Hindawi Publishing Corporation
Mathematical Problems in Engineering
Volume 2015, Article ID 690829, 9 pages
http://dx.doi.org/10.1155/2015/690829

Research Article

Distance Based Root Cause Analysis and Change Impact Analysis of Performance Regressions

Junzan Zhou and Shanping Li

College of Computer Science and Technology, Zhejiang University, Hangzhou 310012, China

Correspondence should be addressed to Junzan Zhou; zhoujunzan@zju.edu.cn

Received 15 February 2015; Revised 7 May 2015; Accepted 11 May 2015

Academic Editor: Evangelos J. Sapountzakis

Performance regression testing is applied to uncover both performance and functional problems of software releases. A performance problem revealed by performance testing can be high response time, low throughput, or even being out of service. Mature performance testing process helps systematically detect software performance problems. However, it is difficult to identify the root cause and evaluate the potential change impact. In this paper, we present an approach leveraging server side logs for identifying root causes of performance problems. Firstly, server side logs are used to recover call tree of each business transaction. We define a novel distance based metric computed from call trees for root cause analysis and apply inverted index from methods to business transactions for change impact analysis. Empirical studies show that our approach can effectively and efficiently help developers diagnose root cause of performance problems.

1. Introduction

Performance is an important nonfunctional requirement for software systems. To assure the software quality, performance testing is an effective way for evaluating system performance, which can uncover high response time and low throughput issues [1]. For new software releases, performance regression tests are often conducted to check whether any performance slow-down or new performance problem had been introduced.

Mature performance testing process can help reveal performance problems. However, it is difficult for testers to diagnose the root cause of performance issues, which needs much expertise to diagnose a root cause. Developers can apply several methods, including performance profiling, performance debugging, and log analysis techniques, to look into performance problems. Performance profiling and debugging are more suitable for performance issues under a low load during development phase. Performance profiling collects detailed performance metrics by instrumentation or sampling [2]. Fine grained instrumentation [3] can collect more information, but, contrary to what we suppose, it may distort the realistic time distribution among functions. This can be augmented under heavy load. Performance debugging can be performed by experienced developers. However, performance debugging is often guided by developers' empirical experience, which may be not efficient enough. Developers can also leverage logs to find out the root cause. The problem is that it is hard for developers to manually analyze large sized logs generated during a load test.

Existing performance testing research focuses on test suite generation, workload models, and load testing frameworks. As far as we know, there is limited work, which proposed the systematic analysis of root cause of performance issues. For analysis of performance test result, Jiang et al. leveraged system level metrics and execution logs for uncovering performance issues and generating load test report [4]. In a realistic project, we should not only pinpoint a performance problem, but also find out the root cause. From the engineering view, we have to evaluate the potential improvement and side effect of adopting a modification. However, little work has been found for such problems in the domain of software performance engineering. What is more, performance analyst should also evaluate the potential improvement and change impact by refining the root cause.

In this paper, we present a root cause analysis and change impact analysis approach of performance regressions. It has been integrated with LTF [5] to optimize the automation level

of performance testing. First, we leverage server side logs to obtain call graphs of each transaction. Next, we compute a novel distance based metric for identifying root causes. Then, we apply inverted index from methods to business transactions for change impact analysis. Our approach improved the efficiency and effectiveness of root cause analysis and change impact analysis in terms of performance.

The main contributions of our work are as follows:

(1) We proposed a framework for automated analysis of performance regression testing.

(2) We proposed a log based root cause analysis and change impact analysis approach for performance regressions.

(3) We proposed a novel distance based score for ranking suspicious methods.

(4) We conducted a case study to evaluate our proposed method and framework in terms of effectiveness and efficiency.

The rest of the paper is organized as follows: Section 2 introduces the background and motivation example. In Section 3, we present our framework for automated performance regression test and analysis, showing the approach we applied for root cause analysis and change impact analysis, respectively. Section 4 shows a case study based on realistic PaaS cloud services. Section 5 presents the related work. Finally, we give conclusions, discussion, and future work.

2. Background and Motivation Example

In this section, we first define the problem we solve in this work. Next, we use a realistic performance regression to show the challenges of analyzing potential issues and evaluating the potential impact of applying an improvement. Then, we introduce how our framework helps performance analysts solve the problems.

2.1. Problem Model. In this paper, we study root cause analysis and change impact analysis in terms of performance: Given an application layer, execution logs consist of a set of traces $E = \{t_1, t_2, \ldots, t_n \mid t_n \in T\}$, where T is the space of different traces. Each trace belongs to a certain type, for example, indicating the start of an event, detailed traces of method calls. A trace type is characterized by various attributes; let $A_{T_i} = \{a_1, a_2, \ldots, a_n\}$, where a_n is an attribute; for example, an event may have a timestamp, event name, and web service id.

In this work, given execution log E as input, we compute ranking score Score_m of methods for root cause analysis and estimated change impact set (ECIS) for methods.

