Towards an Automated Approach to Use Expert Systems in Performance Testing

A. Omar Portillo-Dominguez[1], Miao Wang[1], Philip Perry[1], John Murphy[1], Nick Mitchell[2], Peter F. Sweeney[2], and Erik Altman[2]

[1] Lero - The Irish Software Engineering Research Centre, Performance Engineering Laboratory, UCD School of Computer Science and Informatics, University College Dublin, Ireland
andres.portillo-dominguez@ucdconnect.ie,
{philip.perry,miao.wang,j.murphy}@ucd.ie,
[2] IBM T.J. Watson Research Center, Yorktown Heights, New York, USA
{nickm,pfs,ealtman}@us.ibm.com

Abstract. Performance testing in highly distributed environments is very challenging. Specifically, the identification of performance issues and the diagnosis of their root causes are time-consuming and complex tasks which usually require multiple tools and heavily rely on expertise. To simplify these tasks, hence increasing the productivity and reducing the dependency on human experts, many researchers have been developing tools with built-in expertise for non-expert users. However, managing the huge volume of data generated by these tools in highly distributed environments prevent their efficient usage in performance testing. To address these limitations, this paper presents a lightweight approach to automate the usage of expert tools in performance testing. 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, using real-life applications, 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.

Keywords: Performance testing, automation, performance analysis, expert tools, distributed systems

1 Introduction

It is an accepted fact in the industry that performance is a critical dimension of quality and should be a major concern of any software project. This is especially true at enterprise-level, where system performance plays a central role in usability. However it is not uncommon that performance issues occur and materialize into serious problems in a significant percentage of applications (i.e. outages on production environments or even cancellation of software projects). For example, a 2007 survey applied to information technology executives [1] reported that 50% of them had faced performance problems in at least 20% of their deployed applications.

This situation is partially explained by the pervasive nature of performance, which makes it hard to assess because performance is practically influenced by every aspect of the design, code, and execution environment of an application. The latest trends in information technology (such as Service Oriented Architecture[3] and Cloud Computing[4]) have also augmented the complexity of applications further complicating activities related to performance.

Under these conditions, it is not surprising that doing performance testing is complex and time-consuming. A special challenge, documented by multiple authors [2–4], 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. For instance in Java: thread dumps, garbage collection logs, heap dumps, CPU utilization and memory usage, are a few examples of the information that a tester could need to understand the performance of an application. This problem increases the expertise required to do performance analysis, which is usually held by only a small number of experts inside an organization[5]. Therefore it could potentially lead to bottlenecks where certain activities can only be done by these experts, impacting the productivity of the testing teams[4].

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 for non-expert users [6, 7, 4]. However, various limitations exist in these tools that prevent their efficient usage in the performance testing of highly distributed environments. The data collection usually needs to be controlled manually which, in an 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. To exemplify this problem, let's use the Eclipse Memory Analyzer Tool[5] (MAT), which is a popular open source tool to identify memory consumption issues in Java. If a tester wants to use MAT to monitor an environment composed of 100 nodes during a 24-hour test run and get incremental results every hour, she would need to manually coordinate the data gathering of memory snapshots and the generation of the tool's reports. These steps conducted periodically for the 100 nodes every hour, which yields a total of 2400 iterations. Moreover the tester would have to review the multiple reports she would get per hour to evaluate if any memory issues exist. As an alternative, she may concentrate the analysis

[3] http://msdn.microsoft.com/en-us/library/aa480021.aspx
[4] http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf
[5] http://www.eclipse.org/mat/

on a single node, assuming it is representative of the whole system. However it generates the risk of potentially overlooking issues in the tested application.

In addition to these challenges, the overhead generated by any technique should be low to minimize the impact it has in the tested environment (i.e. inaccurate results or abnormal behavior). Otherwise the technique would not be suitable for performance testing. For example, instrumentation[6] is currently a common approach used in performance analysis to gather input data [8–11]. However, it has the downside of obscuring the performance of the instrumented applications, hence compromising the results of performance testing. Similarly, if a tool requires heavy human effort, this might limit its applicability. On the contrary, automation could encourage the adoption of a technique. As documented by the authors in [12], this strategy has proven successful in performance testing.

Finally, to ensure that our research is helpful to solve real-life problems in the software industry, we have been working with our industrial partner, IBM System Verification Test (SVT), to understand the challenges in their day-to-day testing activities. Their feedback confirms that there is a real need to simplify the usage of expert tools so that testers can do analysis tasks in less time. Another key need is that testers be able to convey their results in a meaningful way with teams in development and in operations. The IBM Whole-system Analysis of Idle Time tool (WAIT)[7], used in this work, meets this need. This publicly available expert system helps to identify the main performance inhibitors that exist in Java systems. It is also sufficiently lightweight and easy to use that it can be and has been deployed in production, test and development environments.

