Automated WAIT for Cloud-based Application Testing

Omar Portillo-Dominguez, Miao Wang, John Murphy, Damien Magoni

Automated WAIT for Cloud-based Application Testing

A. Omar Portillo-Dominguez, Miao Wang, John Murphy
Lero, University College Dublin, Ireland
andres.portillo-dominguez@ucdconnect.ie;
{miao.wang,j.murphy}@ucd.ie

Damien Magoni
LaBRI-CNRS, University of Bordeaux, France
magoni@labri.fr

ABSTRACT

Cloud computing is causing a paradigm shift in the provision and use of software. This has changed the way of obtaining, managing and delivering computing services and solutions. Similarly, it has brought new challenges to software testing. A particular area of concern is the performance of cloud-based applications. This is because the increased complexity of the applications has exposed new areas of potential failure points, complicating all performance-related activities. This situation makes the performance testing of cloud applications very challenging. Similarly, the identification of performance issues and the diagnosis of their root causes are time-consuming and complex, usually require multiple tools and heavily rely on expertise. To simplify these tasks, hence increasing the productivity and reducing the dependency on human experts, this paper presents a lightweight approach to automate the usage of expert tools in the performance testing of cloud-based applications. In this paper, we use a tool named Whole-system Analysis of Idle Time to demonstrate how our research work solves this problem. The validation involved two experiments, which assessed the overhead of the approach and the time savings that it can bring to the analysis of performance issues. The results proved the benefits of the approach by achieving a significant decrease in the time invested in performance analysis while introducing a low overhead in the tested system.

Categories and Subject Descriptors

B.8 [**Performance and Reliability**]: Reliability, Testing, and Fault-Tolerance; D.2.5 [**Testing and Debugging**]: Testing tools

General Terms

Algorithms, Measurement, Performance

Keywords

Cloud Computing, Performance Testing, Automation, Performance Analysis, Expert Tools

1. INTRODUCTION

As documented by multiple authors [9, 17], Cloud computing has proven to be a cost-effective and flexible paradigm by which scalable computing power and software services (e.g., computing infrastructures or software resources) can be delivered. This situation has brought many new business opportunities in the enterprise world. For example, the revenue estimations of Gartner[1] indicate that the worldwide Software as a Service (SaaS) revenue (one of the three most well accepted cloud delivery models [10]) is expected to reach US$ 22.1 billion by 2015.

It is also well documented that the Cloud has brought major challenges to the software testing landscape [19, 20]. A particular area of concern is the performance of SaaS applications. This is because Cloud vendors and clients commonly have very demanding quality of service requirements [7]. However, the innate characteristics of a Cloud environment (such as elasticity, scalability and multi-tenancy) have augmented the complexity of the applications, complicating all performance-related activities [10, 25].

Under these conditions, doing performance testing is complex and time-consuming. A special challenge, documented by multiple authors [3, 18], is that current performance tools heavily rely on human experts to understand their output. Also multiple sources are commonly required to diagnose performance problems, especially in highly distributed environments, such as the Cloud. This increases the expertise required to do performance analysis, which is usually held by only a small number of experts inside an organization [23]. Therefore it could potentially lead to bottlenecks where certain activities can only be done by these experts, impacting the productivity of the testing teams [3]. This risk is higher in Cloud environments, as vendors commonly promise up to 24x7 availability of their services. While Cloud computing provides the means to deliver these services, human effort is still required to complete certain key tasks, such as the performance testing of those environments [20].

To simplify the performance analysis and diagnosis, hence increasing the productivity and reducing the dependency on human experts, many researchers have been developing tools with built-in expertise [2, 3]. However, various limitations exist in these tools that prevent their efficient usage in the performance testing of Cloud environments. The data collection usually needs to be controlled manually which, in an

[1] http://www.gartner.com/newsroom/id/1963815

environment composed of multiple nodes to monitor and coordinate simultaneously, is very time-consuming and error-prone due to the vast amount of data to collect and consolidate. This challenge is more complex because the data needs to be processed periodically during the test execution to get incremental results. A similar problem occurs with the outputs, where a tester commonly gets multiple reports, one for each monitored node per data processing cycle. Even though these limitations might be manageable in small testing environments, they prevent the efficient usage of these tools in bigger environments (probably of hundreds of nodes or more), which commonly exist in the Cloud [9]. Similarly, if a tool requires heavy human effort, this might limit its applicability. On the contrary, automation could encourage the adoption of a technique. This strategy has proven successful in performance testing [8, 22], and specially useful in scenarios of continuous testing in the Cloud [17], which requires to test the SaaS applications whenever a change is detected (e.g., bug fixes or feature enhancements).