2.2. Motivation Case. Our approach is motivated by a realistic requirement. A world top 500 enterprises' financial system (named FS in this paper) is migrated to run on its private PaaS platform. Due to confidentiality, we anonymize the name of system and relevant resources in this case. The web services of FS are built upon a customized web service specification language upon HTTP(s). Performance regression testing is continuously conducted to assure the performance of new releases against production environment with controlled load. Monitoring tools are not allowed to be deployed in production environment because of security policy. Customized tracing is used to track application performance. The execution traces can be used for root cause analysis and change impact analysis. It contains both the structural information and timing information of methods. However, it is difficult to analyze the execution traces, because the log generated by every round of performance regression may be up to 1 GB. Thus, an effective performance analyzer is required to help performance engineers in both root cause analysis and change impact analysis.

2.3. How Our Approach Facilitates Analysis. Traditionally, engineers look for root cause of performance regression by searching key words in logs. However, discovering root cause in a large log is nontrivial. What is more, they cannot predict a proposed improvement and side effects. Thus, manual inspection is error-prone, which is risky from the perspective of managers.

By employing our analysis framework, performance engineers can first figure out the root cause of the performance regression. Then, the performance engineers can apply change impact analysis for methods. The change impact analysis can provide a list of business transactions influenced by methods in terms of performance. Performance engineers can also use this to propose the test suite in the next round of performance regression testing. Our approach can improve the effectiveness and efficiency of performance analysis.

3. Approach

3.1. Framework. In this section, we introduce the framework of our automated performance analysis framework. As shown in Figure 1, our framework is divided into 2 parts: testing and analysis. Arrow 1 indicates the process of executing performance regression testing. Arrow 2 indicates the process of performance analysis, including root cause analysis and change impact analysis, which is the focus of this paper.

Performance regression testing process: the performance regression testing is conducted against the system under test deployed in the private PaaS cloud platform, where there is a central repository of applications and system logs. The performance regression testing can be automatically executed if required data is ready. The performance regression tests generate logs by load generator (located in the framework), system level performance counters, and application level logs in log server. This part is not the focus of this paper.

Performance analysis process: during the analysis process, we fetch the logs from the log server to perform performance analysis. We compute response time and throughput of transactions. The performance analysts can identify performance regressions against an earlier baseline. If any performance regression is found or further analysis is required, they can apply the root cause analysis and change impact analysis.

Figure 2 shows the key processes of our approach. There are mainly 6 key steps.

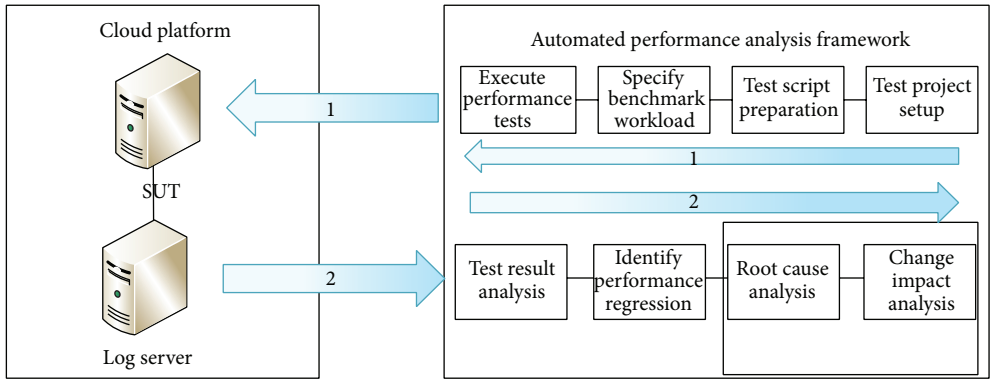

Figure 1: Automated performance analysis framework.

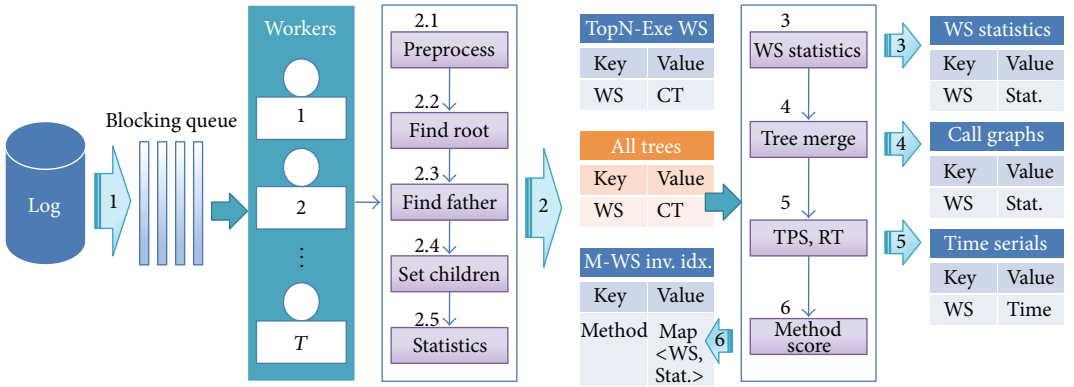

Figure 2: The procedure of root cause analysis and change impact analysis.

In step 1, a producer thread reads log files into memory and divides logs into multiple blocks. Each block contains several traces that composite an entire call graph of a web service. Each block is put into a blocking queue. We decouple the reading and processing by using the producer-consumer pattern. Our experiments show that this method can maximize the utilization of I/O.