This paper proposes a lightweight automation approach that addresses the common usage limitations of an expert system in performance testing. Furthermore, during our research development work we have successfully applied our approach to the WAIT tool. Our work was validated through two experiments using real-world applications. The first experiment evaluated the overhead introduced by our approach. The second experiment assessed the productivity gains that our approach can bring to the performance testing process. 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, including the pinpointing of the responsible classes and methods. Furthermore the introduced overhead was low (between 0% to 3% when using a common industry *Sampling Interval* of 480 seconds).

The main contributions of this paper are:

1. A novel lightweight approach to automate the usage of expert systems in performance testing.

2. A practical validation of the approach consisting of an implementation around the WAIT tool and two experiments using real-life applications. The first experiment demonstrates that the overhead of our approach is minimal, and the second experiment demonstrates its productivity benefits.

[6] http://msdn.microsoft.com/en-us/library/aa983649(VS.71).aspx
[7] http://wait.ibm.com

The rest of this paper is structured as follows: Section 2 discusses the background. Section 3 explains the proposed approach, while Section 4 explores the experimental evaluation and results. Section 5 shows the related work. Finally Section 6 presents the conclusions and future work.

2 Background

Idle-time analysis is a methodology that is used to identify the root causes of under-utilized resources. This approach, proposed in [6], is based on the observed behavior that performance problems in multi-tier applications usually manifest as idle time of waiting threads. WAIT is an expert system that implements the idle-time analysis and identifies the main performance inhibitors that exist on a system. Moreover it has proven successful in simplifying the detection of performance issues and their root causes in Java systems [6, 13].

WAIT is based on non-intrusive sampling mechanisms available at Operating System level (i.e. "ps" command in a Unix environment) and the Java Virtual Machine (JVM), in the form of *Javacores* [8] (diagnostic feature to get a quick snapshot of the JVM state, offering information such as threads, locks 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.

From an end-user perspective, WAIT is simple: A user only needs to collect as much data as desired, upload it to a public web page (wait.ibm.com) and get a report with the findings. This process can be repeated multiple times to monitor a system through time. Internally, WAIT uses an engine built on top of a set of expert rules to perform the analysis. Fig. 1 shows an example of a WAIT Report. The top area summarizes the usage of resources (i.e. CPU or memory) and the types of threads. The bottom section shows all the performance inhibitors that have been identified, ranked by frequency and impact. Each problem category is indicated with a different color. For example, in Fig. 1 the top issue appeared in 53% of the samples and affected 7.6 threads on average. Moreover the affected resource was the network and the issue was caused by waiting on the database.

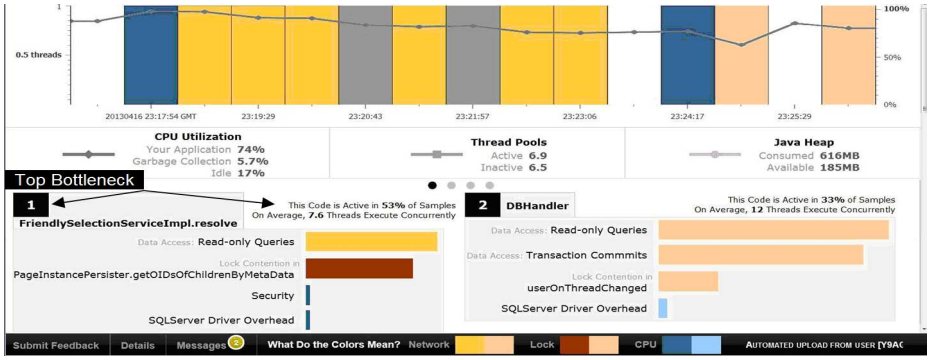

Fig. 1. Example of WAIT Report

[8] 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 reduce the time required for performance analysis. However, as with many expert systems that could be used for testing distributed software systems, the volume of data generated can be difficult to manage and efficiently process this data can be an impediment to their adoption. 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. This makes WAIT a good candidate to apply our proposed approach.

3 Proposed Approach and Architecture

3.1 Proposed Approach

The objective of this work was to automate the manual processes involved in the usage of an expert system (ES). This logic will execute concurrently with the performance test, periodically collecting the required samples, then incrementally processing them with the ES to get a consolidated output. This scenario is depicted in Fig. 2 where the automation shields the tester from the complexities of using the ES, so that she only needs to interact with the load testing tool.

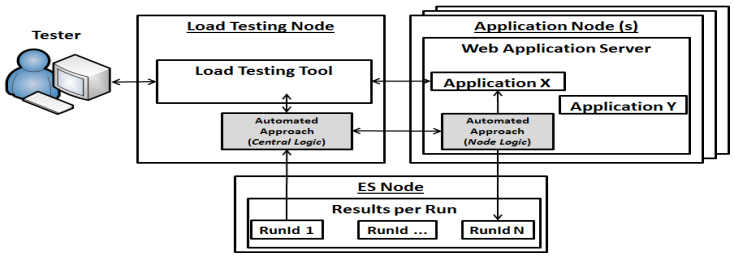

Fig. 2. Contextual View of the Proposed Approach

The detailed approach is depicted in the Fig. 3. To start, some inputs are required: The list of 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 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 initializing the configured parameters. Then it gets a new *RunId*, value which will uniquely identify the test run and its collected data. This value is then propagated to all the nodes. On each node, the *Hard Disk Usage 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 *Hard Disk Threshold* to zero.

