the global resources that software systems use to perform specific tasks. They are particularly useful in helping developers determine the limits of the system through load tests [3], which is not the focus of this work. Finally, some recent research studies are exploring software repository mining techniques to infer information about performance. For example, these studies explore how performance bugs are discovered, reported to and fixed by developers [14], how repositories of performance regression-causes can be used to identify new regressions [15], how mining performance regression testing repositories can automate performance analysis and detect problems that are often overlooked by performance analysts [16], and how performance analysis risk could be applied to prioritize performance regression tests [17]. These approaches do not provide enough details about the sources of the performance deviation, for example, the methods that have contributed to degrade the performance and the commits and development issues that were responsible for introducing changes to them. That is the main limitation of current approaches that our proposal addresses.

This paper presents a scenario-based approach for automating the performance analysis of multiple system releases considering the execution time of methods and scenarios. We applied the approach to three network/web application frameworks: Netty [18], Wicket [19] and Jetty [20]. The study analyzed seven releases of each system, considering 57 scenarios in total. The approach automatically identified 14 scenarios with significant performance deviation for Netty, 13 for Wicket, and 9 for Jetty, almost all of which could be attributed to a source code change. When we asked eight contributors of Netty, Wicket and Jetty whether they had already been aware of these performance deviations

(degradation or optimization), all of them indicated that this was not the case. This preliminary evidence suggests that our approach is able to identify performance deviations that developers are not aware of.

The goal of the proposal is to provide an automated way for revealing performance deviations of scenarios of new releases, suggesting which code assets may cause performance variation and indicating commits and development issues responsible for changing them. Dynamic analysis and repository mining techniques are combined to achieve this goal, and we expect our approach and support framework to help developers identify ways to optimize the performance of their systems. The main contributions are: (i) the proposal of a scenario-based approach and tool for the automated analysis of performance deviation; (ii) the identification of the potential causes of performance deviation for Netty, Wicket and Jetty extracted from an evolutionary study through multiple releases; and (iii) a preliminary evaluation of the approach with eight contributors of Netty, Wicket and Jetty.

The rest of this paper is organized as follows: **Section 2** presents an overview of the approach. **Section 3** describes our study, including goals, research questions and results. **Section 4** discusses the obtained results. **Section 5** presents the threats to validity of the study. **Section 6** reports related work, and **Section 7** concludes the paper.

II. Approach and Framework Overview

The proposed framework, implemented in the Java programming language, automates the evaluation approach by using dynamic analysis and software repository mining techniques. Figure 1 gives an overview of the approach.

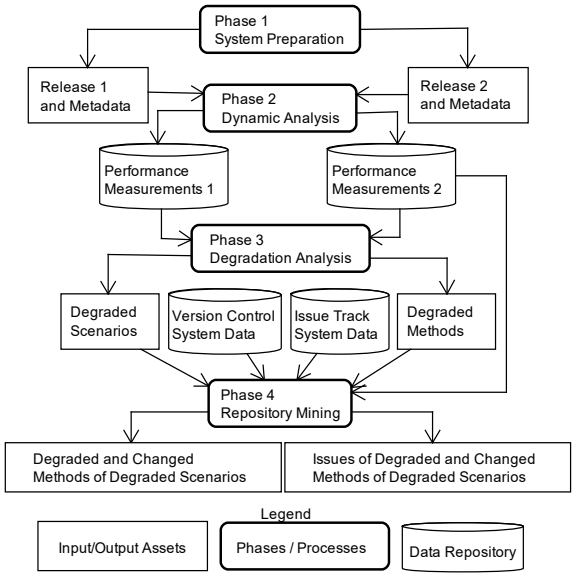

Fig. 1. Approach phases and their inputs and outputs.

The first phase requires the system's source code and additional metadata. The framework uses Java annotations as metadata source to indicate relevant scenarios that will be monitored by the dynamic analysis during the system execution, which should be selected by specialists or reused from previous architectural evaluation processes. The annotation (`@Scenario`) identifies methods in the source code that represent execution entry points of scenarios of interest. An entry point is the method that starts the scenario execution. Thus, the preparation phase output is the target releases with the integrated metadata (see Figure 1). It is also possible to use other annotations, for example, `@Test` from JUnit 4. The preparation is still a manual process, but does not require much effort because only the start methods of scenarios require the annotation (i.e., one annotation per scenario).

The second phase (dynamic analysis) requires the execution of the scenarios. The framework uses AspectJ to instrument the execution of the scenarios. This phase generates the dynamic analysis model, which is persisted in a database and contains information about the execution traces of the system modeled by a call graph that represents every execution of the selected scenarios for the target release. This call graph can be interpreted as a tree structure where each node is a call of a regular method or constructor, and the entry point methods represent the root nodes. Two databases, one for each release under comparison, are the outputs of this phase (see Figure 1).

The degradation analysis is the third phase of the approach. It compares the system execution data (dynamic analysis model) extracted during dynamic analysis for two releases. The comparison reveals methods of the system that were degraded or optimized over the evolution. The framework implements two strategies to compare the execution time: *arithmetic mean* and *statistic test*. It is possible to choose one of these strategies or both (i.e., to generate both results). The outputs of this phase are reports with degraded and optimized scenarios/methods in terms of execution time (see Figure 1), considering one of the comparison strategies (arithmetic mean or statistic test).

The first strategy compares the average execution time for each method in both releases. If the value in the newer release increased or decreased by 5% or more (this threshold can be configured in the framework), it considers that a performance deviation happened in the method. The second strategy uses a two-sided Mann-Whitney U-Test [21] to observe if two independent samples, which do not necessarily follow a normal distribution, have the same tendency. Our framework uses the U-Test to determine if the execution time of the methods of the target scenarios in the first release has the same tendency in the second release. For each method M, the first sample consists of the set of execution time values for the method in the first release, and the second sample consists of the set of values in the second release. Our null hypothesis is "the values of the execution time for method M have the same tendency in both releases", while the alternative hypothesis is "the values of the execution time for method M do not have the same tendency", in other words, they are different.