In step 2, each consumer thread fetches a block to be processed. First, a thread will preprocess a block and scan for the start of a method execution. In this step, traces are interpreted with regular expressions. Useless descriptive traces will be filtered. If a trace is recognized as a method event, we will find the root if the root of the call tree has already been created. Next, we look for the father with the structural information and set the reference of father. For example, if a node is labeled as 1.1.2.1, its father's label is 1.1.2. We put the father's children to its children list. Finally, we compute the self-time and relative percentage to its father and root node. By traversing traces of a block, we can obtain a runtime call tree of a business transaction.

Besides constructing a call tree, we compute a Top-N most time-consuming web services map (not statistically, but the exactly executed ones) and preliminary inverted index mapping from methods to web services. We ignored the mapping between methods, since we consider the information is enough for change impact analysis of performance regression testing. At this point, we get three main data structures: (1) inverted index from method to web service which is just initialized for the relation without detailed statistics; (2) Top-N web services in terms of elapsed time; (3) call trees for all the web services called (each web service call corresponding to a call tree), which store most of the data transformed from raw data. The followed processes are all based on the call trees obtained in step 2.

From steps 3 to 6, all the call trees are traversed to compute some key statistics for root cause analysis and change impact analysis. In step 3, common statistics (e.g., max, min, avg., and std.) for web services is computed. In step 4, we merge the call trees under the same web service for root cause analysis. We will detail it in the next subsection. In step 5, we generate time serials graph in terms of response time and throughput based on a sample window. In step 6, we update the average time occupation into the inverted index.

In order to improve the efficiency of analysis, one consideration of our implement is the performance. Because the size of log is usually up to GB level, it will cost about half a minute for only reading from conventional hard disks. In order to reduce the computation, especially I/O expense, we merge the process of root cause analysis and change impact analysis in our framework, by which we can minimize I/O operations of processing a log. The further computation is all performed in memory.

3.2. Root Cause Analysis. Traditionally, performance analyst can apply profilers for code level root cause analysis of performance issues. Some profilers provide ranked self-time and self-CPU time for analysis. This method neither provides a top-down view like call tree nor considers other noises, like the relations between methods, for identifying root cause. State-of-the-art profilers provide merged call trees for all with method elapsed time and execution counts, which are useful for performance tuning of identifying hot spots. However, it still needs much manual inspection. Experiencing the pitfalls of existing methods, we want to develop a more efficient method by integrating the convenience of ranking methods and insights of call trees.

In realistic systems, different performance antipatterns can cause performance issues. The challenge of designing a single indicator is that the indicator should cover different situations. Consider a method to be a root cause; it can have different behaviors: (1) always perform poorly in all services; (2) perform differently in different services depending on input; and (3) perform poorly when network or some other resource becomes bottleneck, no matter.

A web service can have many call trees with different structures. First, it is easy to understand that a web service can be called by different users at different time many times. This will lead to many call trees in the logs. Next, users may input different parameters for the web service, which may lead to different call tree for conditional branches in program. What is more, a web service can have different runtime call tree because of caches. Thus, the performance variation of a web service may be driven by different inputs and hidden in some methods' implementation. For different web service, a method usually has different positions and different performance influence.

Considering the above situations, we conduct a 3-phase computation for all call trees and methods: (1) merge same call trees of each web service; (2) merge call trees of a web service to a merged call tree; and (3) calculate ranking score for methods.

Merge same call trees of each web service: in step 4, we compute a one-to-many mapping from web service to call trees. In order to make the call tree comparison fast, an identity string is generated for a call tree in step 2.2. A node's identity in a call tree is determined by both position and value. The root contains all the structural and value information with only a string. Then, the string will be transferred to a hash code which is used for comparing the equivalence with others in the merging step (step 4). As mentioned before, a web service may finally have several call trees.

Merge call trees of a web service to a merged call tree: if a web service has different call trees, we compute a merged call tree. The merged call tree contains all the possible branches. Each node, referring to a method, has the information of occupation percentage of a web service. The merged call tree provides a top-down view of performance influence of methods, which can help engineers conduct top-down analysis. By drilling down a web service, we can find out hot spots of the web service.