This paper proposes a lightweight automation approach to address the common usage limitations of an expert system in SaaS performance testing. During our research development work we have successfully applied our approach to the IBM Whole-system Analysis of Idle Time tool (WAIT)[2]. This publicly available tool is a lightweight expert system that helps to identify the main performance inhibitors in Java systems. Our work was validated through two experiments using real-world applications. The results provided evidence about the benefits of the approach: It drastically reduced the effort required by a tester to use and analyze the outputs of the selected expert tool (WAIT). This usage simplification translated into a quicker identification of performance issues. Also the introduced overhead was low (up to 3% when using a common industry *Sampling Interval* of 480 seconds). The contributions of this paper are:

1. A novel lightweight approach to automate the usage of expert systems in SaaS performance testing.
2. A practical validation of the approach consisting of a prototype and two experiments. The first one proves that the overhead of our approach is minimal, and the second proves its productivity benefits.

2. BACKGROUND

Idle-time analysis is a methodology that is used to identify the root causes of under-utilized resources. This approach, proposed in [2], is based on the behavior that performance problems in multi-tier applications usually manifest as idle time of waiting threads. WAIT is an expert system that implements this technique, and it has proven successful in simplifying the detection of performance issues and their root causes in Java systems [2]. WAIT is based on non-intrusive sampling mechanisms available at Operating System level (i.e. "ps" command in Unix) and the Java Virtual Machine (JVM), in the form of *Javacores* [3] (snapshots of the JVM state, offering information such as threads and memory). The fact that WAIT uses standard data sources makes it non-disruptive, as no special flags, restart or instrumentation are required to use it. WAIT also requires infrequent samples to perform its diagnosis, so it has low overhead.

[2]http://wait.ibm.com

[3]http://www-01.ibm.com/support/docview.wss?uid=swg27017906&aid=1

Given its strengths, WAIT is a promising candidate to reduce the dependence on a human expert and the time required for performance analysis in the cloud. However, as with many expert systems that could be used for this purpose, the volume of data generated can be difficult to manage and efficiently process this data can be an impediment to their adoption. For example, the effort required to manually collect data to feed WAIT and the number of reports a tester gets from the WAIT system are approximately linear with respect to the number of nodes and the update frequency of the results. These limitations make WAIT a good candidate to apply our proposed approach.

3. PROPOSED APPROACH

The objective of this work was to automate the manual processes involved in the usage of an expert system. This logic will execute concurrently with the performance test, periodically collecting the required samples, then incrementally processing them with the expert system to get a consolidated output. This scenario is depicted in Figure 1, where our automation shields the tester from the complexities of using the expert system, so that she only needs to interact with the load testing tool.

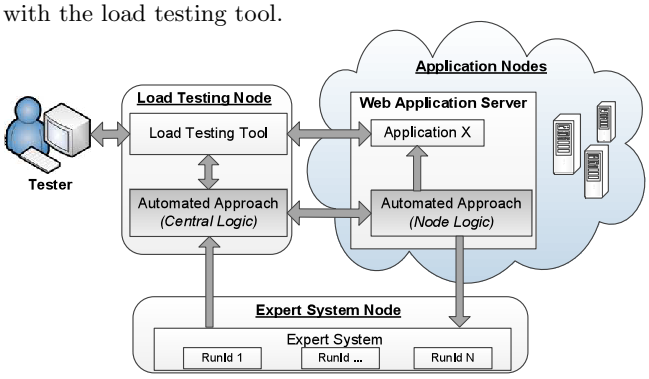

Figure 1: Contextual view of the proposed approach

The detailed approach is depicted in Figure 2. To start, some inputs are required: The list of application nodes to monitor; a *Sampling Interval* to control how often the samples will be collected; a *Time Threshold* to define the maximum time between data uploads; a *Hard Disk (HD) Threshold* to define the maximum storage quota for collected data (to prevent its uncontrolled growth); and a *Backup* flag to indicate if the collected data should be backed up before any cleaning occurs.