For the statistic test, our case study considered a significance level (*alpha*) of 0.05. If the *p-value* calculated using the output of the test is equal to or less than the significance level, we can reject the null hypothesis and keep the alternative hypothesis, i.e., there is a performance deviation between the releases for the method M. In that case, since it is a two-sided test and since we already know that the samples are different, the average execution time is used to determine if

it increased or decreased. Developers are usually interested in degradations, but flagging optimization cases is also interesting because developers could check if some expected modifications had indeed decreased the execution time. Despite the possibility of using the arithmetic mean strategy, we recommend the statistic test, since execution time is a very sensitive property and a pure mean strategy might not represent truly reliable values.

The last phase mines data from the version control and issue tracking systems to find which specific commits changed the methods identified previously. The framework retrieves commits from the version control system for each class that contains methods detected as degraded or optimized. If the commit changed lines inside the method, the framework searches the commit log for issue numbers, which are used to complement the information from the commits. Since the approach is guided by scenarios, the framework only considers degraded or optimized methods that impact at least one degraded or optimized scenario. The final output (see Figure 1) contains degraded/optimized scenarios and changed methods potentially responsible for affecting them, and the associated code changes (commits and development issues).

This information allows developers to analyze commits and development issues in order to understand the modifications and the reasons why they introduced performance deviation. Currently, the framework provides implementations for the Subversion and Git version control systems, and for the GitHub, Jira, Bugzilla and Issuezilla issue track systems.

III. Empirical Study: Evolutionary Analysis

In order to assess our approach, we conducted an empirical study that analyzed multiple evolutions of three different network/web application frameworks – Netty, Wicket and Jetty. The next subsections present the goals, research questions, procedures and results of the study.

A. Goals and Research Questions

The goal of our study was to assess the capacity of our approach to identify performance issues over multiple evolutions of existing systems and to check if the developers of these systems were aware of the issues. By using our approach, developers can be aware of performance issues and reduce them before distribution or deployment of new system releases. Methods and commits related to performance deviation that are discovered by our approach represent guidelines and recommendations that can be used to improve the system performance. Our study was guided by two research questions.

RQ1. Can the proposed approach find corresponding source code changes in the scenarios with performance deviation? We expect that the approach will be able to identify performance issues over multiple releases of a target system and discover their sources. It is important to understand which modifications lead to performance problems and how they could be fixed or optimized. In order to characterize these results, we also identified modules of the system (packages or classes) that concentrated most of the sources and the types of the development issues (bug/defect, new feature, improvement and others) that are most likely to cause performance deviation in the target systems. This information is useful for development teams since it will tell them what aspects of a system they should pay particular attention to when evolving the implementation.

RQ2. Are the developers aware of the performance issues that our approach was able to find? We would like to verify if developers were aware of the performance issues that our approach has found. We collected feedback from eight developers through surveys for each target system.

B. Target Systems and Procedures

Netty is an open-source asynchronous event-driven network application framework for rapid development. We chose seven releases of the fourth version of Netty, because it was the latest stable version when our study began. Thus, we selected the first and last release at that time, respectively, 4.0.0.Final and 4.0.21.Final. Each intermediate release was chosen after manually analyzing the release notes, attempting to identify stable releases that concentrated more significant changes. The seven selected releases were 4.0.0.Final, 4.0.6.Final, 4.0.10.Final, 4.0.15.Final, 4.0.17.Final, 4.0.18.Final, and 4.0.21.Final.

Wicket is an open-source web application framework developed by the Apache Foundation. We also selected seven releases, which represented the last ones at the time the study began. The selected releases were: 6.15.0, 6.16.0, 6.17.0, 6.18.0, 7.0.0-M1, 7.0.0-M2 and 7.0.0-M4. We did not include the release 7.0.0-M3 because we were unable to execute it due to compilation issues.

Jetty is an open-source framework that provides a web server and a Java servlet container. It is part of the Eclipse Project. The seven releases of Jetty that we selected were: 9.2.6, 9.2.7, 9.2.8, 9.2.9, 9.2.10, 9.3.0.M0 and 9.3.0.M1.

In order to run our framework on the target systems we have instantiated it to consider the specific version control and issue tracker systems that the target systems work with. All analyzed systems work with the Git version control system. On the other hand, they use different issue tracker systems: GitHub (Netty), Jira (Wicket), and Bugzilla (Jetty).

For the preparation phase, we selected existing automated tests of each system as scenarios. These selected tests cover important functionalities of the system and are used to reproduce their execution over different releases. In this case, we have considered the tests cases as entry points of scenarios and we have grouped the results by test classes, since they exercise similar functionalities. We are not interested in performance deviations caused by changes in test packages because we want to reveal performance issues caused by the evolution of the application source code and not just because the tests have changed. In order to do that, the framework uses a keyword for excluding certain classes or packages.

The target projects were configured to support AspectJ features and to include our framework libraries, but without any source code modification. The `@Test` annotation from JUnit was reused as scenario entry point annotation. The dynamic analysis was executed on the same computer for all

releases in the exact same conditions and with all non-essential services disabled (e.g., updates, antivirus, indexing services, and virtual memory). The computer was an AMD Phenom II with 8GB of RAM memory running the Windows 7 operating system and Java version 7. We executed the test suite of each release ten times for Netty and Wicket, i.e., each target scenario was executed ten times. For Jetty, we execute the test suite 30 times because most of the tests are shorter compared to the other systems.

After that, we grouped the seven releases of each system in six pairs of evolutions to execute the third and fourth phases. In this study, the statistic test strategy was applied instead of the arithmetic mean for comparison. As *p-value* for the U-Test, a significance level (*alpha*) of 0.05 was used. Finally, we conducted an inspection of the results to get a better understanding of them and to answer our research questions.

C. Evolutionary Analysis Results

RQ1. Can the proposed approach find corresponding source code changes in the scenarios with performance deviation? We have identified 32 scenarios with performance deviation and corresponding source code changes out of 57 (56%). Tables I, II and III summarize the results. An upward pointing arrow indicates an increase in execution time, while a downward pointing arrow indicates a decrease. Blue cells denote deviations greater than a predefined threshold for which we managed to associate source code changes. Yellow cells denote deviations we have considered not relevant because the variation was smaller than the predefined threshold (letter L), or because the changes were in parts of the source code we are not interested in such as the test packages (letter T). The red cells are deviations greater than the threshold, but they could not be associated to source code changes. In this case, it might be the result of external factors, for example, different libraries or settings, or an isolated measurement effect, as we discuss in Section V. We number evolutions from one to six (one evolution between each of the releases) for each system.