Calculate ranking score for methods: in step 6, we compute a global ranking score for all the methods for bottom-up analysis. A method with bigger score_m means it is more likely to be the problem and worth more attention. Method m can be called in many (supposed to be N) web services. We indicate a method as an N dimension vector $m = (r_{m,1}, r_{m,j}, \ldots, r_{m,n})$, where $r_{m,n}$ is the average self-time ratio (excluding the time of callees) of method m in the merged call tree for web service WS_n. We have $0 < r_{m,n} < 1$. We can consider method m has contribution in x dimensions if it has been called in x web services. Imaging m to be a point in an N dimensions space, then the Euclidean distance D_m from m to original point can be a combined metric for x web services. For example, for $m_1 = (0.9, 0.8, 0.7)$ and $m_2 = (0.3, 0.4, 0.7)$, $D_{m1} = 1.39$, $D_{m2} = 0.86$. Then m_1 has more performance impact globally. However, this cannot explain the following example when $m_1 = (0.9, 0.9, 0.9)$ and $m_2 = (0.31, 0.32, \ldots, 0.327)$, which mean m_1 is called for only 3 web services while m_2 is called by 27 web services. Their Euclidean distance is the same, but, for hot spot analysis, m_1 is more likely to be the root cause. Thus, we apply root mean square to eliminate the different dimension problem. The ranking score is defined as $R_m = \sqrt{({r_{m,1}}^2 + {r_{m,j}}^2 + \cdots + {r_{m,n}}^2)/X}$. However, we find the fluctuation is also a problem. For example, when $m_1 = (0.9, 0.2, 0.2)$ and $m_2 = (0.6, 0.5, 0.5)$, $R_{m1}(0.545) > R_{m2}(0.535)$. We consider that the big ratio in m_1 might be caused by a special workload, and m_2 is more likely to influence global performance. Thus, we introduce standard deviation to eliminate the problem. In this case, $\text{Std}_{m1} = 0.329$, $\text{Std}_{m2} = 0.047$. Then, the score is defined as $\text{Score}_m = R_m - f * \text{Std}_m$, where f is an adjustable parameter. Currently, we use $f = 1$ in our program. The setting of the parameter would be one of our future works.

3.3. Performance Change Impact Analysis. Traditional change impact analysis techniques compute the set of program elements that may be affected by the change (change impact set). Those techniques are static analysis based [6], dynamic execution based [7], or history based [8, 9]. Our proposed method is inspired by these works. The key difference is that we focus on the performance impact of the change, instead of only change impact set. We apply inverted index from methods to web services for storing the performance change impact set. In order to minimize the change impact set, we only keep the impacted web services without impact relation between methods. The performance change impact set computed can guide the priority performance tuning and test suite selection of performance regression testing.

The computation of the performance change impact set is performed in two steps. In step 2, when each method is traversed, we add a mapping relation between a method and a web service into the map. The process continues until all the method-WS relations have been inserted. In step 6, we first update the average time occupation of methods in web services. Then, we compute the score_m based on the formula we mentioned before. For method-WS relations, the ratio of execution time occupied is computed and visualized using comparable bar chart.

Performance analyst can use $score_m$ to fast get a more valuable improvement point. Then, for a particular method, the bar chart can be used to guide the performance analyst to find out the performance impact for each web service. In addition, we can find whether the hot spot is more likely caused by coding or outer inputs.

4. Empirical Results and Evaluation

Our approach is adopted to study the root cause of performance issues in realistic applications in an enterprise PaaS cloud. The applications use a logger to log the execution traces during performance regression tests in production environment. The logger can be switched off when the traffic is heavy. During several performance regression tests, the performance testers found that some web services have performance regressions. Our tool is used for root cause and performance change impact analysis.

Our dataset contains millions of traces sized in range from 100 Mb to about 1 GB. For example, we counted that a log with 600 Mb contained more than 1,260,000 lines of traces. The traces are logged by the performance logger and stored in the log server. It is nearly impossible to manually check such large log files. Our study and evaluation provide developers and performance engineers with two main benefits. First, our study and evaluation can uncover the root cause performance regressions and predict change impact in real world. Second, the study and evaluation show that our approach can facilitate performance analysis by reducing efforts of analyzing performance problems.

4.1. Evaluation of Root Cause Analysis. Performance engineers can manually inspect patterns in logs to determine whether those traces hint high-impact performance problems and need further actions towards performance tuning. To show the effectiveness of root cause analysis and how root cause analysis can improve the efficiency of manual inspection, we address the following research questions:

> RQ1: can our ranking score correctly pinpoint root cause of performance issues?
>
> RQ2: to what extent would our approach improve the efficiency of root cause analysis?

We apply the root cause analysis after a round of performance regression test which found several performance regressions. The log file is about 627 Mb with 1589765 lines of execution traces. There are 35906 web service calls of 6 distinct web services in the log. Because the web services can accept different combinations of parameters, thus there are many different requests for a web service. We choose this log for three reasons: (1) the selected log contains a set of typical web services which have performance regressions during a performance regression test, (2) the number of services executed in the selected log is relatively large to avoid data noise and biases, and (3) the performance engineers have manually inspected the log and identified the problems with the developers; thus we can validate our method against the realistic practice.

To address RQ1, we validate whether our ranking method of web services can effectively guide performance analyst to identify root cause of performance problems. We first validate whether $score_m$ can represent high performance impact of methods. In particular, we determine that a high score representing at least one execution has high performance impact. What is more, we check whether the manually identified root cause can be identified by the $score_m$.

For RQ2, we applied our framework on different size of logs to evaluate the performance of our framework. Then we compare the time against realistic time cost for the manual analysis by performance engineers.

4.1.1. RQ1: Effectiveness. We show the effectiveness of root cause analysis by presenting the ranking of computed $score_m$ and time distribution of problematic web services. $score_m$ can help performance engineers apply bottom-up analysis for locating the root cause. And the time distribution visualization of web services can aid engineers apply top-down method for analyzing web service performance.