The process starts by initialising the configured parameters. Then it gets a new *RunId*, value which will uniquely identify the test run and its collected data. This value is propagated to all the nodes. On each node, the *HD Threshold* and the next *Time Threshold* are initialized. These thresholds allow the approach to adapt to different usage scenarios. For example, if a tester prefers to get updated results as soon as new data is collected, she could set the *HD Threshold* to zero. Next each application node starts the following loop in parallel until the performance test finishes: A new set of data samples is collected. After the collection finishes, the process checks if any of the two thresholds have been reached. If either of these conditions has occurred, the data is sent to the expert system (labeling the data with the *RunId* so that information from different nodes can be identified as

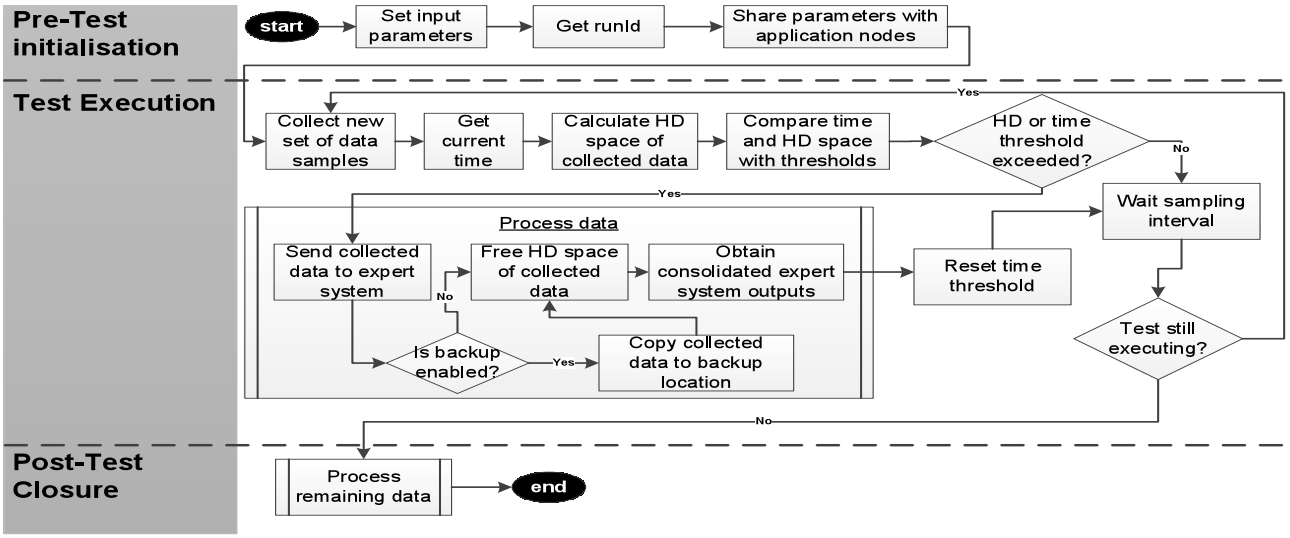

Figure 2: Process flow - automation approach

part of the same test run). If a *Backup* was enabled, the data is copied to the backup destination before it is deleted to keep the HD usage below the threshold. As certain data collections can be costly (i.e. the generation of a memory dump in Java can take minutes and hundreds of megabytes of HD), the *Backup* could be useful to enable off-line analysis of the collected data. Then updated outputs from the expert system are obtained and the *Time Threshold* is reset. Finally, the logic awaits the *Sampling Interval* before a new iteration starts. Once the performance test finishes, any remaining collected data is sent (and backed up if configured) so that it is also processed. Lastly the data is cleared and the final outputs of the expert system are obtained.

