International Journal of Science and Research (IJSR)
ISSN (Online): 2319-7064
Index Copernicus Value (2015): 78.96 | Impact Factor (2015): 6.391

A Survey Based on Performance-Bug Detection

S. Maheswari[1], Dr. K. Chitra[2]

[1]Research Scholar, Dept of Computer Science, Bharathiar University, Coimbatore, TN, India

[2]Asst. Professor Dept of Computer Science, Govt Arts College, Melur, Madurai Dt, TN, India

Abstract: ***Software testing is the process of executing a program or system to discover the intent of bugs. Software testing is an activity designed to assess the properties or capabilities of a program or system and to determine the results that meet its requirements. Software performance is critical to how the user perceives the quality of the software product. Performance bugs are programming bugs that cause significant performance degradation, resulting in poor user experience and low system performance bugs. Designing effective techniques for handling bugs in yield design require a thorough understanding of how bugs are detected, reported and fixed. In this paper, to study how performance bugs, reported by developers to discover and develop fixation, and the results compared with non performance bugs. This paper conducts a detailed study of a sample of real-world performance bugs randomly from three sets of representative programs (Chrome, Mozilla and Apache). The results of this study serve as a guide for future work to prevent, expose, discover and correct performance bugs. First, found that there is little evidence that bug fixes are more likely to introduce new functional bugs to correct for erratic irregularities, which means that developers may not need more attention to correcting the performance of the bugs. Second, although fixing performance bugs is about as bug-prone as fixing nonperformance bugs, fixing performance bugs is more difficult than fixing non-performance bugs, indicating that developers need better tool support for fixing performance bugs and testing performance bug patches. Third, unlike many insects, a large percentage of performance bugs are detected by reasoning code, not by the user observing the negative effects of bugs or via profiles. The results show that the techniques that help developers need to test performance, better test beds and better profiling techniques to find performance bugs.***

Keywords: Automation Tool, Performances Bug, Testing.

1. Introduction

Software testing is the main activity of evaluating and executing software with a view to find out bugs. It is the process where the system requirements and system components are exercised and evaluated manually or by using automation tools to find out whether the system is satisfying the specified requirements and the differences between expected and actual results are determined.

Software performance is important to the overall success of a software project. Performance bugs called programming bugs that create significant performance degradation hurt software performance and quality. They lead to poor user experience, degrade application responsiveness, lower system throughput, and waste computational resources. Even expert programmers introduce performance bugs, which have already caused serious problems. Well tested commercial products such as Internet Explorer, Microsoft SQL Server, and Visual Studio are also affected by performance bugs.

Therefore, both industry and the research community have spent great effort on addressing performance bugs. For example, many projects have performance tests, bug tracking systems have special labels for performance bugs, and operating systems such as Windows 7 provide built-in support for tracking operating system performance. In addition, many techniques are proposed recently to detect various types of performance bugs.

To understand the effectiveness of these techniques and design new effective techniques for addressing performance bugs requires a deep understanding of performance bugs. A few recent papers study various aspects of performance bugs, such as the root causes, bug types, and bug sources, which provide guidance and inspiration for researchers and practitioners. However, several research questions have not been studied at all or in depth, and answers to these questions can guide the design of techniques and tools for addressing performance bugs in the following ways:

Based on maxims such as "premature optimization is the root of all evil", it is widely believed that performance bugs *greatly differ* from non-performance bugs, and that patching performance bugs carries a much greater risk of introducing new functional bugs. A natural question to ask is compared to fixing non-performance bugs, whether fixing performance bugs is indeed more likely to introduce new functional bugs. If fixing performance bugs is not more bug prone than fixing non-performance bugs, then developers may not need to be over-concerned about fixing performance.

Different from most non-performance bugs, whose unexpected behaviors are clearly defined, e.g., crashes, the definition of performance bugs is vague, e.g., how slow is qualified as a performance bug. Therefore, are performance bugs more difficult to fix than non-performance bugs? For example, are performance bug patches bigger? Do performance bugs take longer to fix? Do more developers and users discuss how to fix a performance bug in a bug report? Many techniques are proposed to help developers fix bugs, typically with a focus on nonperformance bugs. If performance bugs are more difficult to fix, we may need more support to help developers fix them.