The thresholds were 15ms for Jetty and 100ms for Netty and Wicket. The Jetty threshold was chosen to be smaller because Jetty has shorter tests. These thresholds were applied in Tables I, II and III to discard very small variations, since they are probably irrelevant for developers. As we can see, for deviations greater than the thresholds (blue and red cells), the approach was able to find corresponding source code changes in most cases (blue cells), and only six cases did not have corresponding code changes (red cells). Thus, we can conclude that practically all deviations above the given thresholds can be correlated to source code changes (at least one commit), which is a strong indication that these deviations actually reflect changes in the system and are not random fluctuations in our measurements.

It is interesting to note that most scenarios exhibited degradations (or optimizations) only for a specific release. There are only a few scenarios that had performance degradation for more than one release. For example, Netty had 13 performance-degraded only in NE3, and Wicket had 6 degradations, and 12 optimizations only in WE4 and WE6, respectively. Jetty had 9 degradations in JE5.

Understanding the Performance Deviation Sources. Next, we detail the code changes corresponding to the performance deviations of the indicated scenarios. Table IV shows an example of the output with the methods that were changed and had variation in Wicket, which are the ones potentially responsible for performance deviation, including the method name, performance impact, commits and number of impacted scenarios. The results are ordered by the performance impact of each method, which is calculated as the arithmetic mean of the impact of the method in each scenario that it affects. The impact of a method is the total time it takes running in a particular scenario. This is just one strategy to show the results, and it does not necessarily imply that methods in the top of the table have not caused more performance issues.

TABLE I. DEGRADED AND OPTIMIZED SCENARIOS OF NETTY.

Scenarios (Test Classes)	E1	E2	E3	E4	E5	E6
Entry Point for DatagramUnicastTest			↑		↓	↑
Entry Point for SocketBufReleaseTest		↓L	↑	↑L	↓	
Entry Point for SocketCancelWriteTest					↓L	↑L
Entry point for SocketConnectionAttemptTest				↑L	↓L	
Entry Point for SocketEchoTest		↑L	↑	↑L	↓L	↑L
Entry Point for SocketFileRegionTest		↑L	↑	↑L	↓L	
Entry Point for SocketFixedLengthEchoTest			↑	↑L	↓L	
Entry Point for SocketObjectEchoTest		↑L	↑	↑L	↓L	
Entry Point for SocketShutdownOutputByPeerTest	↓L	↑L	↑	↑L	↓L	
Entry Point for SocketShutdownOutputBySelfTest		↑L	↑		↓L	
Entry Point for SocketSpdyEchoTest		↑L	↑	↓T	↓L	↓
Entry Point for SocketSslEchoTest	↑	↓	↑	↑L	↓	↓
Entry Point for SocketStartTlsTest		↑	↑	↑L	↓L	↓
Entry Point for SocketStringEchoTest	↑L		↑	↑L	↓L	↓
Entry point for UDTClientServerConnectionTest		↑				
Entry Point for WriteBeforeRegisteredTest		↑L	↑	↑L	↓L	

TABLE II. DEGRADED AND OPTIMIZED SCENARIOS OF WICKET.

Scenarios (Test Classes)	E1	E2	E3	E4	E5	E6
Entry point for AjaxTest				↑L		↓
Entry point for CompressTest				↑	↑L	↓
Entry point for EncodingTest		↑L		↑		↓
Entry point for FormInputTest				↑		
Entry point for GuestbookTest						↓
Entry point for HangManTest	↑L			↑		↓
Entry point for HelloWorldTest		↓L				↓
Entry point for ImagesTest						↓
Entry point for LibraryTest						↓
Entry point for LinkomaticTest				↑		↓
Entry point for NiceUrlTest		↑L	↓L	↑		↓
Entry point for Signin2Test						↓
Entry point for TemplateTest						↓
Entry point for WordGeneratorTest				↓T		

TABLE III. DEGRADED AND OPTIMIZED SCENARIOS OF JETTY.

Scenarios (Test Classes)	E1	E2	E3	E4	E5	E6
Entry point for AsyncContextListenersTest					↑	
Entry point for AsyncContextTest		↓L	↑L	↓L	↑L	↓L
Entry point for AsyncIOServletTest		↓L		↓L	↑	↑L
Entry point for AsyncServletLongPollTest					↑	
Entry point for AsyncServletTest				↑L	↑L	↑L
Entry point for DefaultServletRangesTest		↑L			↑	
Entry point for DefaultServletTest					↑L	↑T
Entry point for DispatcherForwardTest				↓L	↑	
Entry point for DispatcherTest		↑L	↓L	↓L	↑L	↑L
Entry point for ErrorPageTest			↑L		↑	↓L
Entry point for InvokerTest					↑	
Entry point for RequestHeadersTest					↑L	
Entry point for ResponseHeadersTest					↑L	
Entry point for ServletContextHandlerTest					↑	↓L
Entry point for ServletHandlerTest		↑L		↑L	↓L	
Entry point for SSLAsyncIOServletTest					↑	

TABLE IV. SOURCES OF PERFORMANCE DEVIATION FOR WICKET.