Architecture. The proposed approach is implemented with the architecture presented in the component diagram[4] of Figure 3. It is composed of two main components: The *Control Agent* is responsible of interacting with the load testing tool to know when the test starts and ends. It is also responsible of getting the runId and propagate it to all the application nodes. The second component is the *Node Agent* which is responsible for the collection, upload, backup and cleanup in each application node. On each agent, its control logic and *Helper* functionality (i.e. the calculation of the thresholds in the *Node Agent*), are independent of the target expert system and load testing tool. On the contrary, the logic that interfaces with the tools needs to be customized. To minimize the code changes, this logic is encapsulated in *Wrapper* packages which are only accessed through their interfaces. This scenario is presented in Figure 4 which shows the structure of the package *Expert System Wrapper*. It contains a main interface *IExpertSystem* to expose all required actions and an abstract class for all the common functionality. This hierarchy can then be extended to support a specific expert system on different operating systems.

These two components communicate through commands, following the *Command*[5] Design Pattern: The *Control Agent* invokes the commands, while the *Node Agent* implements the logic in charge of executing each concrete command.

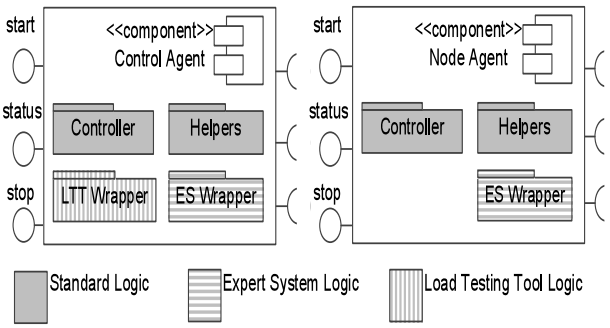

Figure 3: Component diagram of architecture

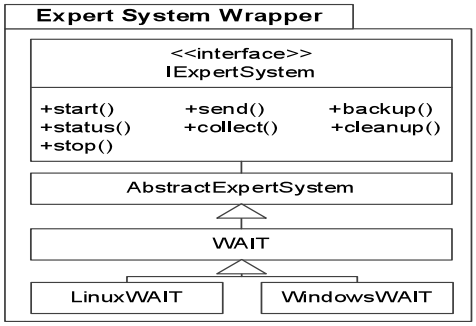

Figure 4: Class diagram of ES Wrapper package

Prototype. Based on the proposed approach, a prototype has been developed. The *Control Agent* was implemented as a plugin for the Rational Performance Tester (RPT) [6], which is a common load testing tool in the industry; the *Node Agent* was implemented as a Java Web Application, and WAIT was the selected expert system due to its analysis capabilities (discussed in Section 2).

[4]https://www.ibm.com/developerworks/rational/library/dec04/bell/

[5]http://www.oodesign.com/command-pattern.html

[6]http://www-03.ibm.com/software/products/us/en/performance

4. EXPERIMENTAL EVALUATION

Two experiments were performed. The first one evaluated the overhead introduced by our proposed approach. Meanwhile, the second one assessed the productivity benefits that a tester can gain by using the proposed approach. Two environment configurations were used: One was composed of an RPT node, one application node and a *WAIT Server* node; the other was composed of a RPT node, a load balancer node, two application nodes and a *WAIT Server* node. All VMs had 2 virtual CPUs, 3GB of RAM, and 50GB of HD; running Linux Ubuntu 12.04L 64-bit, and Oracle Hotspot JVM version 7 with a maximal heap of 2GB.

Experiment 1: Overhead Evaluation. The objective here was to quantify the overhead of the proposed approach and involved the assessment of four metrics: Throughput (tps), response time (ms), CPU (%) and memory (MB) utilisation. All metrics collected through RPT. Also, two real-world applications were used: iBatis JPetStore 4.0 [7] which is an e-commerce shopping cart. It ran over an Apache Tomcat 6.0.35. The other application was IBM WebSphere Portal 8.0.1 [8], a leading enterprise portal solution. It ran over an IBM WebSphere Application Server 8.0.0.5.

