In-Test Adaptation of Workload in Enterprise Application Performance Testing

Maciej Kaczmarski, Philip Perry, John Murphy, A. Omar Portillo-Dominguez
Lero@UCD, School of Computer Science, University College Dublin, Ireland
maciej.kaczmarski@ucdconnect.ie,
{philip.perry,j.murphy,andres.portillodominguez}@ucd.ie

ABSTRACT

Performance testing is used to assess if an enterprise application can fulfil its expected Service Level Agreements. However, since some performance issues depend on the input workloads, it is common to use time-consuming and complex iterative test methods, which heavily rely on human expertise. This paper presents an automated approach to dynamically adapt the workload so that issues (e.g. bottlenecks) can be identified more quickly as well as with less effort and expertise. We present promising results from an initial validation prototype indicating an 18-fold decrease in the test time without compromising the accuracy of the test results, while only introducing a marginal overhead in the system.

Keywords

Performance; Engineering; Testing; Analysis

1. INTRODUCTION

Performance is a critical dimension of quality, as it plays a central role in software usability. However, it is not uncommon that performance issues materialise into serious problems (e.g., outages on production). For example, a survey applied to information technology practitioners [10] reported that 50% of them had faced major performance problems in at least 20% of their deployed enterprise applications. This situation is explained by the pervasive nature of performance, which makes it hard to assess because it is influenced by every aspect of the design, code, and execution environment of an application. Moreover, the latest trends in information technology (e.g., Big Data and Internet of Things) have also increased the complexity of applications, further complicating all activities related to performance [15].

Under these conditions, it is not surprising that the performance testing of enterprise applications is complex and time-consuming [19]. A particularly important challenge is that a significant number of performance issues which occur in enterprise applications are dependent on the input workloads [23]. Although these issues can be identified using current performance testing tools (e.g., Apache JMeter [1]), this strategy is typically inefficient because such tools mainly use static workloads. Therefore, they rely on the expert knowledge of testers to identify a workload which is adequate to find any existing performance issues. Some works have proposed the use of pre-configured workloads [20, 22] (where the workload might vary over time), trying to simulate real-user behaviour using Markov chains. Even though such approaches can create an appropriate workload, for the modelled user behaviour, they still require a considerable amount of testing time. For these reasons, testers usually utilise "standard" workloads, which might be insufficient to identify issues that may not surface on small (or even relatively large) workloads [14]. Moreover, it is often unclear how large is large enough to expose such issues. For example, a recent research study [12] found that 41 (out of 109) studied performance issues were caused due to the wrong assumption of workloads.

The above problem causes two main inefficiencies in the performance testing process: Firstly, it increases the complexity of conducting proper performance testing and analysis, indirectly increasing the cost and time required for these activities (which are commonly limited due to project constraints, like budget or time). Secondly, this problem also increases the risk of overlooking performance issues in the tested applications (which might have serious business impacts, such as unavailability of services or monetary costs).

To tackle these issues, our research work has centred on developing techniques to increase the productivity of performance engineers (hereinafter referred as testers), by decreasing the effort and expertise needed to identify workload-dependent issues and their root causes. The contributions of this paper are: (1) A novel automated approach to dynamically adapt the workload used by a testing tool in application performance testing. The approach is based on a set of diagnostic metrics, evaluated in real-time, to determine if any test workload adjustments are required for the tested application. Consequently, the approach addresses the need of manually configuring the exact workload required to identify these issues. (2) A practical validation of the approach consisting of a prototype, and a set of experiments, to assess the benefits that the approach brings to performance testing as well as its costs.

2. BACKGROUND AND RELATED WORK

When developing any application, it is important to evaluate how well the application performs its functionality when multiple concurrent users are accessing it. This is achieved

with performance testing, which aims to evaluate the behaviour of an application under a given workload [11]. A workload is composed of various types of transactions (e.g., log-in, log-out, search) and a number of concurrent users (commonly virtual). Traditionally, a test run involves using a load generator (e.g., Apache JMeter [1]) for a certain period of time (typically several hours or longer) to simulate the desired set of concurrent users (i.e., test workload) interacting with the application. During a test run, testers usually collect performance-related counters (e.g., performance or resource metrics) in various sampling intervals. The objective is to analyse the behaviour of the monitored counters through time to identify performance issues. Finally, testers normally use some type of diagnosis tools (e.g., IBM WAIT [3]) to further investigate the collected data.