Since the definitions of expected and unexpected behaviors for performance bugs are vague compared to those of nonperformance bugs, are performance bugs less likely to be discovered through the observation of unexpected behaviors

Licensed Under Creative Commons Attribution CC BY

than non-performance bugs? Are performance bugs discovered dominantly through profiling because many profiling tools are available and used? If performance bugs are less likely to be discovered through the observation of unexpected behaviors compared to non-performance bugs, or performance bugs are rarely discovered through profiling, then it is important for researchers and tool builders to understand the reasons behind the limited utilization of these techniques and address the relevant issues. If developers resort to other approaches to discover performance bugs, we may want to provide more support for those approaches to help developers detect performance bugs.

To answer these and related questions, we conduct a comprehensive study to compare performance and non-performance bugs regarding how they are discovered, reported, and fixed.

The manual effort needed to study more performance bugs is an inherent limitation of our and any similar study. Nonetheless, the lessons learned from comparing these bugs should provide a good initial comparison between performance and nonperformance bugs on discovering, reporting, and fixing them.

2. Motivation

Slow and inefficient software can easily prevent users from causing economic losses. While researchers spent decades improving software performance transparently, bugs continue to reduce the performance of a wide range of high-performance computing resources and waste in this area. At the same time, the preliminary performance of the current support for the fight against bugs is due to the poor understanding of the actual performance of the bug.

We refer to performance bugs, and where changing relatively simple source code can significantly speed software flaws while preserving functionality. These shortcomings cannot be done by the state, resulting in inconvenience to the end-user compiler being optimized.

There is a performance bugs in widely released software. For example, Mozilla Developer has set a monthly user-reported 5-60 performance bug for the past 10 years. Because almost nothing to do to help developers avoid bugs with the performance of the wrong kind of popularity is unavoidable. In addition, the performance test is mainly based on the black box test invalid and design manual entry, so that most performance bugs escaped.

Performance bugs lead to performance degradation, increased latency, and wasted resources in this area. In the past, they have caused several well-known failures, resulting in hundreds of millions of software projects were abandoned. To make matters worse, performance problems are expensive to diagnose because its symptoms do not stop. Software companies may need to work hard for a few months to find a pair of bugs that cause hundreds of milliseconds to delay performance in their service's 99th percentile latency period.

The following trends will lead to erroneous performance issues that are most critical in the future:

Hardware: Over the years, Moore's Law states that hardware will make software end time fast without software development work. In the era of multi-core, when it is impossible for each core to be faster, the bug is particularly detrimental to performance.

Software: The ever-complex changes in software systems and workloads provide new opportunities for new challenges and challenges in waste performance and diagnostics.

Energy Efficiency: Increased energy costs provide powerful economic parameters to avoid performance bugs. When a person is willing to sacrifice the quality of service, reduce energy consumption, ignoring the performance of the bug is inexcusable. For example, to correct a double run-time bug, you may halve the carbon footprint of the purchase and operate the computer.

You may return a bug that has not been reported as frequently as possible by a functional bug, and they will not cause a malfunction detention bug. However, given the initial support for bug-fighting performance, it is now time to pay more attention to them as we enter a new world of computational resource constraints.

3. Characteristics of performance Bugs

Many empirical studies have carried out the traditional mistakes that result in incorrect functional software, called functional bugs. These studies have successfully guided the design of functional software testing, functional fault detection and fault diagnosis.

Main Characteristics are,
Bug Avoidance. Two-thirds of the bugs studied were developed by misunderstandings about the workload or performance characteristics of API developers. More than a quarter of the bug from previous correct code, because of the workload or changing the API. In order to avoid performance bugs, developers need to annotate system performance-oriented and change impact analysis.

Performance Testing. Almost half of the research insects are required with special features and large scale inputs to embody. The combined use of the input functional tests takes into account the large scale significant improvements in the state of the art performance test generation scheme.

Bug Detection. Recent work has demonstrated the potential for bug detection performance. Our research has found common root causes with erroneous real-world performance structural patterns that can help improve coverage and bug detection performance accuracy.