Firstly, the overhead was measured in a single-node environment using three combinations of WAIT: The applications alone to get a baseline (BL), the applications with manual WAIT data collection, and the applications with the automated WAIT (AWAIT). For each combination using WAIT, the *Sampling Interval* (SI) was configured to 480 seconds (a commonly used value) and 30 seconds (minimum value recommended for WAIT). The remaining test configurations were set to reflect real-world conditions: A workload of 2,000 concurrent users; a duration of 1 hour; a *HD Threshold* of 100MB; and a *Time Threshold* of 10 minutes.

For JPetStore, each test run produced around 500k transactions. The results presented in Table 1 showed that using WAIT with a SI of 480 seconds had practically no impact in terms of response time and throughput. Furthermore the difference in resource consumption between the two modalities of WAIT was around 1%. This difference was mostly related to the presence of the *Node Agent* because the uploaded data was very small (around 200KB every 10 minutes). When a SI of 30 seconds was used, the impact on response time and throughput appeared. Since the throughput was similar between the WAIT modalities, the impact was caused by the *Javacore* generation as it is the only step shared between the modalities. On average, the generation of a *Javacore* took around 1 second. Even though this cost was insignificant in the higher SI, with 30 seconds the impact was visible. On the contrary, the difference in response time between the two modalities of WAIT was caused by the upload and backup processes (around 4MB of data every 10 minutes) which are steps exclusive to AWAIT. In terms of resource consumption, the differences between the WAIT modalities remained within 1%.

For Portal, each test run produced around 400k transactions and the results are presented in Table 2. They show similar trends to the results in Table 1, but a few key differences were identified: First, the impact on response time and throughput were visible even with the SI of 480 seconds. Also, the differences between the results for the two SIs were bigger. As the experimental conditions were the same, it was initially assumed that these differences were related to the dissimilar functionality of the tested applications. This was confirmed after analysing the *Javacores* generated by Portal, which allowed to quantify the differences in behavior of Portal: The average size of a *Javacore* was 5.5MB (450% bigger than JPetStore's), its average generation time was 2 sec (100% bigger than JPetStore's), and a maximum generation time of 3 sec (100% bigger than JPetStore's).

Table 1: JPetStore - Overhead results

WAIT Modality & SI	RT_{AVG} (ms)	RT_{MAX} (ms)	T_{AVG} (tps)	CPU_{AVG} (%)	MEM_{AVG} (MB)
None (BL)	1889	44704	158	36	1429
WAIT, 480s	0.0%	0.0%	0.0%	1.1%	3.0%
AWAIT, 480s	0.0%	0.0%	0.0%	2.0%	3.7%
WAIT, 30s	1.6%	0.4%	-3.1%	1.47%	4.1%
AWAIT, 30s	4.4%	0.5%	-4.0%	2.53%	4.4%

Table 2: Portal - Overhead results

WAIT Modality & SI	RT_{AVG} (ms)	RT_{MAX} (ms)	T_{AVG} (tps)	CPU_{AVG} (%)	MEM_{AVG} (MB)
None (BL)	4704	40435	98	76	3171
WAIT, 480s	0.7%	0.6%	-0.1%	0.63%	2.2%
AWAIT, 480s	3.4%	1.0%	-2.8%	1.13%	4.1%
WAIT, 30s	14.9%	5.4%	-5.6%	2.23%	5.3%
AWAIT, 30s	16.8%	9.1%	-5.7%	2.97%	6.0%

To explore the small differences between the runs and the potential environmental variations that were experienced during the experiments, a Paired t-Test [9] was done (using a significance level of $p<0.1$) to evaluate if the differences were statistically significant. This analysis indicated that they were only significant when using a SI of 30 seconds. This reinforced the conclusion that the overhead was low and the observation that the SI of 480 seconds was preferable.

A second test was done to validate that the overhead remained low in a multi-node environment over a longer test run. This test used JPetStore and the AWAIT with a SI of 480 seconds. The rest of the set-up was identical to the previous tests except the workload which was doubled to compensate for the additional application node and the test duration which was increased to 24 hours. Even though the results were slightly different than the single-node run, they proved that the proposed approach was reliable, as using AWAIT had minimal impact in terms of performance (0.5% in RT_{AVG}, 0.2% in RT_{MAX} and 1.4% in T_{AVG}). The consumption of resources behaved similarly (an increment of 0.85% in CPU_{AVG} and 2.3% in MEM_{AVG}). A paired t-Test also indicated that these differences, compared with the baseline, were not significant.