Evolution	Methods	Number of Scenarios	Number of Commits	Issues	Performance Impact
WE4	MarkupContainer.addedComponent	7	5	5410, 3335	605ms
	MarkupContainer.add	7	1	3335	603ms
	MarkupContainer.dequeue	7	10	3355	548ms
	Page.onBeforeRender	7	1	5426	406ms
	WebPageRenderer.isPageStateless	7	1	5426	400ms
	WebPageRenderer.shouldRenderPageAndWriteResponse	7	5	5426, 5484, 5522	380ms
	WebPageRenderer.respond	7	6	3347, 5309, 5426	370ms
	ListView.onPopulate	7	1	-	357ms
	MarkupContainer.newDequeueContext	7	1	-	331ms
	AbstractRepeater.dequeue	7	5	3335	215ms
	DefaultPageFactory.newPage	6	1	5215	159ms
	Page.renderPage	7	1	5426	144ms
	WebPageRenderer.renderPage	7	1	-	130ms
	MarkupContainer.onInitialize	7	1	-	118ms
	Component.internalRenderHead	7	1	4964	108ms
	Initializer.register	1	1	-	106ms
	MarkupContainer.dequeueAutoComponents	7	3	3335	105ms
	Initializer.createProxy	1	1	-	102ms
	Initializer.init	1	2	-	95ms
	MarkupCache.loadMarkupAndWatchForChanges	7	2	5294	54ms
	MarkupCache.loadMarkup	7	2	5294	53ms
	MarkupContainer.canDequeueTag	1	6	3335	35ms
	Component.setMetaData	1	1	5459	27ms
	JavaSerializer.deserialize	3	1	-	22ms
	WebPageRenderer.shouldRedirectToTargetUrl	1	4	5426	20ms
	XmlPullParser.parse	6	1	5398	16ms
WE6	WebPageRenderer.respond	12	2	5689	614ms
	MarkupContainer.dequeueAutoComponents	12	1	5730	356ms
	Application.initializeComponents	5	1	5713	342ms
	MarkupContainer.newDequeueContext	5	1	5730	47ms

The scenario names and commit codes are not presented in Table IV due to space constraints. For the same reason, we will not detail every source code change that was found. The complete study data for all releases is available online [22]. The full description of commits can be found in the Netty [18], Wicket [19] and Jetty [20] version control systems. In the following, we describe part of the information that our approach automatically extracted from the repositories.

Netty: From release 4.0.0.Final to 4.0.6.Final (NE1). The framework found four commits and an improvement issue (#1606). Most of the changes affected the `validatePromise()` method by adding some extra code validation. A new way to instantiate the `ChannelOutboundBuffer` class in order to make its objects recycled was introduced in the `newInstance()` method. The commit added 7 and deleted 45 lines of code, respectively, and there is a performance improvement when objects from `ChannelOutboundBuffer` are recycled, and Netty avoids creating them again, but for new objects the performance decreased.

Netty: From release 4.0.6.Final to 4.0.10.Final (NE2). All versions in between these releases were bug fixing with some improvements. The framework found five commits and three development issues, including one improvement (#1707) and two unlabeled issues (#1697 and #1832). An important unlabeled issue (#1697), also highlighted in the releases notes of 4.0.7.Final, fixed a bug related to buffer management. The solution introduced a new way to estimate the size of messages that should be written in buffers. Another interesting change introduced by a commit was intended to fix a callback problem when writing to a channel in 4.0.8.Final.

Netty: From release 4.0.10.Final to 4.0.15.Final (NE3). For the third evolution, the framework detected that most of the scenarios were degraded (13 out of 20 scenarios). Seven commits and three development issues were found related to seven methods. The issues were two bug fixing (#1908 and #2060) and one unlabeled (#1947). The unlabeled issue (#1947) changed the `DefaultChannelHandlerContext` class in order to deal with a problem related to reject execution exceptions. It also added a new method named `safeExecute()` to the class, which affected part of the degraded scenarios during this evolution. One bug fixing issue (#1908) introduced changes to the method `NioEventLoop.openSelector()` to validate if internal objects are assignable. Another interesting change was a commit intended to improve the buffer leak report, which introduced wrappers. Now, a leak-aware buffer can detect and report memory leaks, as result of 1800 added lines and 17 changed files. Table I (NE3) shows the list of performance-degraded scenarios in this evolution.

Netty: From release 4.0.17.Final to 4.0.18.Final (NE5). As shown in Table I (NE5), this evolution has optimized some scenarios. Two commits related to three development issues, including two improvements (#808 and #2264) and one new feature (#2311), were found. One of the modifications has introduced changes to the `PoolThreadCache` and `PoolArena` classes and is responsible for the optimization. According to the commit description, the changes "remove the synchronization bottleneck in `PoolArena` and so speed up things". The problem was solved by improving the synchronization and implementing a thread-local cache for pooled buffers.

Netty: From release 4.0.18.Final to 4.0.21.Final (NE6). The framework detected one commit responsible for the optimized scenarios (see Table I), which changed the `DefaultChannelPipeline` class to improve memory usage and initialization time. It refactored some source code and modified the strategy to generate the names of a communication channel.

Wicket: From release 6.18.0 to 7.0.0-M1 (WE4). This was the only evolution with significant performance degradation for Wicket (see Table II). The framework found 38 commits and 12 issues (one new feature, four improvements, and seven bugs). A new feature (#3335) implemented a queueing strategy for adding and extracting hierarchy information from markup. It added a substantial amount of new code to 14 files, including `MarkupContainer`, `Page`, and `AbstractRepeater`. Methods such as `add()`, `addedComponent()` and `queue()` introduced new validations. Obviously, new features and new code might cause execution time increases, but it is a team/developer decision to say if these increases are suitable or not. Our approach automatically detects the deviations and corresponding changes, so that developers can be aware of the specific consequences of their work. Another change caused by a bug fixing issue (#5426) corrected problems related to component states when they are rendering. One of the main classes changed was `WebPageRenderer`. An improvement issue (#3347) tried to simplify the way in which the `WebPageRenderer.respond()` method decides whether it will redirect or directly render the current page depending on several complex conditions. This issue resulted in 830 added lines and 95 deleted lines.

Wicket: From release 7.0.0-M2 to 7.0.0-M4 (WE6). For this evolution, the framework detected only performance optimization in terms of execution time. We believe that the change from version 6.x to 7.x (WE4) introduced many problems and unsolved situations due to unstable code that was responsible for the degradation in previous releases, which were then addressed in this evolution. A bug fixing issue (#5689) changed a lot of source code in order to solve conflict problems in the `WebPageRenderer` class. An extra commit also changed this class in order to refactor and improve the `respond` method, replacing a big part of the code introduced by the issue #3347 in WE4. Another bug fixing issue (#5730) simplified and corrected the de-queueing component process.