Bug Fixing and Detection. Almost half of the inspection bug fixes include reusable efficiency rules that can help detect and correct performance bugs.

Licensed Under Creative Commons Attribution CC BY

Comparison with functional bugs. Performance bugs tend to be hidden much longer in software functional bugs. Unlike functional bugs, performance bugs cannot be modeled as rare events, since they are not small parts that can be activated using almost any input.

4. Related Work

Adrian Nistor et al. In order to improve performance and detect performance problems, several techniques identify slow code runs that expand or increase execution time. Unlike all these technologies, CARAMEL enables novel design decisions to focus on performance bugs with both simple and non-intrusive hardware. In particular, CARAMEL detects a performance bug with a CondBreak file. These bugs are not included in the previous work. For automatic bug fixes, several recent techniques have been proposed to automatically fix bugs. It uses genetic programming, and LASE uses edits that are similar to previous edits. Other techniques use methods such as SMT, semantic analysis, software contracts, developer input, and others to fix bugs. Unlike these technologies, CARAMEL automatically fixes performance bugs. In addition, with its unique attributes of detected bugs, CARAMEL successfully solved 149 of the 150 bugs. CARAMEL detects a performance bug with the CondBreak file: When a condition becomes true during a loop execution, it simply breaks through the loop. CARAMEL in real-world applications, including 11 popular Java applications (Ant, Groovy, JMeter, Log4J, Lucene, PDFBox, Sling, Solr, Struts, Tika and Tomcat) and 4 widely used C / C ++ applications (Chromium, GCC, Mozilla, and MySQL). CARAMEL found 61 new performance bugs in Java applications and 89 new performance bugs in C / C ++ applications. In these bugs, the developer has defined 51 performance bugs and 65 performance bugs in the C / C ++ application. CARAMEL has made a promising first step in detecting performance bugs with non-intrusive software [1].

Qi Luo et al. The performance tests for enterprise applications are manual, labor-intensive, expensive, and not particularly effective. Several methods are proposed to improve the efficiency of performance testing. For example, the function and parameter values of the operating profile schema have been introduced to test the probability of occurrence of a distribution of the most commonly used operations. Rule-based techniques are effective to identify performance bottlenecks by identifying problematic patterns of source code because of misinterpretation or invocation of sequence problems with API calls. However, these techniques always work for a specific type of performance bottleneck and are not widely used in industry. In practice, one of the primary methods of performance is intuitive testing, which is the tester's intuition and experience in the exercise of AUT, to guess possible bugs. Intuitive testing was first introduced in 1970 to take advantage of the expertise of test engineers with a focus on prone to bugs and related system functions without the need to write test specifications for time-consuming methods. As a result, the pre-investment and the indirect costs of the process can be reduced. When you run many different test cases and observe the behavior of the application, the tester intuitively perceives that some test case may have performance bottlenecks exposed in the future. However, one of the main risks of intuitive evidence is the loss of key people (ie, key testers). When they leave the company's knowledge and experience of test engineers are gone. Culturing new tests takes time and is expensive. Therefore, it is necessary to automatically reveal the performance bottleneck with distilled test cases to avoid wasting time and money in the properties. These attributes are automatically extracted to describe how the performance of the application will affect these rules for the secondary goals of our methods [2].

Oswaldo Olivo et al. Some recent projects use analysis software to automatically detect performance bugs. Some of them detect the use of useless temporary objects that are more focused on the use of invalid or incorrect data sets and others who use dynamic analysis to identify memory-expensive computations. The instrument uses a dynamic detection to determine the "repeatable" calculation by monitoring the repetitive access to the memory, in part a similar pattern. The work is based on the observation that the following set of duplicate paths may be an erroneous performance. The method is purely static, so it has no run-time overhead and does not require the programmer to provide a performance test. The tool analyzes PerfChecker static Android applications to determine common performance bugs. Unlike the clear, punctured-check detection and delay fulfillment obligations of the graphical user interface, leakage and swollen memory failures. The Perspective tool helps users diagnose performance problems related to configuration settings. X-ray uses a technique called comprehensive performance, which combines the dynamic analysis of information flow performance overhead. CLARITY different, X-ray dynamic analysis, and focus on the user's bug, rather than the developer's performance problems. Trace analysis is a technique for identifying the cause of performance anomalies. For example, the Trace Analyzer tool builds performance traces that can capture different performance times during program execution. Another approach involves impact analysis and tracing causal relationships to discover patterns related to performance issues. These techniques can be revealed in a variety of performance anomalies, but are not fully automated [3].