In conclusion, the results proved that the overhead caused by the automated approach was low, therefore the results of a performance test are not compromised. Due to the impact that the SI and the application behavior might have on the overhead, it is important to consider these factors in the configuration. In our case, a SI of 480 seconds proved efficient in terms of overhead for the two tested applications.

Experiment 2: Assessment of productivity benefits. Here the objective was to assess the benefits our approach

[7] http://sourceforge.net/projects/ibatisjpetstore/

[8] http://www-03.ibm.com/software/products/us/en/portalserver

[9] http://www.aspfree.com/c/a/braindump/comparing-data-sets-using-statistical-analysis-in-excel/

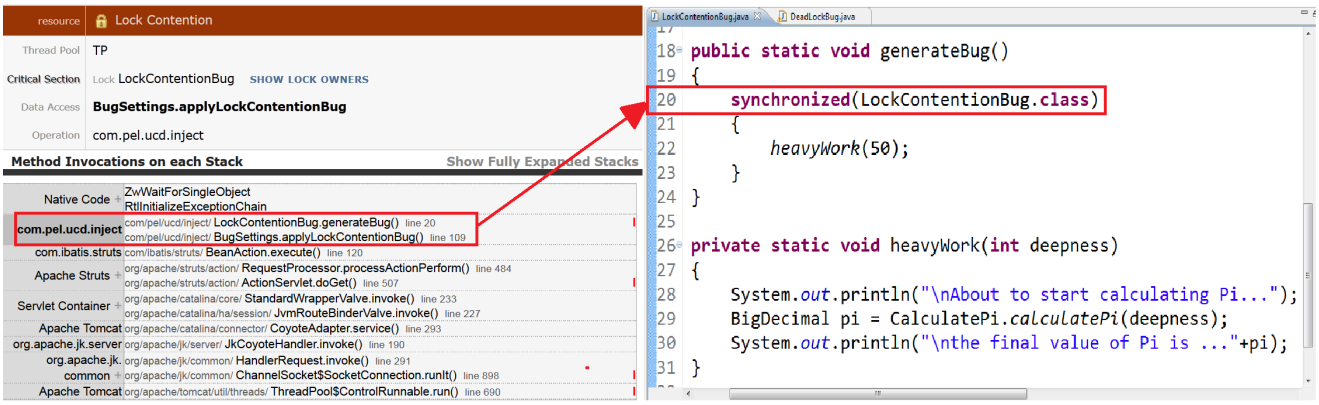

Figure 5: Lock contention issue in the WAIT report and the actual source code

brings to a performance tester. First, three common performance issues were injected in JPetStore: A lock contention bug (composed of a heavy calculation within a synchronized block of code), an I/O latency bug (composed of an expensive file reading method) and a deadlock bug (composed of an implementation of the classic "friends bowing" example[10]). Then AWAIT monitored the application to assess how well it was able to identify the injected bugs and estimate the corresponding time savings in performance analysis. The set-up was identical to the multi-node test previously described except the duration which was one hour.

The 1st ranked issue was not an injected bug but an issue related to the clustering set-up of Tomcat. The 2nd ranked issue was the lock contention. It is worth noting that both issues were detected since the early versions of the incremental report from the automated tool. Their high frequency (above 96% of the samples) could have led a tester to pass this information to the development team so that the diagnosis could start far ahead of the test completion. The final report reinforced the presence of these issues.

After identifying an issue, a tester can see more details, including the type of problem, involved class, method and source code line. Figure 5 shows the information of our lock contention bug, which was located in the class LockContentionBug, the method generateBug and the line 20. When comparing this information with the actual code, one can see it is precisely the line where the bug was injected (taking a class lock before doing a very CPU intensive logic). The 3rd ranked issue in the report was a symptom of the lock contention bug, suggesting it was a major problem (the issues were correlated by comparing their information, which pinpointed to the same class and method). Finally, the I/O latency bug was identified in 4th place.