Multiple research works have aimed to improve performance testing and its involved processes. For example, the authors of [19] proposed an approach to simplify the identification of performance regressions. Moreover, the work on [13] introduced an approach to assess the performance of a distributed memory program in a clustered environment. Meanwhile, other works have focused on generating realistic testing data [9], or providing techniques to facilitate the monitoring of performance counters [21]. Finally, other efforts have also centred on reducing the expertise required. For instance, by automating the usage of diagnosis tools [17, 18]. Unlike these works, which have been designed to improve other aspects of performance testing, our approach has been designed to address the specific need of a tester to set an appropriate test workload, hence isolating her from the complexities of identifying it.

3. PROPOSED APPROACH

Our solution is a new automated approach to dynamically adapt the workload used by a performance testing tool, so that it stresses the functionality which is suspicious of having a performance issue (i.e., bug). The aim is to improve the results obtained by a test run (which can be very time-consuming). This is illustrated in Fig. 1, which depicts the contextual view of our solution within the traditional performance testing process (previously described in Section 2). There it can be noticed how our approach enhances this process by proactively monitoring the intermediate test results in order to automatically adjust the workload during the test run (as required), hence eliminating the need of costly trial-and-error test cycles (which traditionally requires the manual adjustment of the workload by the tester).

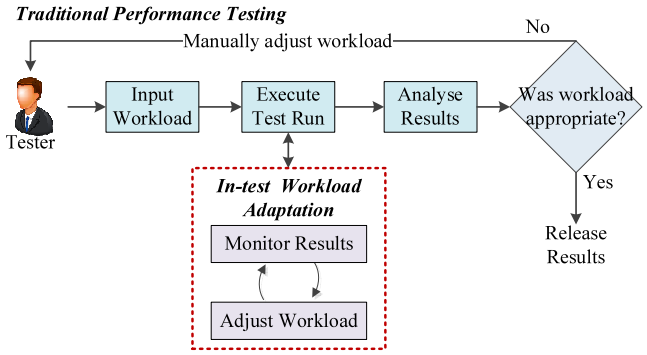

Figure 1: Contextual View

From a workflow perspective, our solution uses a core process (depicted in Fig. 2). The process starts by initialising its configured parameters as well as the chosen policies. There are two different types of policies and our solution requires at least one policy of each type: A diagnosis policy defines the criteria used to determine if a transaction is suspicious of suffering a performance issue, the data sources required to perform the assessment (e.g., performance metrics), and any specific input information required by the policy. An adjustment policy defines the rules to adapt the workload whenever a change is required. The usage of policies allows our approach to be easily extensible (as multiple policies, potentially suitable for different scenarios, can be incorporated). After an initialisation phase occurs, the following loop starts (in parallel to the test run): First, the logic awaits the configured sampling interval, so that the application-under-test has some time to process a certain amount of transactions (as per an initial test workload) before any diagnosis is done. Then, a new set of samples is collected (following the data sources defined by the diagnosis policy). After the collection finishes, the process checks if any transaction is suspicious of suffering a performance issue, following the evaluation criteria defined in the chosen diagnosis policy. Then, if any transaction is suspicious of suffering a performance issue, the workload gets automatically adjusted (as per the configured adjustment policy). This loop continues until the test run finishes.