Sebastiano Panichella et al. Lack of evidence, and proposed a technique based on static down-regulation to generate code annotations describing faults and their causes. To generate a human-readable document that throws an unexpected exception. However, these methods need to test failed or thrown Java unexpected exception. This does not occur when the auto-generated test case, as an automatically generated assertion reflects the actual behavior of the class. Therefore, if the current behavior fails, the resulting assertions do not fail because they reflect incorrect behavior. Comprehension is also very much related to the size of the trial and the number of assertions. For these reasons, prior work on automated testing focused on (i) reducing the number of generated tests to minimize post-processing, and (ii) reducing the number of assertions using mutant analysis or split testing multiple claims. In order to improve the

Licensed Under Creative Commons Attribution CC BY

generated code, a read-test unit based on the use of a particular domain model is constructed based on human judgment of the readability of the post-evidence processing technique. Use the language model to generate a more readable input string. It shows that abstracts are an important complement to and enhance the readability of automatically generated component test cases [4].

Mario Linars et al. For the bug performance of automatic detection and other works of the design approach to identify significant performance bugs (ie, the delay can be perceived by the user) of the inspection rules, investigate 109 bugs, automatic identification performance bug five systems. He proposes to use performance counters and perform log events to link memory leaks to specific intervals of memory performance to diagnose methods. There has also been support for performance bottlenecks in in-situ visual propositions in understanding Java applications. The survey found that fewer bugs were detected in the desktop application using the configuration file. They also found that many performance bugs are based on code reasoning rather than direct observation of most of the recognition. Our results emphasize that, in the case of Android applications, people rely heavily on user reviews, manual execution and direct observation [5].

Xue Han et al. There has been a recent performance testing, debugging, fixing and preventing erroneous work. For example, a loop is determined that computes a repetitive pattern with memory access. The stack trace of the lightning protection call stack is found with the high performance impact call sequence to generate test case actions. Select the test case performance test. While the above techniques are encouraging and effective, they assume the default settings and do not take into account performance bugs due to configuration. Use common learning techniques to detect regressions due to specific changes in the environment. His work has focused on specific system configurations when we delve into a variety of configurations. From the performance modeling point of view, there has been a lot of work in building performance models for different purposes, such as using learning methods and influencing performance profiles, creating performance model profiles and performance modeling using static and dynamic program analysis technology. All of these techniques provide a good idea of the factors involved in the performance model. However, our study further examines how bugs are configured with performance and therefore complementary [6].

5. Terminologies of Performance Bug-Detection

The NaiveBayes categorized by software bugs showed an average accuracy of several data sets of 83.47. We set MC1, PC2 and PC5 data, which precision results are more than 95%, the performance is very good. The PC3 with the worst performance of the data set used, with an accuracy of less than 50% can be seen. MLP (Multilayer Perceptron) also performed well for MC1 and PC2 and achieved overall accuracy of 89.14% on several data sets. SVM (Support Vector Machine) and bagging performed very well, compared to machine learning and other methods, and obtained about 89% of the overall accuracy. Adoboost gets 88.59 accuracy, bagging gets around 88.47 to achieve accuracy of 89386, decision trees, random forests get 89.08, get 88.33 in J48 and unsupervised learning KNN (K Nearest Neughbor) 71.99 RBF To the basis function to 87.29 K-means. Compared to machine learning methods such as MLP, the SVM and bagging performance of all the datasets selected is still good. The lower accuracy is achieved by the KNN method.

The best MAE (Maekawa's) is implemented by the SVM method on multiple data sets 0.10 and 0.00 to obtain all data for the MAE PC2. MAE is the worst KNN method for 0.27. K-means, MLP, random forest and J48 also increased around MAE 0.14. In the case of this measure, F is greater and better.