Then each 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 system 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

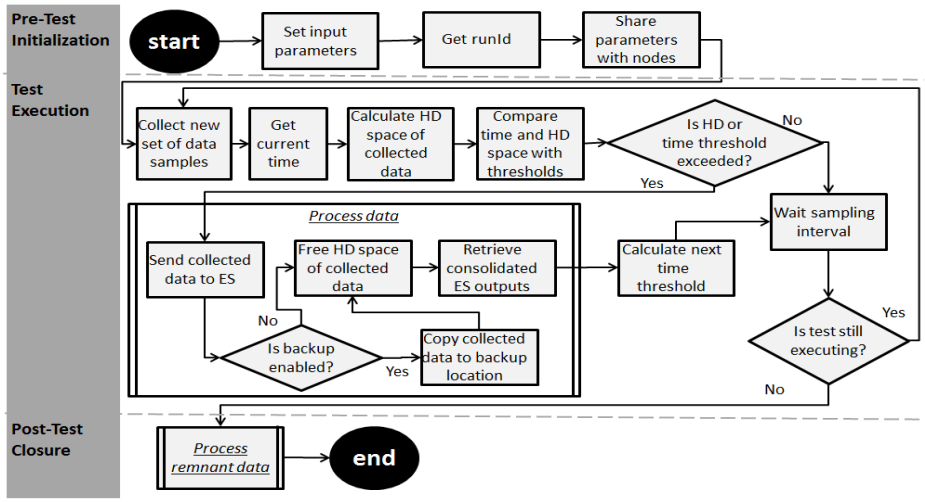

Fig. 3. Process Flow - Automation approach

with the *RunId* so that information from different nodes can be identified as 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 require hundreds of megabytes of HD), this step could be useful to enable further off-line analysis of the collected data. Next updated outputs from the ES are retrieved and the new *Time Threshold* is calculated. 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 by the expert system. Lastly the data is cleared and the final outputs of the expert system are obtained.

3.2 Architecture

The approach is implemented with the architecture presented in Fig. 4. It is composed of two main components: The *Control Agent* is responsible of interacting with the load testing tool (LTT) to know when the test starts and ends. It is also responsible of getting the runId and propagate it to all the nodes. The second component is the *Node Agent* which is responsible for the collection, upload, backup and cleanup in each 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 ES and LTT. On the contrary, the logic that interfaces with the tools need to be customized. To minimize the code changes, this logic is encapsulated in two *Wrapper* packages which are only accessed through their interfaces. This scenario is presented in Fig. 5 which shows the structure of the package *ES 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 ES on different operating systems.

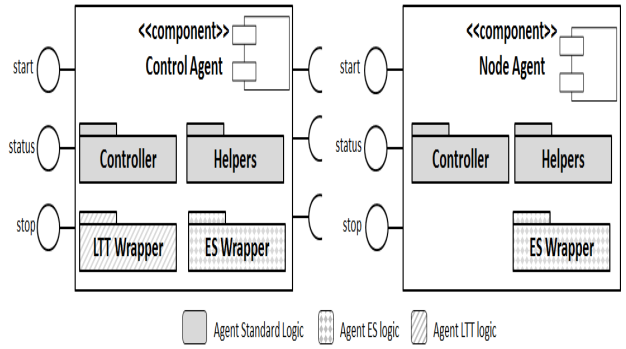

Fig. 4. Component Diagram of Architecture

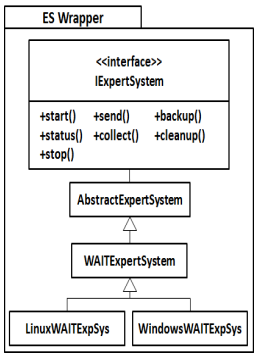

Fig. 5. Class Diagram of ES Wrapper package

These two components communicate through commands, following the *Command*[9] Design Pattern: The *Control Agent* invokes the commands, while the *Node Agent* implements the logic in charge of executing each concrete command. An example of these interactions is depicted in Fig. 6. Once a tester has started a performance test (step 1), the *Control Agent* propagates the action to all the nodes (steps 2 to 4). Then each *Node Agent* performs its periodic data collection (steps 5 to 9) until any of the thresholds is satisfied and the data is sent to the ES (steps 10 and 11). This continues iteratively until the test ends. At that moment, the *Control Agent* propagates the stop action (steps 21,22 and 24). At any time, the tester might choose to review the intermediate results of the ES (steps 12 to 14) until getting the final results (steps 25 to 27).