Jetty: From release 4.2.10 to 4.3.0.M0 (JE5). This was the only evolution in which the framework found significant performance deviation for Jetty. Changes were introduced in three methods and related to seven commits and one bug issue (#439375). The main changes were introduced by a commit that aimed to pre-encode HTTP fields. It modified 25 files with 449 line additions and 147 line deletions, which was enough to affect all six blue scenarios from Table III.

It is out of the scope of this paper to provide a detailed description of each commit and development issue found. We conducted a manual inspection of the results in order to better understand the changes and to ensure that they make sense. The data with all methods and issues are available online [22] and the links to version control and issue track systems can be found at the Netty [18], Wicket [19], and Jetty [20] websites.

In order to characterize these results and to provide developers with a better understanding of them, Figures 2 and 3 show how the total number of degraded and optimized methods of scenarios are spread over the packages for each evolution. The complete data that shows how these methods are spread over the classes is also available online [22].

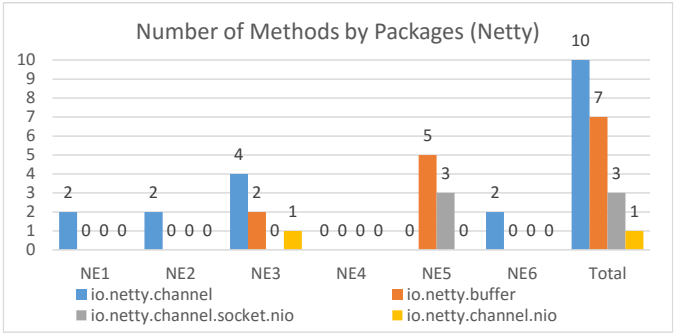

Fig. 2. Number of Methods by Packages for Netty.

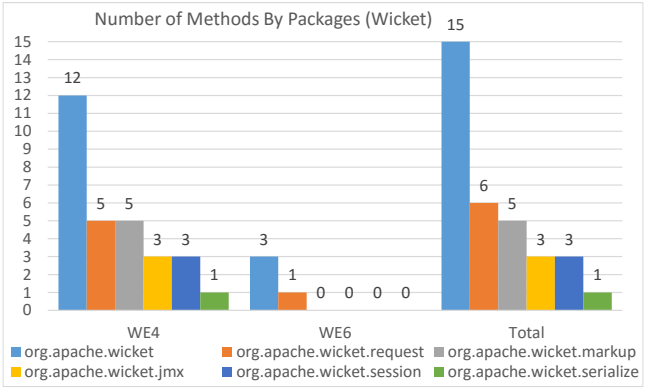

Fig. 3. Number of Methods by Packages for Wicket.

We identified only four packages that contain all 21 methods responsible for performance deviations in Netty. According to Figure 2, the `channel` package concentrates the majority of the methods potentially responsible for the variations with a total of ten methods, which are distributed over ten classes. Thus, the `channel` package may represent one of the sensitive points for performance in Netty.

For Wicket, the `MarkupContainer` and `WebPageRenderer` classes contain the majority of the methods responsible for performance deviations (ten and six, respectively). The classes inside the `org.apache.wicket` package also concentrated most of these methods, 15 in total (see Figure 3). These numbers are indicators that developers should pay particular attention when evolving these code assets because changing them might lead to performance deviation that could increase or decrease the execution time of methods and scenarios.

For Jetty, we found only three methods that belong to the `MimeTypes$Type`, `ServerConnector` and `PathResource` classes from the `org.eclipse.jetty.http`, `org.eclipse.jetty.server` and `org.eclipse.jetty.util.resource` packages, respectively.

Based on the results, we conclude that the `channel` package from Netty, and the `MarkupContainer` and `WebPageRenderer` classes as well as the `org.apache.wicket` package from Wicket are the parts of the systems that concentrate most of the sources of performance variation and represent elements that developers should pay attention to when evolving the systems. On the other hand, there was no package or class that concentrated most of the deviation-related changes for Jetty.

To complete the characterization of the study results we also identified the following kinds of issues associated with

deviations: (i) improvement (9 issues), (ii) bug/defect (12 issues), (iii) new feature (2 issue), and (iv) unlabeled (3 issues). Defining the issue type is not mandatory in most of the issue tracker systems, and not all commits are associated with an issue. We found that 32 commits out of 66 – 10 for Netty, 16 for Wicket and 6 for Jetty – are not linked to any issue. This represents almost 50% of the commits we found, which makes it difficult to draw conclusions about the types of issues that are most likely to cause performance deviation. Table V summarizes the types of issue for each system.

TABLE V. KINDS OF ISSUES FOR EACH SYSTEM.

Type	Netty	Wicket	Jetty	Total
Unlabeled	3	0	0	**3**
Improvement	4	5	0	**9**
Bug/defect	2	9	1	**12**
New Feature	1	1	0	**2**
Total:	**10**	**15**	**1**	**26**

RQ2. Are the developers aware of the performance issues that our approach was able to find? Performance deviation is difficult to notice, it may occur in a progressive way and some problems might be realized only after multiple evolutions. We collected feedback from eight developers using a web-based survey to investigate if they were aware of the deviations we found and to give us a preliminary perception of what they think about the usefulness of our approach and results.

We prepared surveys for developers of Netty, Wicket and Jetty. They asked developers about tools and strategies they usually use to manage performance and if they were aware of some of the performance deviations that our approach found. The total number of developers, extracted from the contributor pages on GitHub, is 100, 30 and 20, for Netty, Wicket and Jetty, respectively. We contacted the developers that had published their email address, and received responses from four, three and one developer for those systems, respectively.

For the scenarios with identified with performance degradation, we gave three examples with increase of time that we judged representative samples and important functionalities to the participants of each system and asked questions in order to investigate whether they already were aware of the variations. The examples were (extracted from Tables I, II and III): (i) Netty – `WriteBeforeRegisteredTest`, `SocketEchoTest`, and `SocketObjectEchoTest`; (ii) Wicket – `HangManTest`, `NiceUrlTest`, and `LinkomaticTest`; (iii) Jetty – `AsyncServletLongPollTest`, `AsyncIOServletTest`, and `DispatcherForwardTest`.