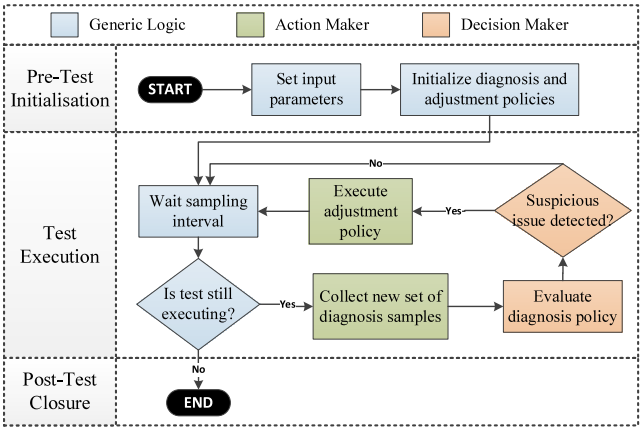

Figure 2: Core Process

From a configuration perspective, the tester needs to (1) indicate a sampling interval to specify how frequent the diagnosis policy will be evaluated; (2) indicate which diagnosis and adjustment policies will be used (among the available alternatives); (3) provide the particular inputs required by the chosen policies. For example, an adjustment policy might require an upper and lower limits that define the valid range within which the policy might adjust the workload.

The component architecture is shown in Fig. 2. The generic component contains all the policy-independent functionality (e.g., the control logic). The other components are the action and decision makers, which encapsulate the adjustment and diagnosis logics, respectively. This architecture was chosen to minimise the changes required to extend the approach. Following the same line of thought, the components are only accessed through interfaces. Each package contains the main interface to expose the supported actions, as well as an abstract class with all common functionality. Then, the hier-

archy can be extended to support specific types of policies. For instance, a possible diagnosis policy could be based on a set of performance metrics (such as response time and throughput). Whenever the performance of any transaction exceeds a defined threshold (above an expected baseline), that transaction would be considered suspicious of suffering a performance issue. Alternatively, another diagnosis policy could use supplementary sources such as the outputs of a diagnosis tool (e.g., IBM WAIT). Meanwhile, a possible adjustment policy might increase the workload of those transactions suspected of suffering a performance issue (probably by a defined workload amount), while also taking care of not exceeding the maximum allowed workload. Similarly, it might decrease the workloads of well-behaved transactions to keep the balance between the maximum allowed workload, while stressing the suspicious transactions as much as possible.

4. EXPERIMENTAL EVALUATION

Setup. The experiments aimed to assess the benefits and costs of using our approach. They were performed in an isolated test environment so that the entire load was controlled. It was composed of two virtual machines (VMs). Each had 2 virtual CPUs at 2.20GHz, 4GB of RAM, and 50GB of hard disk; running Linux Ubuntu 12.04L, and OpenJDK JVM 7 with a 1.6GB heap. One VM used an Apache JMeter 2.9 (a leading open-source tool for performance testing), and the other VM ran an Apache Tomcat 6.0.35 (a popular open-source Java Web Application Server) [2]. As the application to test, we used JPetStore [5], an open-source e-commerce application, which is commonly used in the literature [17, 18]. Also, we used IBM WAIT as diagnosis tool due to its strong analytic capabilities to detect performance issues (e.g., lock contention or database bottleneck) in Java systems [8]. Two types of runs were performed: Following common industry practices [7], the first run type used the traditional approach of static workloads (in the range of [100..2000] concurrent virtual users in increments of 100) and was considered the baseline. The second type used dynamic workloads (our approach). Each test run lasted 2 hours. The main monitored metrics were throughput (tps), response time (ms), CPU (%) and memory (MB) utilisations. They were collected with JMeter and nmon [6]. Finally, the performance bugs were retrieved from WAIT's outputs, which was fed with Javacores [4] (snapshots of the JVM state). Finally, we built our prototype on top of the JMeter tool with Java, which is an object-oriented programming language widely used for being open source and highly portable [16]. Internally, the prototype supports a first heuristic version of our approach, in which any adjustments to the test workload affects all transaction types.

Results. In terms of bugs, our approach was able to identify almost as many bugs as the best static workload. To offer a more comprehensive perspective of the results, we also classified the bugs based on their occurrence frequency (a bug was considered *minor* if it occurred below 5% out of the test run duration, otherwise it was *major*). This is shown in Fig. 3, which compares the best (b-static), the worst (w-static), and the average (a-static) of the static workloads against our approach (dynamic). It can be noticed how our approach outperformed w-static and a-static, while it was competitive against b-static (e.g., it found 2% less minor bugs). It is important to highlight that the workload for b-static was different per bug classification (i.e., 400 for *all*, 1200 for *major*, and 200 for *minor*), further exemplifying the challenges faced by testers.