The deadlock bug did not appear in this test run, somehow prevented by the lock contention bug which had a bigger impact than planned. As in any regular test phase, the identified bugs were fixed and a new run was done to review if any remaining performance issues existed. Not surprisingly, the deadlock bug appeared in the second test run.

As all injected bugs were identified, this experiment was considered successful. In terms of time, two main savings were documented. First, the automated approach practically reduced the effort of using WAIT to zero. After a one-time installation which took no more than 15 minutes for all nodes, the only additional effort required to use the automated approach was a few seconds spent configuring it (i.e. to change the SI). The second time saving occurred in the analysis of the WAIT reports. Previously, a tester would have ended with multiple reports. Now a tester only needs to monitor a single report which is refreshed periodically. Overcoming the usage constraints of our selected expert system (WAIT) also allowed exploiting its expert knowledge capabilities. Even though it might be hard to define an average time spent identifying performance issues, a conservative estimate of 2 hours per bug could help to quantify these savings. In our experiment, instead of spending an estimated 6 hours analysing the issues, it was possible to identify them and their root causes in a matter of minutes with the information provided by the WAIT report. As seen in the experiment, additional time can be saved if the relevant issues are reported to developers in parallel to the test execution. This is especially valuable in long-term runs, which are common in performance testing and typically last several days.

To summarize the results, they were very promising as it was possible to measure the benefits that a tester can gain by using an expert system through our proposed automation approach: After a quick installation, the time required to use the automated WAIT was minimal. Moreover a tester now only needs to monitor a single WAIT report, which offers a consolidated view of the results.

5. RELATED WORK

Performance testing. Multiple research works have focused on improving its involved processes. For example, the authors of [14] introduced a cloud-based testing harness which uses an execution engine to parallelise the run of test scripts on large shared-clusters of nodes. Meanwhile, the work on [4] presents an approach to automate the deployment of the major open-source IaaS cloud kits (such as OpenNebula[11]) on Grid 5000 environments. Other efforts have concentrated on evaluating the benefits of cloud testing in practice. For example, the work on [15] presents an industrial study case of the new methodologies and tools

[10]http://docs.oracle.com/javase/tutorial/essential/concurrency/deadlock.html

[11]http://opennebula.org/

that have developed to streamline the testing of their cloud-based applications. Finally, other works have focused on automation. However, most of this research has focused on automating the generation of load test suites [1, 13]. For example, [16] proposes an approach to automate the generation of test cases based on specified levels of load and combinations of resources.

Performance Analysis. Other works have focused on addressing the challenges in performance analysis in the Cloud [11, 24]. A high percentage of the proposed techniques require instrumentation. For example, the authors in [26] instrument the source code of the monitored applications to mine the sequences of call graphs under normal operation, information which is later used to infer any relevant error patterns. A similar case occurs with the works presented in [6, 21] which rely on instrumentation to dynamically infer invariants within the applications and detect programming errors. In all these cases, instrumentation would obscure the performance of an application during performance testing hence discouraging their usage. On the contrary, our proposed approach does not require any instrumentation.

Finally, the authors of [5, 12] present frameworks to monitor software services. Unlike these works, which have been designed to assist on operational support activities, our approach has been designed to address the specific needs of a tester in the performance testing of cloud environments, isolating her from the complexities of an expert system.

6. CONCLUSIONS

The identification of performance problems in cloud environments is complex and time-consuming. Performance testing is challenging because the complexity of the cloud-based applications has exposed new areas of potential failure points. While the identification of performance issues and the diagnosis of their root causes usually require multiple tools and heavily rely on expertise. To simplify these tasks, this work proposed a novel approach to automate the usage of an expert system in a cloud test environment. A prototype was developed around the WAIT tool and then its benefits and overhead were assessed. The results showed that the introduced overhead was low (between 0% to 3% when using a common industry *Sampling Interval*). Also the results showed the time savings gained by applying the approach. In our case, the effort to utilize WAIT was reduced to seconds. This optimization then simplified the identification of performance issues. In our case, all defects injected in JPetStore were detected in a matter of minutes using WAIT's consolidated outputs. In contrast, a manual analysis might have taken hours.