This higher F gain is achieved by approximately 0.94 SVM and bagging method. The worst-case F obtained by the KNN method is in multiple data sets 0.82. Identify software defects early in the software lifecycle to help software quality assurance and direct measures to improve process management software. The cash forecast bug depends entirely on the good forecasting model. This study includes different machine learning methods that can be used to predict against bugs. Software for analyzing the performance of different algorithms in multiple datasets. Sky SVM, MLP and bagging techniques performed well in the database. Appropriate methods for selecting experts in the field of prediction bug must consider several factors, such as the data set, the problem domain, the uncertainty of the data set, or the nature of the project. You can combine multiple technologies with more accurate results.

5.1 Performance indicators

In this study, performance metrics such as accuracy, absolute mean bug, and measurement were used based on the accuracy and memory. The precision can be defined as the correct identification bug divided by the total number of bugs, calculated by the following formula:

Accuracy = (TP TN) / (TP TN FP FN)

Accuracy (%) = (Software Bug Correct Classification / Total Software Bug) * 100

Accuracy is a measure of the correction and is the relationship between the bug and the actual number of software bugs that the software correctly categorizes the categories assigned to it. It is calculated by the following formula:

Precision = TP /(TP+FP)

Recall is the correct classification of software bugs and falls within the scope of their relationship between software defects. It represents the machine learning method, the ability to seek extension, and is calculated by the following equation.

Recall = TP / (TP + FN)

F-measure is a combined measure of recall and precision, and is calculated by using the following equation.

F = (2 * precision * recall) / (Precision + recall)

Licensed Under Creative Commons Attribution CC BY

6. Case Studies

This section describes the faulty design of our study. For a particular project, we first extract the necessary data for your warehouse bugs. Next, we determine the random sampling of 100 performance bugs and 100 non performance bug reports. We then study the sampling bug to determine the different sizes and can be used to compare the bug performance of the fusing bug and the sub-size in the non-executing process. Finally, we proceeded based on these analyzes of our sub-dimensions. This section describes each of these steps.

6.1. Choosing Data from System

In our case study, we studied the web browsers Mozilla Firefox and Google Chrome, as these are the two most popular web browsers for public data tracking system bugs. We specialize in web browsers, and performance is one of the major software quality requirements for Web browsers that support multiple platforms and system environments. For this reason, bug performance is reported in both systems, and the problem tracking system in both projects is explicitly marked.

Firefox, we first came to identify the relevant web browser Firefox bug because Mozilla bug tracking system manages several projects. It contains 567595 bug reports for merging all the different components of Mozilla Firefox, SeaMonkey and Thunderbird Mozilla.

For Chrome Web Browsers, our data is provided from the dump question tracking chrome challenge 2011 MSR Extractor.

6.2. Classification of Bug Types

In Bugzilla (Mozilla Firefox's problem tracking system), the "Keywords" field is marked with "perf" for performance bugs. In the Google Chrome issue tracker, for performance-related bugs, the "Label" field is labeled "Performance." However, in these two projects, this mark is not mandatory, we found that many performance bugs, there is no such mark. Therefore, in order to identify performance bugs, we had to use heuristics. We look for the keywords "perf", "slow", and "hang" in the bug report "Title" and "Keyword" fields. Although the keyword 'perf' gave us a "performance" bug report in its header, we found that there are many bug reports that contain "perfect", "execute" or "execute" and nothing related to performance issues. We have to use regular expressions to automatically exclude these keywords from the list.

6.3. Identification of bug report and comment dimensions

Before we can compare the performance of the bug and did not perform, we must first establish the criteria for comparison. The ability to record the repository in different areas and bug logs is a good start, but our research requires knowledge of the testers and the way developers work together and think about bugs (none). This knowledge is part of the natural language content of the discussion and misinterpretation, but we do not know of any classifications or other work to analyze the qualitative data for bug reporting. Therefore, we conducted a sampling of the data to determine the classification of this manual study.