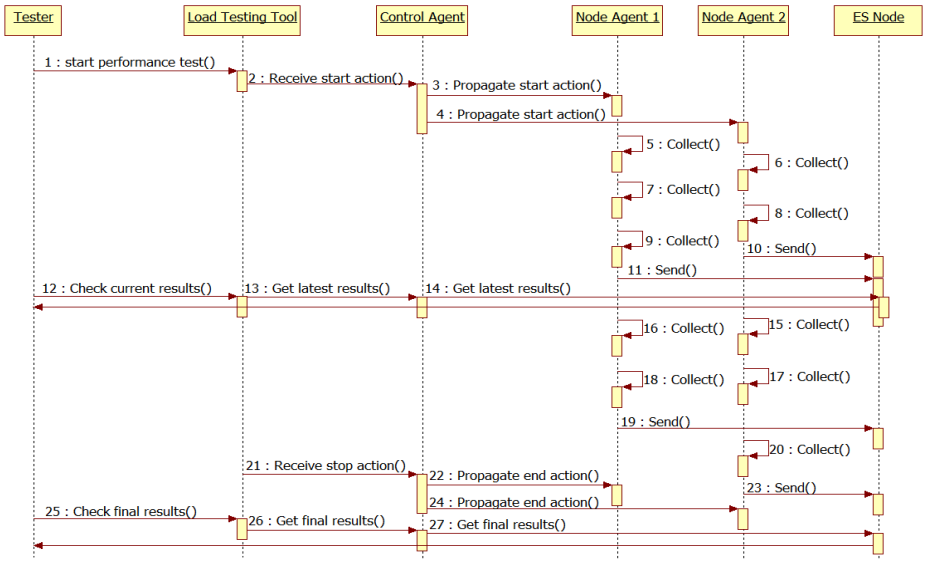

Fig. 6. Sequence diagram of the automated approach

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

4 Experimental Evaluation

4.1 Prototype

Based on the proposed approach, a prototype has been developed in conjunction with our industrial partner IBM. The *Control Agent* was implemented as a plugin for the Rational Performance Tester (RPT) [10], 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).

Once the agents are installed, WAIT can be configured as any other resource in RPT as shown in Fig. 7. Similarly, WAIT can be monitored in the *Performance Report* of RPT under the *Resource View*. This is depicted in Fig. 8, which shows some metrics that a tester can check per node: The number of monitored processes, the started data collections and the completed ones. Additionally, the consolidated WAIT report is accessible within RPT, so a tester does not need to leave RPT during the whole performance test. This is shown in Fig. 9.

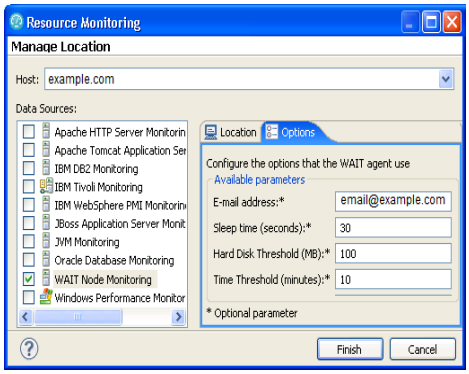

Fig. 7. WAIT configuration in RPT

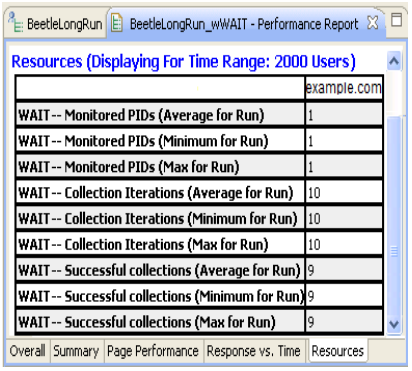

Fig. 8. WAIT monitored in RPT

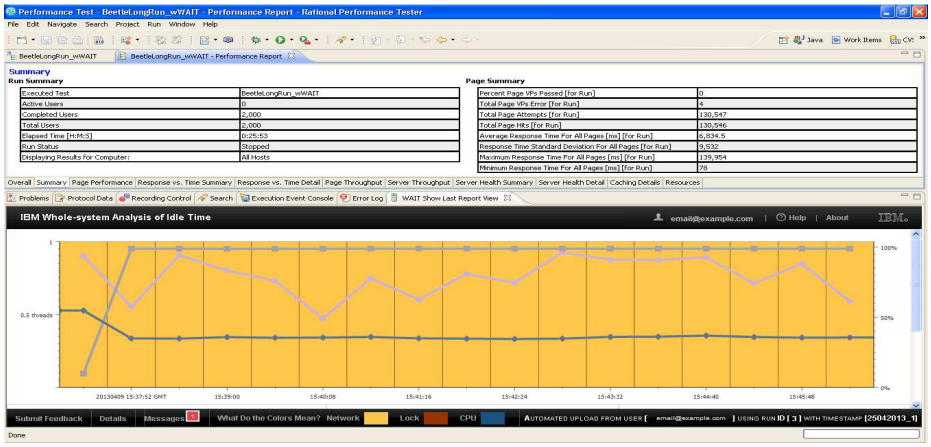

Fig. 9. WAIT Report accessible within RPT

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

4.2 Experimental Set-up

Two experiments were performed. The first one aimed to evaluate if the overhead introduced by the proposed approach was low so that it does not compromise the results of a performance test. Meanwhile, the second experiment shows the productivity benefits that a tester can gain by using the proposed approach. Additionally two environment configurations were used, as shown in Fig. 10. 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 connected by a 10-GBit LAN.