Considering the space, we only show top 20 hot functions ranked by $score_m$ with other statistics in Table 1. We compare the results with manual inspection results. We checked method with top 30 (about 5% of 591 methods) $score_m$ to find out (1) whether the problems revealed by manual inspection can be revealed with our solution; (2) whether the high $score_m$ does represent a performance issue. Manual inspection applied combined thresholds (e.g., time percentage >70 and average time >1 s) to filter traces with help of some ad hoc scripts. Manual inspection totally revealed 5 identified problematic methods which are ranked 1, 4, 8, 12, and 23 in our ranking of 591 methods. For the top 30 $score_m$, we check the web services to verify whether the methods are performance issues by looking into the merged call tree visualized as Figure 3. The methods ranked 4, 10, 15, 16, 21, 23, 26, and 29 are identified as performance issues with both high processing time and time percentage occupied. Thus, the accuracy of top 30 is about 43%. The most false positive is caused by methods that occupy a high ratio of a web service while its execution time is low. It is easy to filter them out by checking the average execution time and maximum execution time. By manually filtering out the methods with relatively small execution times, our accuracy can reach 100%.

The pitfall of our ranking score cannot distinguish which method takes more time. We do not apply any factor to balance this in $score_m$ for two reasons: (1) for some web services, all the execution times of an individual method are small that a method with big $score_m$ is indeed the root cause of performance issue; (2) we can filter the methods with small execution time by checking another metric average execution time; the time needed for filtering false negative is trivial. In our framework, $score_m$ is a leading indicator but not the only indicator.

4.1.2. RQ2: Efficiency. Our framework is well tuned to maximize the utilization of CPU, memory, and I/O resources. A log with size of 600 Mb can be analyzed with our framework on desktop PC within 50 seconds. For small logs, it can return the analysis results in 5 seconds. We surveyed

Table 1: Top 20 hot functions ranked by impact score in the case study.

Functions	Impact score	Max time (ms)	Avg. time (ms)	Min time (ms)
com.∗.dataaccess.doa.AFCMCHGetTrlOOBWizardDataSetsDBDAO - executeSQL	0.944	4492.79	2860.96	1229.13
com.∗.msgHandler.idfHandler.AFCIDFSecurityMasterHandler - Do proccess	0.496	36.79	10.56	1.18
com.∗.msgHandler.idfHandler.AFCIDFLotLevelHoldingsHandler - Do proccess	0.488	11.29	3.84	1.24
com.∗.dataaccess.doa.AFCMCHAcctLedgerBalanceWSDAO - select	0.41	3136	1879.54	499.85
com.∗.msgHandler.idfHandler.AFCIDFPositionHandler - Do proccess	0.384	8.87	4.72	1
com.∗.msgHandler.AFCPingIdfMsgHandler - Do proccess	0.344	5.32	2.31	0.75
com.∗.msgHandler.idfHandler.AFCIDFIndexHandler - init IDF Param	0.294	0.74	0.60	0.31
com.∗.dataaccess.doa.AFCMCHAcctLedgerCapstockSummaryWSDAO - select	0.279	3382.25	1900.44	604.07
com.∗.msgHandler.idfHandler.AFCIDFSecurityMasterHandler - IDF Handler Process Total time	0.248	53.38	15.57	2.436
com.∗.dataaccess.util.AFCDBConnectionFactory - GenConnectionPool	0.214	6363.50	234.47	5.92
com.∗.msgHandler.idfHandler.AFCIDFIndexHandler - Close Listener	0.208	0.73	0.423	0.26
com.∗.dataaccess.doa.AFCMCHAcctFundExtRatioWSDAO - select	0.174	4139.08	2183.72	716.816
com.∗.msgHandler.AFCPingIdfMsgHandler - init IDF Param	0.17	0.96	0.68	0.52
com.∗.msgHandler.idfHandler.AFCIDFIndexHandler - IDF Handler Process Total time	0.169	3.23	2.03	0.98
com.∗.dataaccess.doa.AFCMCHAcctKeyFinStatsDBDAO - executeSQL	0.161	2719.07	220.54	8.49
Com∗.dataaccess.doa.AFCMCHAcctAllFundHierarchyTreeDBDAO - executeSQL	0.123	2569.33	425.44	11.87
Com.∗.msgHandler.idfHandler.AFCIDFLotLevelHoldingsHandler - init IDF Param	0.117	0.88	0.63	0.24
com.∗.msgHandler.idfHandler.AFCIDFPositionHandler - init IDF Param	0.112	0.72	0.69	0.66
com.∗.msgHandler.AFCPingIdfMsgHandler - IDF Handler Process Total time	0.102	8.48	3.99	1.79
com.∗.msgHandler.idfHandler.AFCIDFLotLevelHoldingsHandler - IDF Handler Process Total time	0.098	13.11	5.44	1.90

the performance engineers for their time cost for manual inspection. We found that most of the time was used for manually looking into the logs to adjust the thresholds to filter out as many traces as possible, by which they can merge and compute some statistics for the left traces to find the root causes. They also need to manually verify the root causes by searching in the original log. Finally, they generate a diagnosis report including evidences for the root causes. The total process involves much manual effort, which would take nearly one day. Our framework can help them to get the analysis result just in a few minutes. They only need half an hour to work out a final report. For the total analysis process, we saved nearly 90% of the time for performance engineers.