In order to determine if a particular system has a statistically significant higher percentage of bug rate performance associated with a sub-size (e.g., lock) than an unrelated bug, greater than the sub-size, a & quot; performance joint confidence interval & quot; or & quot; . This measure is often used to compare two separate samples. If the bug-rate performance, rather than the difference in the performance of the sub-size than the comparison calculated bug, the difference is considered statistically significant.

7. Performance Tool Studies

Httperf: Httperf is a high-performance testing tool for measuring and analyzing the performance of any Web service and Web application.

NeoLoad: NeoLoad is a load and performance testing software designed to improve the quality of Web and mobile applications by actually simulating users and analyzing server behavior.

QTEST: a network test tool loaded complete and accurate analysis of applications. It supports all Windows platforms. The original user interface (UI) is easy to use and understand, and is used to host on-demand or OnPremise applications. It is compatible with all Windows platforms.

Open STA: OpenSTA refers to the open system test architecture. OpenSTA is a distributed test structure that allows you to create and run performance tests to evaluate the network application environment (WAE) and production system.

Load Storm: The LoadStorm test tool is the least expensive performance and available load. In this tool, we are going to create our own test plan, test criteria and test program options. Through this tool, you can end all the costly performance testing tools.

Load Impact: LoadImpact is a load testing tool that is primarily used in cloud-based services. This also helps to optimize the site and improve the performance of any Web application.

QEngine (ManageEngine): QEngine (ManageEngine) is a tool, the most common, easy-to-use performance test that can automatically test their Web applications for and load.

Load UI: Loading UI is another kind of load testing software and open source is used to measure the performance of Web applications. This tool works well with the soap UI integration of functional testing tools.

Load Runner: HP is a product that can be used for performance testing tools. This allows you to purchase HP software from HP Products.

Licensed Under Creative Commons Attribution CC BY

8. Conclusion

The bug performance presented in this article has been applied in real-world applications. This paper reports our findings and finds some unique performance characteristics in the application of the results of the bug. It also identifies common patterns of bugs that can support bug detection, performance testing and debugging related research. The study covers the functional, broad-spectrum and future performance of the study provides guidance - prevention errors, performance testing, and fault detection and so on. Under the guidance of this study, we explored the performance implications of the bug detection performance based on the rule of implied efficiency through patches, and found a number of previously unknown performance problems. This work has only the starting point, understanding and operational performance bugs. We want to deepen our understanding of performance bugs, and pay more attention to performance bugs.