On the last page of the survey, we presented an overview of the approach and we summarized the main study results with a table, similar to Table IV for Wicket. This information was used to introduce the approach and ask what participants think about it, including different strategies they would use, and to ask for additional comments.

In response to the question "*In your opinion, how important is execution time (response time) for the system?*", seven participants confirmed that performance in terms of execution time is important in the context of these systems, while only one was neutral about it. Some participants (3 out of 8) were aware that some releases had execution time issues when we asked them "*Are you aware of any execution time variation (increase or decrease) in any of these releases: <list of target releases we selected for each system>?*".

However, when we asked "*We noticed that the execution time increased for several test cases between releases 6.18.0 and 7.0.0M1. After that, the execution time decreased again between releases 7.0.0M2 and 7.0.0M4. This was the case for scenarios tested by classes such as HangManTest, NiceUrlTest, and LinkomaticTest. Are you aware of these execution time variation?*" and "*Considering the examples of the previous question (HangManTest, NiceUrlTest, and LinkomaticTest), what methods, commits, or development issues caused the increase in execution time?*", none of our participants were aware of the specific execution time variations that our approach had identified. We recognize these questions might be very difficult to answer, but it confirms that without a suitable performance analysis tool it is very difficult to indicate causes, even if developers are aware of the performance deviations. These questions were adapted for each system with the appropriate target releases and the three selected examples indicated previously.

In addition, the majority of our participants (5 out of 8) indicated the usefulness of the approach, while two were neutral, and only one thought it was not very useful. Three participants also mentioned the profiling tool YourKit when we asked about tools they usually use for performance testing. However, this tool does not automatically compare execution time between releases, which is why one of the participants said about our work: *"Nice tool if it is really automatic since currently comparison is a manual process"*. In addition, one of the participants was going to check specific methods based on the data we provided in the survey: *"It is interesting enough that I will be looking at* `MimeType$Type` *and* `checkAliasPath` *changes to double check we've not done anything stupid"*.

Some participants were concerned about micro-benchmarking, mainly because of the use of test cases for performance testing, since test cases can be very small and may not have practical impact. We recognize that this may be a threat in our study especially for Jetty that has the smallest test cases. However, the use of test cases as scenarios was simply a choice we made for the evaluation study. Because of the usage of annotations, the approach is flexible and can instrument any part of the code. Thus, it is a limitation of the study rather than of the tool or approach. Nevertheless, it is interesting to notice that even when using test cases, the framework had found several performance degradations in scenarios.

In this context, it is also important to analyze the significance of the deviation. For very small deviations, developers may not notice the variation and will not fix it in a future release because it actually does not matter. A workaround is to configure the framework to only report results with a specific minimum impact. This can be currently done by defining different values for predefined thresholds.

We conclude that, in general, our approach can find performance deviations that developers are not aware of. They could realize such deviation by running a profiling tool, but these tools do not automate the comparison of different

releases and they do not provide details about methods, commits, and issues responsible for deviations. Additional details and data regarding the survey are available online [22].

IV. Discussion and Limitations

Empirical Study Conclusions. The results returned by the framework and our inspection show the feasibility of the proposed approach to evaluate performance of scenarios in terms of execution time. The approach identified degraded and optimized scenarios over the evolutions of Netty, Wicket and Jetty, and determined the potential causes of such variation by indicating code assets, development issues, and commits **(RQ1)**. The feedback obtained through surveys indicated that developers are unaware of the performance variations we found **(RQ2)**. In addition, we were able to characterize the modified methods responsible for the performance deviation by highlighting packages and classes where such methods were declared and associated issues.

AspectJ instrumentation. Our framework uses AspectJ to instrument the execution of scenarios, intercepting the entry point methods – annotated with `@Scenario` in the first phase of the approach (see Section II) – to build the call graph and collect the execution time of scenarios and methods. AspectJ was the most appropriate way we found to cope with the complexity of dynamic analysis and it is a common solution adopted for other studies [11] [23]. Dynamic analysis generates a large amount of data, even for non-large-scale systems, which becomes a problem when the persistence of the traces that represent the call graph of the system execution is required for posterior analysis. To overcome this situation, a solution that integrates database persistence after each scenario execution should be used. In this context, implementing our own instrumentation mechanism that uses specific models for our needs proved to be more suitable than using the output of other profiling tools, such as YourKit, JProfiling, or JMH.

Approach Execution Challenges. We recognize that our approach and current framework implementation have some execution challenges. The first one is the need for the manual annotation of the scenarios (when not using JUnit annotation), which requires architectural knowledge of the target software. For systems with automated functional tests, these can be used as evaluation scenarios. Another requirement is the availability of all code artifacts in different versions as well as traceability data between development issues and commits.

Execution Time Limitations. The performance was measured only in terms of execution time. Other possible metrics for performance are memory consumption, disk activity, and CPU usage, for example. We are currently analyzing new performance properties for future studies, since we know that for some systems, memory, for example, might be a more relevant performance requirement. Another problem is that new lines of code caused by the addition of new features or bug fixes might potentially increase the execution time, which will be detected by the framework. However, it is important to realize that some deviations cannot be avoided, and, in such cases, the developers need to decide if the increase is suitable or not, or even be aware of the impact of such changes to the system scenarios. We consider that the primary use case of our approach is when it finds deviations which developers did not expect. This will allow them to investigate the deviations further. In case of expected deviations, our approach can confirm exactly which scenarios were affected.

V. Threats to Validity

Measuring Risks. Our approach relies on multiple executions of scenarios to increase the confidence in the measurements. Even though we executed the test suite only ten times for each release (30 for Jetty) in the case study, a particular method might be executed much more often. For example, considering the repetitions, the Wicket method `respond()` (Table IV) was executed 30 times inside the *Entry point for NiceUrlTest* because it is called three times as part of the corresponding test case. On the other hand, the method `addedComponent()` was executed 1020 times inside the same entry point. In these cases, we are able to obtain samples that are more representative. This and other precautions, such as disabling every non-essential service of the environment, running each test in its own VM, using a random order for each repetition and conducting a manual inspection of the results help us to decrease the risks of measurement bias [24] due to the high sensitivity of measuring execution time.