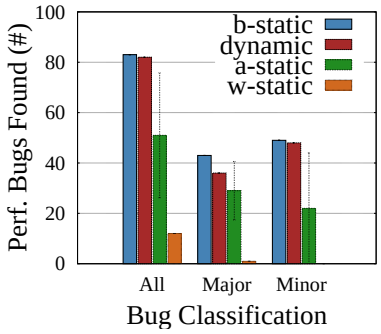

Figure 3: Bugs detection

To complement the discussion of bug identification efficiency, we also present the time-savings that our solution achieved. As the workload was adjusted dynamically (hence, avoiding the need of costly trial-and-error test runs), the tester only required one 2-hour test run (instead of 18 test runs - one per static workload). This led to a reduction in the duration of the performance testing activities of 94% (as shown in Fig. 4).

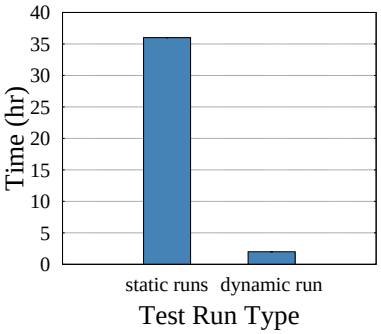

Figure 4: Execution time

In terms of costs, we centred our analysis on the JMeter machine because it is where our solution resides. Results are shown in Fig. 5. Two main observations were identified: Firstly, our solution is, in general terms, more CPU efficient than static workloads (e.g., it required, in average, 13% less average CPU than a-static). This is the result of making a better usage of JMeter's resources (by avoiding costly and ineffective workloads, such as 1800, which was the w-static in terms of resources). Secondly, our solution is marginally more memory-intensive than the static workloads (e.g., an average of 5% more compared to a-static). This increase is due to the internal structures used to keep track of the behaviour of the workloads (mostly Java Collections), as well as the additional classes loaded by the Java Virtual Machine (JVM) to support our logic. It is important to highlight that the b-static in terms of resources was 100, which was one of the w-static in terms of bug finding accuracy. Finally, the overhead introduced by our approach was considered acceptable because the machine was not near exhausting its resources.

Furthermore, a limitation of our current prototype is the usage of the same workload for all types of tested transactions. This design decision was taken to expedite the validation of our research work, even though it was not an optimal

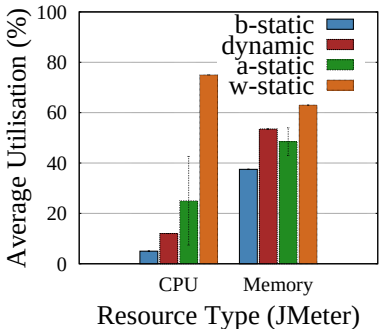

Figure 5: Resource utilisation

adjustment strategy. This is because it is possible that not all transactions suffer performance issues (scenario observed in our experiments). Therefore, it might be possible to stress more some types of transactions (i.e. those with suspected problems), while leaving the others to use smaller workloads. This strategy might lead to additional gains (and probably cost reductions).

5. CONCLUSIONS AND FUTURE WORK

The identification of workload-dependent problems, in the performance testing of enterprise applications, is complex and time-consuming. Furthermore, current testing tools use static workloads, relying on human expert knowledge to set appropriate test workloads. This limits the effectiveness of such tools. To address this problem, this paper proposed a novel automated approach to dynamically adapt the workload used by a performance testing tool. The aim is to increase testers' productivity by decreasing the number of test runs needed to identify workload-dependent performance issues. A prototype was developed around the JMeter tool and then the benefits and overhead of the approach were assessed. The obtained results showed significant time-savings gained by applying the approach: The time required to identify bugs was reduced by 94% (compared to the use of a set of static workloads). These gains were the result of eliminating the need of manually configuring the exact workload required to identify the performance issues. Also, the approach was able to identify almost as many relevant bugs as the best test run (from the tests using static workloads), while only introducing a moderate level of overhead in memory (i.e., 5% increment) utilisation in the JMeter machine.