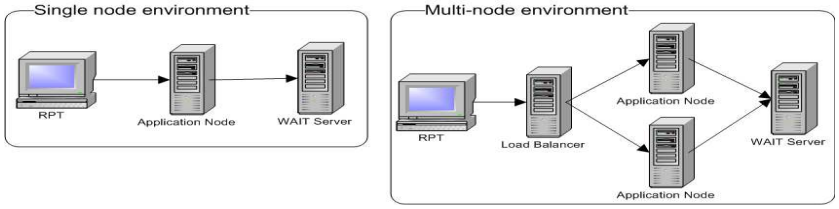

Fig. 10. Environment Configurations

The RPT node ran over Windows XP with an Intel Xeon CPU at 2.67 GHz and 3GB of RAM using RPT 8.2.1.3. The *WAIT Server* was run over Red Hat Enterprise Linux Server 5.9, with an Intel Xeon CPU at 2.66 GHz and 2GB of RAM using Apache Web Server 2.2.3. Each application node was a 64-bit Windows Server 2008, with an Intel Xeon E7-8860 CPU at 2.26 GHz and 4GB of RAM running Java 1.6.0 IBM J9 VM (build 2.6). Finally, the load balancer node had the same characteristics of the *WAIT Server* node.

4.3 Experiment #1: Overhead Evaluation

The objective here was to quantify the overhead of the proposed approach and involved the assessment of four metrics: Throughput (hits per second), response time (milliseconds), CPU (%) and memory (MB) utilization. All metrics were collected through RPT. Furthermore, two real-world applications were used: iBatis JPetStore 4.0 [11] which is an upgraded version of Sun's PetStore, an e-commerce shopping cart. It ran over an Apache Tomcat 6.0.35. The other application was IBM WebSphere Portal 8.0.1 [12], a leading solution in the enterprise portal market. 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, the applications with manual WAIT data collection, and the applications with an automated WAIT. For each combination using WAIT, the *Sampling Interval* was configured to 480 seconds (a commonly used value) and 30 seconds (minimum value recommended for WAIT). The remaining test configurations were suggested by IBM SVT to reflect real-world conditions: A workload of 2,000 concurrent users; a duration of 1 hour; a *Hard Disk Threshold* of 100MB; and a *Time Threshold* of 10 minutes. Finally, each combination was repeated three times.

[11] http://sourceforge.net/projects/ibatisjpetstore/
[12] http://www-03.ibm.com/software/products/us/en/portalserver

For JPetStore, each test run produced around 500,000 transactions. The results presented in Table 1 showed that using WAIT with a *Sampling Interval* 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 *Sampling Interval* 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 *Sampling Interval*, 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 the automated WAIT. In terms of resource consumption, the differences between the WAIT modalities remained within 1%.

Table 1. JPetStore - Overhead Results

WAIT Modality	**Avg Response Time (ms)**	**Max Response Time (ms)**	**Avg Throughput (hps)**	**Avg CPU Usage (%)**	**Avg Memory Usage (MB)**
None (*Baseline*)	1889.6	44704.0	158.8	36.9	1429
Manual, 480s	0.0%	0.0%	0.0%	1.1%	3.0%
Automated, 480s	0.0%	0.0%	0.0%	2.0%	3.7%
Manual, 30s	1.6%	0.4%	-3.1%	1.47%	4.1%
Automated, 30s	4.4%	0.5%	-4.0%	2.53%	4.4%

For Portal, each test run produced around 400,000 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 *Sampling Interval* of 480 seconds. Also, the differences between the results for the two *Sampling Interval* 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 analyzing 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).