References

[1] Adrian Nistor, Po-Chun Chang, Cosmin Radoi, Shan Lu," CARAMEL: Detecting and Fixing Performance Problems That Have Non-Intrusive Fixes", IEEE/ACM 37th IEEE International Conference on Software Engineering, 2015.
[2] Qi Luo, Aswathy Nair ·Mark, Grechanik, Denys Poshyvanyk, "FOREPOST: finding performance problems automatically with feedback-directed learning software testing",Springer,2015.
[3] Oswaldo Olivo, Isil Dillig, Calvin Lin, "Static Detection of Asymptotic Performance Bugs in Collection Traversals", 2015.
[4] Sebastiano Panichella, Annibale Panichella, Moritz Beller, Andy Zaidman,Harald C. Gall, "The Impact of Test Case Summaries on Bug Fixing Performance: An Empirical Investigation", IEEE/ACM 38th IEEE International Conference on Software Engineering, 2016.
[5] Mario Linares-Vásquez, Christopher Vendome Qi Luo and Denys Poshyvanyk by How Developers Detect and Fix Performance Bottlenecks in Android Apps, 2015 IEEE ICSME 2015
[6] Xue Han and Tingting Yu by An Empirical Study on Performance Bugs for Highly Configurable Software Systems, ISBN,ESEM '16 ,September 08-09, 2016.
[7] Guo, C., Zhang, J., Yan, J., Zhang, Z., and Zhang, Y. 2013. Characterizing and detecting resource leaks in Android applications. In *Proc. ACM/IEEE Int'l Conf. Automated Soft. Engr*. ASE '13, 389-398.
[8] Hao, S., Li, D., Halfond, W.G.J., and Govindan, R. 2013. Estimating mobile application energy consumption using program analysis. In *Proc. 35th Int'l Conf. Soft. Engr*. ICSE '13. 92-101.
[9] Adiu, Surendra & Geethanjali, N. (2013) "Classification of defects in software using decision tree algorithm", International Journal of Engineering Science and Technology (IJEST), Vol. 5, Issue 6, pp. 1332-1340.
[10] Dommati, Sunil J., Agrawal, Ruchi., Reddy, Ram M. & Kamath, Sowmya (2012) "Bug classification: Feature extraction and comparison of event model using Naïve Bayes approach", International Conference on Recent Trends in Computer and Information Engineering (ICRTCIE'2012), pp. 8-12.
[11] S. Han, Y. Dang, S. Ge, D. Zhang, and T. Xie, "Performance debugging in the large via mining millions of stack traces," in ICSE, 2012.
[12] E. Coppa, C. Demetrescu, and I. Finocchi, "Input-sensitive profiling," in PLDI, 2012.
[13] D. Zaparanuks and M. Hauswirth, "Algorithmic profiling," in PLDI'12.
[14] J. Oh, C. J. Hughes, G. Venkataramani, and M. Prvulovic, "LIME: A framework for debugging load imbalance in multi-threaded execution," in ICSE, 2011.
[15] N. Siegmund, S. S. Kolesnikov, C. K¨astner, S. Apel, D. S. Batory, M. Rosenm ¨uller, and G. Saake, "Predicting performance via automated feature-interaction detection," in ICSE, 2012.
[16] M. Jovic, A. Adamoli, and M. Hauswirth. Catch me if you can: performance bug detection in the wild. In *OOPSLA*, 2011.
[17] N. Meng, M. Kim, and K. S. McKinley. Systematic editing: generating program transformations from an example. In *PLDI*, 2011.
[18] G. Xu, N. Mitchell, M. Arnold, A. Rountev, E. Schonberg, and G. Sevitsky. Finding low-utility data structures. In *PLDI*, 2010.
[19] M. Attariyan and J. Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *OSDI*, 2010.
[20] W. Baek and T. M. Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. In *PLDI*, 2010.
[21] Jin G, Song L, Shi X, et al. Understanding and detecting real-world performance bugs[J]. ACM SIGPLAN Notices, 2012, 47(6): 77-88.
[22] Nistor A, Ravindranath L. SunCat: Helping developers understand and predict performance problems in smartphone applications[C]//Proceedings of the 2014 International Symposium on Software Testing and Analysis. ACM, 2014: 282-292.
[23] Nistor A, Chang P C, Radoi C, et al. CARAMEL: Detecting and Fixing Performance Problems That Have Non-Intrusive Fixes[C]. ICSE, 2015.
[24] Roussel K, Song Y Q, Zendra O. RIOT OS Paves the Way for Implementation of High-Performance MAC Protocols[J]. arXiv preprint arXiv:1504.03875, 2015.
[25] Casado L, Tsigas P. Contikisec: A secure network layer for wireless sensor networks under the contiki operating system[M]//Identity and Privacy in the Internet Age. Springer Berlin Heidelberg, 2009: 133-147.
[26] A. Diwan, M. Hauswirth, T. Mytkowicz, and P. F. Sweeney, "Traceanalyzer: A system for processing performance traces," *Softw. Pract. Exper.*, vol. 41, no. 3, March 2011.
[27] S. Han, Y. Dang, S. Ge, D. Zhang, and T. Xie, "Performance debugging in the large via mining millions of stack traces." in *ICSE*, June 2012.
[28] M. Hauswirth, P. F. Sweeney, A. Diwan, and M. Hind, "Vertical profiling: Understanding the behavior of object-priented applications," in *OOPSLA*, October 2004.

Licensed Under Creative Commons Attribution CC BY

[29] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *PLDI*, June 2005.
[30] A. Muzahid, D. Suarez, S. Qi, and J. Torrellas, "Sigrace: signature- based data race detection," in *ISCA*, June 2009.
[31] A. Nistor, L. Song, D. Marinov, and S. Lu, "Toddler: Detecting performance problems via similar memory-access patterns," in *ICSE*, June 2013.

www.ijsr.net
IJSR
ISSN (Online): 2319-7064

Licensed Under Creative Commons Attribution CC BY