JUnit Tests and Micro-benchmarking. Some developers were concerned about micro-benchmarking, since test cases may not have practical impact for execution time because they were not written for performance-testing purposes. We recognize this threat, but the use of test cases as scenarios was simply a choice we made for the evaluation study. Developers could use the provided scenario annotations instead of the test annotation since the approach can instrument any part of the code. Thus, any strategy to exercise the scenarios will work. Despite the usage of such automated system tests in our study, they still allowed us to find degradation scenarios for the investigated systems.

Impact of Instrumentation. The instrumentation process causes another threat related to the measurement strategy. It needs to intercept the methods during the system execution, which might affect the execution by contributing to increase the execution time. We have not measured the impact that instrumentation causes to execution time in this context, but we believe it does not form a problem since our analysis compares pairs of values of execution time from two releases, and the instrumentation should cause the same or very similar increases in both values.

Rename Problem. The framework currently considers renamed code elements as new elements that will potentially affect the performance of scenarios. However, our manual inspection showed that for this evolutionary study, names were consistent in general. We could also highlight these methods for the developers to indicate which of them might not be new. It could be problematic for refactoring because the approach might indicate many changes, but as mentioned in the previous section (*Execution Time Limitations*), developers should already expect that.

Results: Generalization and Limitation. Some results of our study cannot be generalized to other releases of the target systems. For example, we tried to characterize the kinds of development issues that are most likely to contribute to performance deviation. The results showed it was bug/defect, but it is still a small amount of issues. We also recognize the low number of developers that participated in our surveys. However, while we cannot generalize these results, they offer preliminary evidence that our approach and support framework are able to find performance issues that developers are not aware of. In addition, we could not check if every commit flagged was correctly selected by the framework because to the best of our knowledge none of these systems have any kind repository to keep or flag performance regression changes.

VI. Related Work

Any research work related to evolution and performance could be relevant for this work. However, we noticed that there has not been much work focusing on the identification of the sources of performance deviation, considering software evolution, dynamic analysis and mining software repositories. Thus, the main novelty of this work is the possibility of automatically indicating the causes of detected performance deviations for scenarios in terms of methods and corresponding source code changes, what is achieved by the combination between dynamic analysis and repository mining techniques. The next paragraphs detail some related research work.

Malik et al. [3] propose strategies for helping performance analysts to more effectively compare results of load tests to find performance deviations in large-scale systems. They provide a reduced and manageable number of measurements, such as CPU and memory utilization, related to performance deviation by comparing two releases. Their case study is based on load tests from an industrial and an open source system. The identification of the problems introduced during the evolution are indicated in terms of measurements and their related elements. There is no mention of repository mining or attempts to identify the changes related to the performance problems.

Koziolek et al. [25] present a methodology to predict the quality attributes of performance and reliability using response time and failure rate. They evaluate a large-scale control process system. The goal was to quantitatively predict the quality attributes for different architectural alternatives and then to choose the best alternative considering the trade-off among them. Their work differs from ours, which focuses on the analysis of existing system releases in order to detect existing performance deviation and their potential causes.

Nguyen et al. [15] propose mining a regression-causes repository to identify causes of new performance regressions. The repository contains the results of performance tests and causes of past regressions. They use machine-learning techniques to determine the causes of new regressions based on data from the repository. The causes are a pre-defined set of situations extracted from bug reports that represent actions that usually cause performance regression, such as adding frequently executed logic or adding blocking I/O access. Thus, the authors are able to categorize causes of new performance regression based on past data. Our approach does not categorize causes of regressions, although it indicates a set of commits, which is a more detailed, and fine-grained result, but developers have to interpret the results themselves. Nguyen's work does not mention any usage of dynamic analysis or repository mining for providing more fine-grained results.

Foo et al. [16] introduce an automatic approach to derive performance signatures by capturing the correlations among metrics in performance regression repositories and comparing new test results against these correlations. The reports signal potential problematic metrics that violate the extracted performance signatures. Performance analysts can leverage the report to ensure better coverage in their assessments of performance regression tests and to derive the causes. Foo's approach is able to reveal performance regressions related to different performance attributes, not only execution time, but the performance analysts still need to derive the causes manually, what could be time-consuming. In our approach, the framework is able to indicate the deviations in more fine-grained way (methods) and the corresponding changes (commits and development issues).

Ghaith et al. [26] conducted an experiment to show that a transaction profile approach, which the authors consider a load independent representation of transaction response time, can detect performance anomalies when applied to two different releases of a web application. The first release was used as a baseline, while a known anomaly was added to the second one to cause extra processing. Despite the similarity of comparing performance of software releases, the possibility of discovering potential causes of performance deviation is not present in their work and there is no indication for future support.

Finally, there are some works focused on the impact of changes on software energy consumption [4] [5], which are also related to this work, since poor performance may increase software energy consumption. In this context, Hindle [4] has proposed a green mining methodology of relating software changes to power consumption. The main goal is to give recommendations based on past evidences extracted by looking at each change in a version control system and dynamically measuring its effects on power consumption and alerting developers before they make a software change that negatively affects power consumption. The power tests dynamically measure the resources used by the system in a global way by monitoring CPU, disk and memory usage of the entire system. Thus, it is not possible to give a detailed report relating the changes to specific code assets, such as methods. There is no mention if the approach can also help developers by indicating the sources when the changes have already been made and an energy consumption regression was introduced.

VII. Conclusion

This paper introduces a framework that automates an approach for scenario-based evaluation of performance. To demonstrate the feasibility of the proposed approach, we presented an evolutionary study aimed at performing a scenario-based evaluation of the quality attribute of performance, in terms of execution time, for multiple releases of Netty, Wicket and Jetty. Through the study results, we

expect to help developers detect performance issues before releasing software systems, and easing the process of fixing these issues by identifying their causes.