To explore the small differences between the runs and the potential environmental variations that were experienced during the experiments, a Paired t-Test [13] was done (using a significance level of $p<0.1$) to evaluate if the differences in response time and throughput were statistically significant. This analysis indicated that these differences were only only significant when using a *Sampling*

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

Table 2. Portal - Overhead Results

WAIT Modality	Avg Response Time (ms)	Max Response Time (ms)	Avg Throughput (hps)	Avg CPU Usage (%)	Avg Memory Usage (MB)
None (*Baseline*)	4704.75	40435.50	98.05	76.73	3171.20
Manual, 480s	0.7%	0.6%	-0.1%	0.63%	2.2%
Automated, 480s	3.4%	1.0%	-2.8%	1.13%	4.1%
Manual, 30s	14.9%	5.4%	-5.6%	2.23%	5.3%
Automated, 30s	16.8%	9.1%	-5.7%	2.97%	6.0%

Interval of 30 seconds. This analysis reinforced the conclusion that the overhead was low and the observation that the *Sampling Interval* 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 automated WAIT tool with a *Sampling Interval* 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 solution was reliable, as using the automated approach had minimal impact in terms of response time (0.5% average and 0.2% max) and throughput (1.4%). Moreover the consumption of resources behaved similarly to the single-node test (an increment of 0.85% in CPU and 2.3% in Memory).

[POSSIBLE MOVING THE PAIR T-TEST PARAGRAPH TO REMOVE THESE TWO LINES] A paired t-Test also indicated that these differences were not statistically significant when compared against the application alone.

In conclusion, the results of this experiment 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 *Sampling Interval* and the application behavior could have on the overhead, it is important to consider these factors in the configuration. In our case, a *Sampling Interval* of 480 seconds proved efficient in terms of overhead for the two tested applications using WAIT.

4.4 Experiment #2: Assessment of productivity benefits

Here the objective was to assess the benefits our approach brings to a performance tester. First, the source code of JPetStore was modified and three common performance issues were injected: A lock contention bug (composed of a very heavy calculation within a synchronized block of code), an I/O latency bug (composed of a very expensive file reading method) and a deadlock bug (composed of an implementation of the classic "friends bowing" deadlock example[14]). Then an automated WAIT monitored the application to assess how well it was able to identify the injected bugs and estimate the corresponding time savings in performance analysis. All set-up parameters were identical to the multi-node

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

test previously described except the duration which was one hour. Due to space constraints, only the most relevant sections of the WAIT reports are presented.

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 report from the tool and 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 by offering similar rankings. Fig. 11.a shows the results of the early report, while 11.b shows the results of the final report.

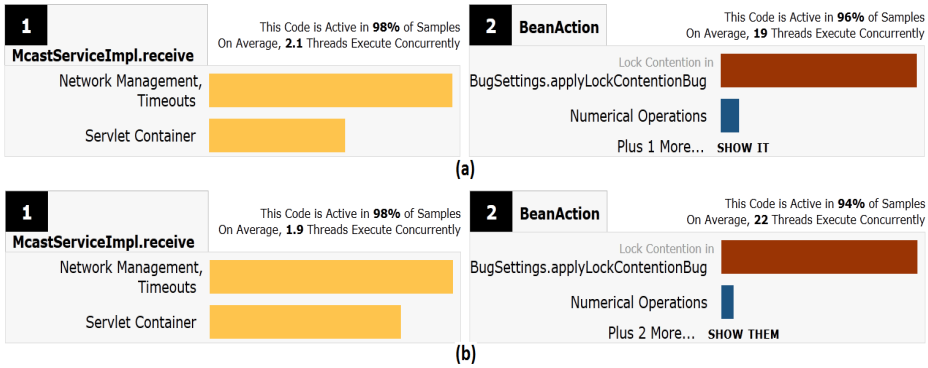

Fig. 11. Top detected performance issues in modified JPetStore application

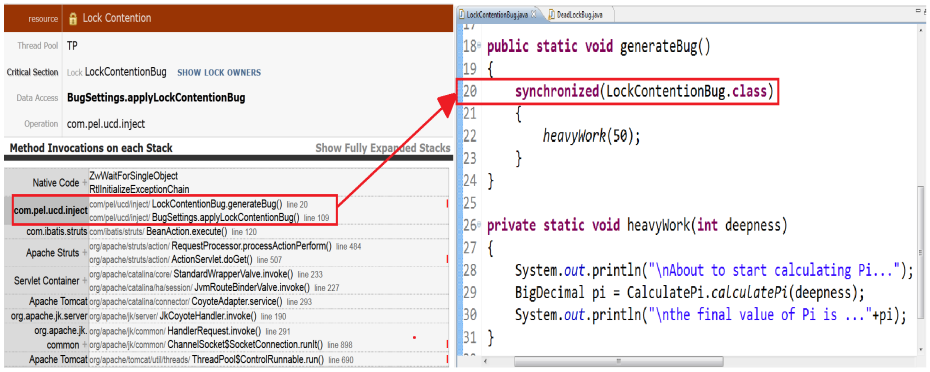

Fig. 12. Lock contention issue in the WAIT report and the actual source code

After identifying an issue, a tester can see more details, including the type of problem, involved class, method and method line. Fig. 12 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). In 3rd place the report showed a symptom of the lock contention issue, 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. Fig. 13 shows the details of these issues.