4.2. Results of Change Impact Analysis. Figure 4 shows an example of performance change impact analysis for the method ranked 10 (com.∗.dataaccess.util.AFCDBConnectionFactory - GenConnectionPool). The self-time ratio and children time ratio are computed for each web service calling the method. In this case, the method is a leaf in the call graph, and the children ratio is zero. From the graph, we can see that the method has different performance impact on 6 web services. The method occupies a large part of execution time near 70% in the top 2 web services. Thus, we can estimate that the improvement of the method may dramatically improve the performance of those 2 web services. The 6 web services should be regression tested in the next round of performance regression testing if we apply improvement for this method. For the web service impacted by the method, we can further look into the call graph to verify the performance change impact. Based on the feedback of performance engineers, our framework is useful for helping them figure our performance tuning and performance regression testing plan. It can greatly increase their confidence of prioritizing tuning tasks and estimate potential efforts of both tuning and regression testing.

5. Related Work

Performance regression testing is a subtype of performance testing. There is plenty of work on performance testing. This work can be divided according to performance testing life cycle. The performance testing life cycle can generally

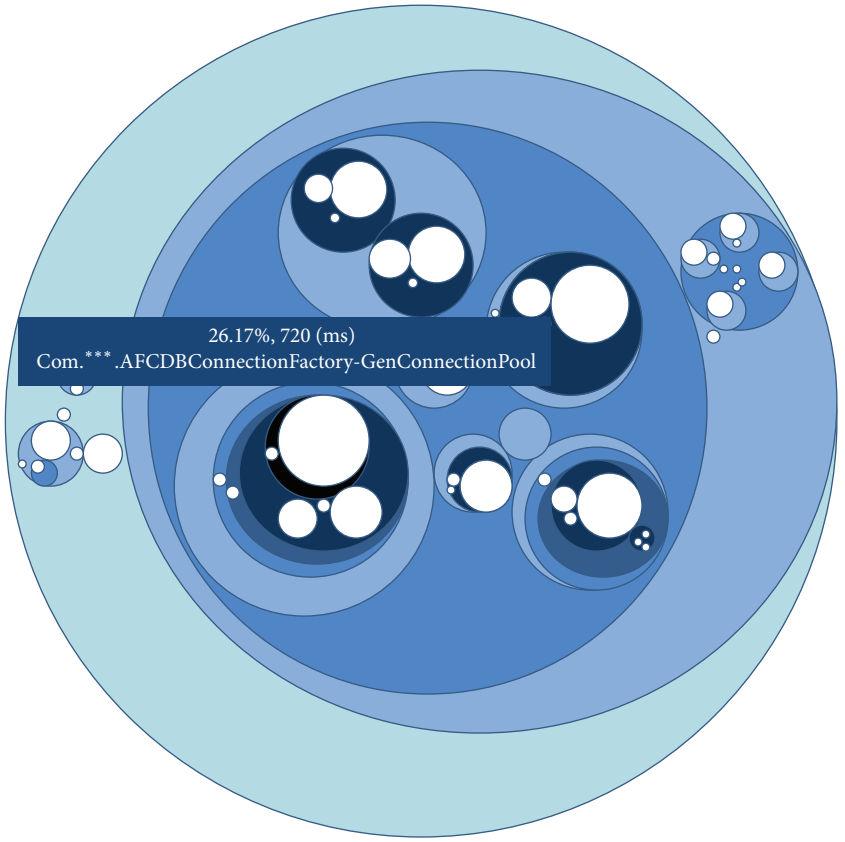

Figure 3: Zoom circle visualization for hot spots of a web service.

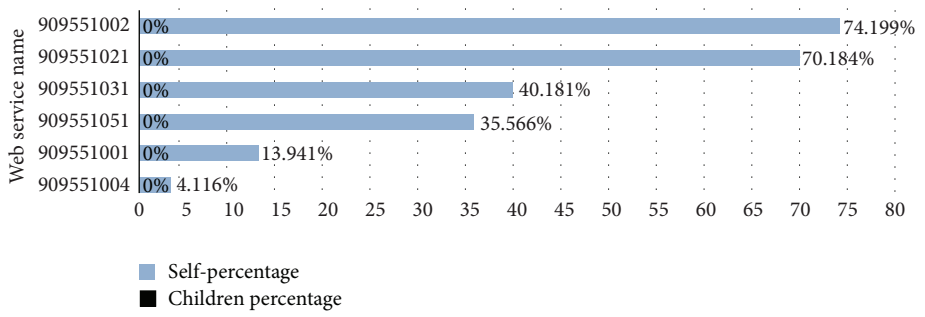

Figure 4: Performance impact set for method com.∗.dataaccess.util.AFCDBConnectionFactory - GenConnectionPool.

be 4 steps: preparation, performance testing, performance analysis, and performance tuning and validation. In preparation, workload characterization is conducted to capture the system workload for further simulation. In testing phase, performance testing and monitoring tools are used to test and monitor the system for obtaining required indicators. In the analysis phase, the obtained test data is analyzed to generate test report and identify potential performance problems. In the tuning phase, performance issues are tuned and validated. In this section, we introduce the related work that is most relevant to the theme of this work.