Future work will focus on strengthening the experimental validation of our approach. For instance, by applying diversification to the tested application behaviours, the diagnosis tools used to identify the bugs, the size and composition of the test environments, as well as the test duration. Furthermore, we plan to investigate how best to extend the capabilities of the approach. For instance, we plan to explore the idea of using different workloads, per transaction type. This is because, as our current results have suggested, this alternative decision strategy might improve the benefits of the approach (by making a better usage of the available resources), while also reducing its computational costs.

6. ACKNOWLEDGMENTS

This work was supported, in part, by Science Foundation Ireland grant 13/RC/2094 and co-funded under the European Regional Development Fund through the Southern & Eastern Regional Operational Programme to Lero - the Irish Software Research Centre (www.lero.ie).

7. REFERENCES

[1] Apache JMeter. http://jmeter.apache.org/.
[2] Apache Tomcat. http://tomcat.apache.org/.
[3] IBM WAIT Tool. https://wait.ibm.com/.
[4] Javacores. http://www-01.ibm.com/support/docview.wss?uid=swg27017906&aid=1.
[5] JPetStore. http://sourceforge.net/projects/ibatisjpetstore/.
[6] Nmon monitor. http://nmon.sourceforge.net/.
[7] Performance Workload Design. Technical report, IBM.
[8] E. Altman, M. Arnold, S. Fink, and N. Mitchell. Performance analysis of idle programs. *ACM SIGPLAN Notices*, 45(10), Oct. 2010.
[9] V. Ayala-Rivera, A. O. Portillo-Dominguez, L. Murphy, and C. Thorpe. COCOA: A synthetic data generator for testing anonymization techniques. *PSD*, 2016.
[10] Compuware. *Applied Perf. Management Survey*. 2007.
[11] Z. M. Jiang. Automated analysis of load testing results. *ISSTA*, page 143, 2010.
[12] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and detecting real-world performance bugs. *ACM SIGPLAN Notices*, 47(6):77–88, 2012.
[13] S. Medya, L. Cherkasova, G. Magalhaes, K. Ozonat, C. Padmanabha, J. Sarma, and I. Sheikh. Towards performance and scalability analysis of distributed memory programs on large-scale clusters. *ICPE*, 2016.
[14] I. Molyneaux. *The Art of Application Performance Testing: Help for Programmers and Quality Assurance.* " O'Reilly Media, Inc.", 2009.
[15] A. O. Portillo-Dominguez, J. Murphy, and P. O'Sullivan. Leverage of extended information to enhance the performance of JEE systems. *ITT*, 2012.
[16] A. O. Portillo-Dominguez, P. Perry, D. Magoni, M. Wang, and J. Murphy. TRINI: an adaptive load balancing strategy based on garbage collection for clustered java systems. *Software: Practice and Experience*, 2016.
[17] A. O. Portillo-Dominguez, M. Wang, J. Murphy, and D. Magoni. Automated wait for cloud-based application testing. *ICSTW*, 2014.
[18] A. O. Portillo-Dominguez, M. Wang, J. Murphy, D. Magoni, N. Mitchell, P. F. Sweeney, and E. Altman. Towards an automated approach to use expert systems in the performance testing of distributed systems. *JAMAICA*, 2014.
[19] W. Shang, A. E. Hassan, M. Nasser, and P. Flora. Automated detection of performance regressions using regression models on clustered performance counters. *ICPE*, 2015.
[20] A. Van Hoorn, J. Waller, and W. Hasselbring. Generating probabilistic and intensity-varying workload for web-based software systems. *SPEC*, pages 124–143, 2008.
[21] A. Van Hoorn, J. Waller, and W. Hasselbring. Kieker: A framework for application performance monitoring and dynamic software analysis. *ICPE*, 2012.
[22] E. Weyuker and A. Avritzer. A metric for predicting the performance of an application under a growing workload. *IBM Systems Journal*, 41(1):45–54, 2002.
[23] E. Xiao, Xusheng. Context-Sensitive Delta Inference for Identifying Workload-Dependent Performance Bottlenecks. *ISSTA*, 2013.