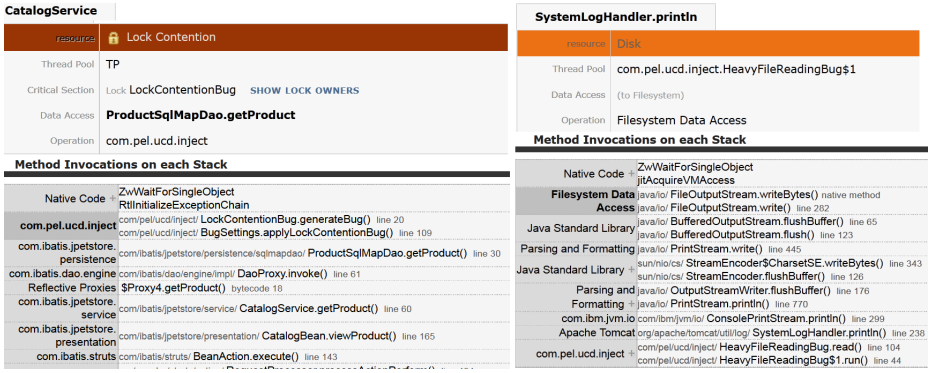

Fig. 13. Details of issues ranked 3rd and 4th

The deadlock issue 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. Fig. 14 shows the information of our Deadlock bug, which was located in the line 30 of the DeadLockBug class. This is precisely the line where the bug was injected (as the deadlock occurs when the friends bow back to each other).

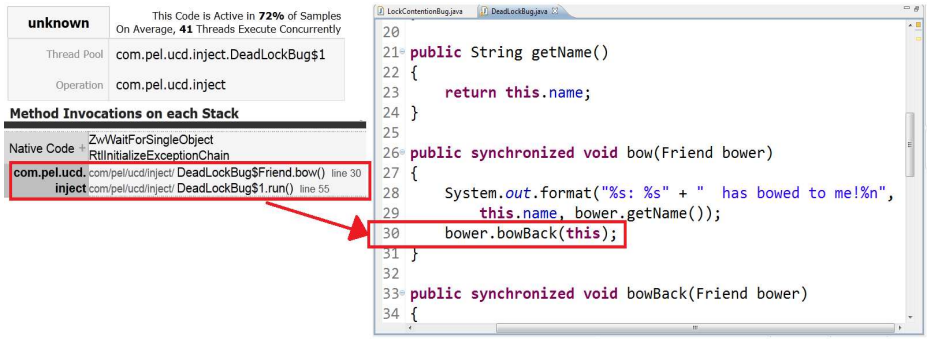

Fig. 14. Deadlock issue in the WAIT report and the actual source code

As all injected bugs were identified, including the involved classes and methods, 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 were a few seconds spent configuring it (i.e. to change the *Sampling Interval*). 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 WAIT also allowed to exploit WAIT's 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 analyzing 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 these experimental results, they were very promising because it was possible to measure the productivity benefits that a tester can gain by using WAIT through our proposed automation approach: After a quick installation (around 5 minutes per node), 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. A direct consequence of these time savings is the reduction in the dependence on human expert knowledge and a reduced effort required by a tester to identify performance issues, hence improving the productivity.

4.5 Threats to Validity

Like any empirical work, there are some threats to the validity of these experiments. First the possible environmental noise that could affect the test environments because they are not isolated. To mitigate this, multiple runs were executed for each identified combination. Another threat was the selection of the tested applications. Despite being real-world applications, their limited number implies that not all types of applications have been tested and wider experiments are needed to get more general conclusions. However, there is no reason to believe that the presented approach is not applicable to other environments.

5 Related Work

The idea of applying automation in the performance testing domain is not new. However, most of the research has focused on automating the generation of load test suites[14–20]. For example [15] proposes an approach to automate the generation of test cases based on specified levels of load and combinations of resources. Similarly, [18] presents an automation framework that separates the application logic from the performance testing scripts to increase the reusability of the test scripts. Meanwhile [20] presents a framework designed to automate the performance testing of web applications and which internally utilizes two usage models to simulate the users behaviors more realistically.

Regarding performance analysis, a high percentage of the proposed techniques require some type of instrumentation. For example, the authors in [8] 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 [9, 10] which rely on instrumentation to dynamically infer invariants within the

applications and detect programming errors; or the approach proposed by [11] which uses instrumentation to capture execution paths to determine the distributions of normal paths and look for any significant deviations in order to detect errors. In all these cases, the 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.

Moreover the authors of [21] present a non-intrusive approach which automatically analyzes the execution logs of a load test to identify performance problems. As this approach only relies on load testing results, it can not determine root causes. A similar approach is presented in [22] which aims to offer information about the causes behind the issues. However it can only provide the subsystem responsible of the performance deviation. On the contrary, our approach allows the applicability of the idle-time analysis in the performance testing domain through automation, which allows the identification of the classes and methods responsible for the performance issues. Moreover the techniques presented in [21, 22] require information from previous runs to baseline their analysis, information which might not always be available.