Performance testing: the research on performance testing aims to increase the efficiency and effectiveness. Some works proposed to use model based performance testing to generate appropriate workload and increase the automation level [5, 10–13]. Some proposed techniques for test suite or load generation to increase test effectiveness. Grechanik et al. offered a feedback-directed learning testing system to learn rules from execution traces for selecting test input data automatically [14]. Zhang et al. [15] applied symbolic execution to generate load test suites that expose program's diverse resource consumption behaviors. Huang et al. [16] used static analysis techniques for performance risk analysis to improve performance regression testing efficiency via testing target prioritization. There are some commonly used performance testing tools, both commercial and open source. LoadRunner of HP [17] and Rational Performance Tester of IBM are dominant in traditional load testing market. Apache JMeter

[18] and The Grinder [19] are two relatively popular open source frameworks. There are also cloud-based performance testing frameworks, such as SOASTA's CloudTest [20] and LoadStorm [17]. There are also some frameworks proposed in literature for fractal based [21], REST web application [22], and better automation [23].

Performance regression testing: there are some case studies and research on software performance regression testing. Chen et al. presented their performance regression testing methodology to keep Linux kernel performance [24]. Kalibera et al. showed their performance regression testing experience of Momo project [25]. They studied the impact of system initialization problem and its solution. Huang et al. applied static analysis technique to build risk model for estimating performance risks in a new release [16]. The estimated risks are used to prioritize performance regression tests to reveal potential performance regressions and save cost.

Performance analysis: Jiang et al. [4] presented an approach that automatically analyzes the execution logs of a load test for performance problems. Malik et al. [26] showed their methodology of automating the process of comparing the important performance counters to identify performance gains and losses in load tests. Nguyen [27] proposed to use control charts and statistical process control technique to help performance analyst identify root cause of performance regressions. Recently, the SAIL of Queen's University has published many papers on performance analysis of performance testing (or load testing).

Change impact analysis: there is fruitful work on software change impact analysis techniques to compute the estimated impact set in terms of functionality [28]. These techniques can be divided into three categories: static analysis based [29, 30], dynamic analysis based [6, 7, 31–34], and mining software repositories based [8, 35]. Li et al. conducted a comprehensive survey on software change impact analysis [36].

Our proposed method is inspired by these works. The key difference is that we focus on the performance impact of change, instead of traditional impact set. Our proposed metric can guide performance analysts prioritize performance tuning activities and evaluate potential change impact. The change impact set computed can also be used to guide the next round of performance regression testing.

6. Conclusions and Future Work

In this paper, we present a distance based root cause and change impact analysis framework to help performance analyst efficiently identify root cause of performance regressions. We provide useful information for conducting both top-down and bottom-up analysis methods. The score_m and change impact set can be used to evaluate the change impact from both functional and performance views. The generated call trees and visualization can help performance analyst quickly find root cause of performance issues. We evaluate our approach and implementation based on several logs generated by realistic web services in an enterprise PaaS cloud. Our empirical experience is that our method can efficiently help performance analyst to shorten the diagnosis process. Our further work will focus on the improvement of this tool. The computation of score_m can be refined based on more experiments and theoretic proof. We would also like to make our framework run faster by employing distributed computing like MapReduce. Thus, our tool can be accessed by concurrent users for processing bigger logs.

Conflict of Interests

The authors declared that they have no conflict of interests to this work.

References

[1] H. Sarojadevi, "Performance testing: methodologies and tools," *Journal of Information Engineering and Applications*, vol. 1, no. 5, pp. 5–13, 2011.

[2] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: a call graph execution profiler," *ACM SIGPLAN Notices*, vol. 17, no. 6, pp. 120–126, 1982.

[3] A. Sarimbekov, A. Sewe, W. Binder, P. Moret, and M. Mezini, "JP2: call-site aware calling context profiling for the Java Virtual Machine," *Science of Computer Programming*, vol. 79, pp. 146–157, 2014.

[4] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora, "Automated performance analysis of load tests," in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM '09)*, pp. 125–134, September 2009.

[5] J. Zhou, B. Zhou, and S. Li, "Automated model-based performance testing for PaaS cloud services," in *Proceedings of the IEEE 38th International Computer Software and Applications Conference Workshops (COMPSACW '14)*, pp. 644–649, Vasteras, Sweden, July 2014.

[6] M. Acharya and B. Robinson, "Practical change impact analysis based on static program slicing for industrial software systems," in *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*, pp. 746–755, May 2011.

[7] J. Law and G. Rothermel, "Whole program path-based dynamic impact analysis," in *Proceedings of the 25th International Conference on Software Engineering*, pp. 308–318, May 2003.

[8] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, "Mining version histories to guide software changes," *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 429–445, 2005.

[9] D. M. German, A. E. Hassan, and G. Robles, "Change impact graphs: determining the impact of prior codechanges," *Information and Software Technology*, vol. 51, no. 10, pp. 1394–1408, 2009.