Based on the analysis over multiple releases, we found 13 changed-degraded scenarios out of 20 analyzed scenarios for Netty, 6 out of 16 for Wicket, and 6 out of 21 for Jetty. The potential causes of these deviations were found in the form of methods, commits and issues. Our approach was able to identify scenarios with performance deviation that developers were unaware of and it also identified the classes and packages that contain most of the sources. Our results indicate that the approach is feasible and useful for helping developers to identify and understand the reasons of performance problems because it is able to substantially reduce the amount of information that developers have to analyze manually. Thus, the framework can be used as a preventive tool.

We are working on several directions to improve our results: (i) planning how to use our approach in the development process of a software company; (ii) conducting new studies to measure the impact of the instrumentation process during system execution; and (iii) investigating which are the features of commits that are more likely to lead to performance deviation in terms of execution time.

ACKNOWLEDGMENT

This work is partially supported by the Federal Institute of Rio Grande do Norte (IFRN), the National Institute of Science and Technology for Software Engineering (INES), CNPq grants 573964/2008-4 and 552645/2011-7, and CAPES/PROAP.

REFERENCES

[1] L. Silva, D. Balasubramaniam. Controlling software architecture erosion: A survey. J. Syst. Softw. 85, 1 (January 2012), 132-151.

[2] P. Clements, R. Kazman, M. Klein. 2002. Evaluating Software Architectures: Methods and Case Studies. Addison-Wesley.

[3] H. Malik, H. Hemmati, A. E. Hassan. Automatic detection of performance deviations in the load testing of large scale systems. In Proceedings of the ICSE 2013. IEEE Press, Piscataway, NJ, USA, 1012-1021.

[4] A. Hindle. Green mining: a methodology of relating software change to power consumption. In Proceedings of the Working Conference on MSR 2012. IEEE Press, Piscataway, NJ, USA, 78-87.

[5] R. Perez-Castillo, M. Piattini. Analyzing the Harmful Effect of God Class Refactoring on Power Consumption. IEEE Software. 31, 3 (April 2014), 48-54.

[6] M. Ali Babar, I. Gorton. Comparison of Scenario-Based Software Architecture Evaluation Methods. In Proceedings of the APSEC 2004. IEEE Computer Society, Washington, DC, USA, 600-607.

[7] B. Roy, T. C. N. Graham. Methods for Evaluating Software Architecture: A Survey. Technical Report 2008-545, School of Computing, Queen's University at Kingston, Ontario, Canada.

[8] D. Ganesan, M. Lindvall, R. Cleaveland, R. Jetley, P. Jones, Y. Zhang. Architecture Reconstruction and Analysis of Medical Device Software. In Proceedings of the WICSA 2011. IEEE Computer Society, Washington, DC, USA, 194-203.

[9] D. Ganesan, T. Keuler, Y. Nishimura. Architecture compliance checking at run-time. Information and Software Technology. 51, 11 (November 2009), 1586-1600.

[10] M. Abi-Antoun, J. Aldrich. Static extraction and conformance analysis of hierarchical runtime architectural structure using annotations. In Proceedings of the OOPSLA 2009. ACM, New York, NY, USA, 321-340.

[11] S. Ciraci, H. Sozer, B. Tekinerdogan. An Approach for Detecting Inconsistencies between Behavioral Models of the Software Architecture and the Code. In Proceedings of the COMPSAC 2012. IEEE Computer Society, Washington, DC, USA, 257-266.

[12] S. S. Gokhale. Architecture-Based Software Reliability Analysis: Overview and Limitations. IEEE Trans. Dependable Secur. Comput. 4, 1 (January 2007), 32-40.

[13] L. G. Williams and C. U. Smith. PASA^{SM}: a method for the performance assessment of software architectures. In Proceedings of the WOSP 2012. ACM, New York, NY, USA, 179-189.

[14] A. Nistor, T. Jiang and L. Tan. Discovering, reporting, and fixing performance bugs. In Proceedings of the Working Conference on MSR 2013. IEEE Press, Piscataway, NJ, USA, 237-246.

[15] T. H. D. Nguyen, M. Nagappan, A. E. Hassan, M. Nasser, P. Flora. An industrial case study of automatically identifying performance regression-causes. In Proceedings of the Working Conference on MSR 2014. ACM, New York, USA, 232-241.

[16] K. C. Foo, Z. M. Jiang, B. Adams, A. E. Hassan, Y. Zou, P. Flora. Mining Performance Regression Testing Repositories for Automated Performance Analysis. In Proceedings of the QSIC 2010. IEEE Computer Society, Washington, DC, USA, 32-41.

[17] P. Huang, X. Ma, D. Shen, and Y. Zhou. Performance regression testing target prioritization via performance risk analysis. In Proceedings of the ICSE 2014. ACM, New York, NY, USA, 60-71.

[18] The Netty Project. August 2015: http://netty.io.

[19] Apache Wicket. August 2015: https://wicket.apache.org.

[20] Jetty Project. August 2015: http://eclipse.org/jetty.

[21] M. Neuhäuser. International Encyclopedia of Statistical Science: Wilcoxon–Mann–Whitney Test. Lovric, Miodrag, 2014. ISBN 978-3-642-04897-5. Pages: 1656-1658.

[22] Automating the Performance Analysis for Multiple System Releases: an Evolutionary Study. August 2015: https://sites.google.com/site/perfevolutionarystudy.

[23] R. Holmes, D. Notkin. Identifying program, test, and environmental changes that affect behaviour. In Proceedings of the ICSE 2011. ACM, New York, NY, USA, 371-380.

[24] T. Mytkowicz, A. Diwan, M. Hauswirth, P. F. Sweeney. Producing wrong data without doing anything obviously wrong! In Proceedings of the ASPLOS 2009. ACM, New York, NY, USA, 265-276.

[25] H. Koziolek, B. Schlich, S. Becker, M. Hauck. Performance and reliability prediction for evolving service-oriented software systems. Empirical Softw. Engg. 18, 4 (August 2013), 746-790.

[26] S. Ghaith, M. Wang, P. Perry, J. Murphy. Profile-Based, Load-Independent Anomaly Detection and Analysis in Performance Regression Testing of Software Systems. In Proceedings of the CSMR 2014. IEEE Computer Society, Washington, DC, USA, 379-383.