Finally, the authors of [23, 24] present frameworks to monitor software services. Both frameworks monitor the resource utilization and the component interactions within a system, but target different technologies ([23] focuses on Java and [24] on Microsoft technologies). Unlike these works, which have been designed to assist on operational support activities, our proposed approach has been designed to address the specific needs of a tester in the performance testing, isolating her from the complexities of an expert system.

6 Conclusions and Future Work

The identification of performance problems in highly distributed environments is complex and time-consuming. Even though researchers have been developing expert systems to simplify this task, various limitations exist in those tools that prevent their effective usage in performance testing. To address these limitations, this work proposed a novel approach to automate the usage of an expert system in a distributed testing 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 in distributed environments 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. Thus, the approach was shown to have a low overhead and to reduce the time required to analyze performance issues, thereby reducing the costs associated with performance testing.

Future work will focus on assessing the approach and its benefits through broader experiments with our industrial partner IBM with a special interest in the trade-off between the *Sampling Interval* and the nature of the applications.

It will also be investigated how best to exploit the functional information that can be obtained from a test environment to improve the idle-time analysis.

Acknowledgments

We would like to thanks Amarendra Darisa and Patrick O'Sullivan, from IBM SVT, as their experience in performance testing helped us through the scope definition and validation. 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).

References

1. Compuware: Applied Performance Management Survey. (2007)
2. Woodside, M., Franks, G., Petriu, D.C.: The Future of Software Performance Engineering. Future of Software Engineering (May 2007)
3. Parsons, T., Murphy, J.: Detecting Performance Antipatterns in Component Based Enterprise Systems. In: International Middleware Doctoral Symposium. (2008)
4. Angelopoulos, V., Parsons, T., Murphy, J., O'Sullivan, P.: GcLite: An Expert Tool for Analyzing GC Behavior. COMPSACW (July 2012)
5. Spear, W., Shende, S., Malony, A., Portillo, R., Teller, P.J., Cronk, D., Moore, S.: Making Perf. Analysis and Tuning Part of SDLC. HPCMP-UGC (2009)
6. Altman, E., Arnold, M., Fink, S., Mitchell, N.: Performance analysis of idle programs. ACM SIGPLAN Notices (October 2010)
7. Ammons, G., Choi, J.d., Gupta, M., Swamy, N.: Finding and Removing Performance Bottlenecks in Large Systems. In: ECOOP. (2004)
8. Yang, J., Evans, D., Bhardwaj, D., Bhat, T., Das, M.: Perracotta: mining temporal api rules from imperfect traces. ICSE (2008)
9. S. Hangal, M.: Tracking down software bugs using automatic anomaly detection. International Conference on Software Engineering (2002)
10. C. Csallner, Y.: Dsd-crasher: a hybrid analysis tool for bug finding. ISSTA (2006)
11. Chen, M.Y., Accardi, A., Kiciman, E., Lloyd, J., Patterson, D., Fox, A., Brewer, E.: Path-based failure and evolution management. NSDI (2004)
12. Shahamiri, S., Kadir, W., Mohd-Hashim, S.: A Comparative Study on Automated Software Test Oracle Methods, ICSEA (2009)
13. Wu, Haishan, Asser N. Tantawi, T.: A Self-Optimizing Workload Management Solution for Cloud Applications. (2012)
14. Albert, Elvira, Miguel Gmez-Zamalloa, J.: Resource-Driven CLP-Based test case generation. Logic-Based Program Synthesis and Transformation (2012)
15. M. S. Bayan, J.: Automatic stress and load testing for embedded systems. International Computer Software and Applications Conference (2006)
16. J. Zhang, S.: Automated test case generation for the stress testing of multimedia systems. Softw. Pract. Exper. (2002)
17. L. C. Briand, Y. Labiche, M.: Using genetic algorithms for early schedulability analysis and ST in RT systems. Genetic Prog. and Evolvable Machines (2006)
18. Chen, S., Moreland, D., Nepal, S., Zic, J.: Yet Another Performance Testing Framework. Australian Conference on Software Engineering (2008)
19. V. Garousi, L. C. Briand, Y.: Traffic-aware stress testing of distributed systems based on uml models. International conference on Software engineering (2006)
20. Xingen Wang, Bo Zhou, W.: Model-based load testing of web applications (2013)

21. Jiang, Z.M., Hassan, A.E., Hamann, G., Flora, P.: Automated performance analysis of load tests. In: International Conference on Software Maintenance. (2009)
22. Haroon Malik, Bram Adams, A.: Pinpointing the subsys responsible for the performance deviations in a load test. Software Reliability Engineering (2010)
23. van Hoorn, A., Rohr, M., Hasselbring, W., Waller, J., Ehlers, J., Frey, S., Kieselhorst, D.: Design and Application of the Kieker Framework. (2009)
24. Barham, P., Donnelly, A., Isaacs, R., Mortier, R.: Using magpie for request extraction and workload modelling. (2004)