[10] X. Wang, B. Zhou, and W. Li, "Model-based load testing of web applications," *Journal of the Chinese Institute of Engineers*, vol. 36, no. 1, pp. 74–86, 2013.

[11] F. Abbors, T. Ahmad, D. Truscan, and I. Porres, "Model-based performance testing in the cloud using the MBPeT tool," in *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering (ICPE '13)*, pp. 423–424, April 2013.

[12] A. Bahga and V. K. Madisetti, "Synthetic workload generation for cloud computing applications," *Journal of Software Engineering and Applications*, vol. 4, no. 7, pp. 396–410, 2011.

[13] C. Barna, M. Litoiu, and H. Ghanbari, "Model-based performance testing (NIER track)," in *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*, pp. 872–875, May 2011.

[14] M. Grechanik, C. Fu, and Q. Xie, "Automatically finding performance problems with feedback-directed learning software testing," in *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*, pp. 156–166, June 2012.

[15] P. Zhang, S. Elbaum, and M. B. Dwyer, "Automatic generation of load tests," in *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE '11)*, pp. 43–52, November 2011.

[16] P. Huang, X. Ma, D. Shen, and Y. Zhou, "Performance regression testing target prioritization via performance risk analysis," in *Proceedings of the 36th International Conference on Software Engineering (ICSE '14)*, pp. 60–71, May 2014.

[17] Loadrunner, http://www8.hp.com/cn/zh/software-solutions/load-runner-load-testing/index.html.

[18] JMeter, https://jmeter.apache.org/.

[19] "Grinder," http://grinder.sourceforge.net/.

[20] SOASTA, http://soasta.com/.

[21] B. Dillenseger, "CLIF, a framework based on Fractal for flexible, distributed load testing," *Annals of Telecommunications*, vol. 64, no. 1-2, pp. 101–120, 2008.

[22] C. H. Kao, C. C. Lin, and J.-N. Chen, "Performance testing framework for REST-based web applications," in *Proceedings of the 13th International Conference on Quality Software (QSIC '13)*, pp. 349–354, July 2013.

[23] J. Gao and Y. Lan, "Automatic test task allocation in agent-based distributed automated testing framework," in *Proceedings of the International Conference on Computational Intelligence and Software Engineering (CiSE '09)*, pp. 1–5, December 2009.

[24] T. Chen, L. I. Ananiev, and A. V. Tikhonov, "Keeping kernel performance from regressions," *OSL*, vol. 1, pp. 93–102, 2007.

[25] T. Kalibera, L. Bulej, and P. Tuma, "Automated detection of performance regressions: the Mono experience," in *Proceedings of the 13th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS '05)*, pp. 183–190, September 2005.

[26] H. Malik, Z. M. Jiang, B. Adams, A. E. Hassan, P. Flora, and G. Hamann, "Automatic comparison of load tests to support the performance analysis of large enterprise systems," in *Proceedings of the 14th European Conference on Software Maintenance and Reengineering (CSMR '10)*, pp. 222–231, March 2010.

[27] T. H. D. Nguyen, "Using control charts for detecting and understanding performance regressions in large software," in *Proceedings of the 5th IEEE International Conference on Software Testing, Verification and Validation (ICST '12)*, pp. 491–494, April 2012.

[28] N. Kama, "Change impact analysis for the software development phase: state-of-the-art," *International Journal of Software Engineering and Its Applications*, vol. 7, no. 2, pp. 235–244, 2013.

[29] R. S. Arnold and S. A. Bohner, "Impact analysis-towards a framework for comparison," in *Proceedings of the Conference on Software Maintenance*, pp. 292–301, Montreal, Canada, September 1993.

[30] A. Orso, T. Apiwattanapong, and M. J. Harrold, "Leveraging field data for impact analysis and regression testing," *ACM SIGSOFT Software Engineering Notes*, vol. 28, no. 5, pp. 128–137, 2003.

[31] A. Orso, T. Apiwattanapong, J. Law, G. Rothermel, and M. J. Harrold, "An empirical comparison of dynamic impact analysis algorithms," in *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*, pp. 491–500, May 2004.

[32] H. Cai and R. Santelices, "Diver: precise dynamic impact analysis using dependence-based trace pruning," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE '14)*, vol. 33, pp. 343–348, ACM, Vasteras, Sweden, September 2014.

[33] T. Apiwattanapong, A. Orso, and M. J. Harrold, "Efficient and precise dynamic impact analysis using execute-after sequences," in *Proceedings of the 27th International Conference on Software Engineering (ICSE '05)*, pp. 432–441, May 2005.

[34] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley, "Chianti: a tool for change impact analysis of java programs categories and subject descriptors," *ACM SIGPLAN Notices*, vol. 39, no. 10, pp. 432–448, 2004.

[35] R. J. Turver and M. Munro, "An early impact analysis technique for software maintenance," *Journal of Software Maintenance: Research and Practice*, vol. 6, no. 1, pp. 35–52, 1994.

[36] B. Li, X. Sun, H. Leung, and S. Zhang, "A survey of code-based change impact analysis techniques," *Software Testing Verification and Reliability*, vol. 23, no. 8, pp. 613–646, 2013.