7. ACKNOWLEDGMENTS

This work was supported, in part, by Science Foundation Ireland grant 10/CE/I1855 to Lero - the Irish Software Engineering Research Centre (www.lero.ie).

8. REFERENCES

[1] J. Albert, Elvira, Miguel Gomez-Zamalloa. Resource-Driven CLP-Based test generation. *LOPSTR*, 2012.

[2] E. Altman, M. Arnold, S. Fink, and N. Mitchell. Performance analysis of idle programs. *ACM SIGPLAN Notices*, Oct. 2010.

[3] V. Angelopoulos, T. Parsons, J. Murphy, and P. O'Sullivan. GcLite: An Expert Tool for Analyzing GC Behavior. *CSACW*, 2012.

[4] L. N. Badia, S., Carpen-Amarie, A. Enabling Large-Scale Testing of IaaS Cloud Platforms on the Grid 5000 Testbed. *TTC*, 2013.

[5] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using magpie for request extraction and workload modelling. *OSDI*, 2004.

[6] Y. C. Csallner. Dsd-crasher: a hybrid analysis tool for bug finding. *ISSTA*, 2006.

[7] ClausM Oldt. Behind-the-Scenes at Salesforce. 2009.

[8] S. Dosinger, Stefan, Richard Mordinyi. Continuous integration servers for increasing effectiveness of automated testing. *ASE*, 2012.

[9] T. Gao, J., Bai, X. Cloud Testing- Issues, Challenges, Needs and Practice. *Software Engineering: An International Journal*, 2011.

[10] T. U. Gao, J., Bai, X. SaaS Testing on Clouds. Issues, Challenges, and Needs. *SOSE*, 2013.

[11] Z. George Candea, Stefan Bucur. Automated Software Testing as a Service (TaaS). *CloudCom*, 2010.

[12] V. Hoorn, M. Rohr, W. Hasselbring, J. Waller, J. Ehlers, and D. Kieselhorst. Design and Application of the Kieker Framework. 2009.

[13] M. L. C. Briand, Y. Labiche. Using genetic algorithms for early schedulability analysis and ST in RT systems. *Genetic Prog. and Evolvable Machines*, 2006.

[14] S. V. G. L. Ciortea, C. Zamfir. Cloud9: A software testing service. *LADIS*, 2009.

[15] T. Lynch, M., Cerqueus, T. Testing a Cloud Application: IBM SmartCloud iNotes. *TTC*, 2013.

[16] J. M. S. Bayan. Automatic stress and load testing for embedded systems. *ICSA*, 2006.

[17] M. P. Jogalekar. Evaluating the scalability of distributed systems. *IEEE Trans. Parallel and Distributed Systems*, June 2000.

[18] T. Parsons and J. Murphy. Detecting Performance Antipatterns in Component Based Enterprise Systems. In *Middleware Doctoral Symposium*, 2008.

[19] R. Collard. Performance innovations, testing implications. *Software Test and Performance Magazine*, Aug. 2009.

[20] S. Riungu, L. M., Taipale, O. Research Issues for Software Testing in the Cloud. *CloudCom*, 2010.

[21] M. S. Hangal. Tracking down software bugs using automatic anomaly detection. *ICSE*, 2002.

[22] S. Shahamiri, W. Kadir, and S. Mohd-Hashim. A Comparative Study on Automated Software Test Oracle Methods. ICSEA, 2009.

[23] W. Spear, S. Shende, A. Malony, R. Portillo, P. J. Teller, D. Cronk, S. Moore, and D. Terpstra. Making Performance Analysis and Tuning Part of the Software Development Cycle. *DoD HPCM*, 2009.

[24] S. Vijayanathan Naganathan. The Challenges Associated with SaaS Testing. *Infosys*, 2011.

[25] Q. M. W. Tsai, Y. Huang. Testing the Scalability of SaaS applications. *RTSOAA*, 2011.

[26] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: mining temporal api rules from imperfect traces. *ICSE*, 2008.