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 a very challenging task. Specifically, the identification of performance issues and the diagnosis of their root causes are time-consuming and complex activities which usually require multiple tools and heavily rely on expertise. To simplify the above tasks, hence increasing the productivity and reducing the dependency on expert knowledge, many researchers have been developing tools with built-in expertise knowledge 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 case studies, using real-life applications, which assessed the overhead and time savings that the proposed approach 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, idle-time analysis, distributed environments, multi-tier applications

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 using the software system to achieve business goals. 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. Latest trends in the information technology world (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 expert knowledge 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 tasks, hence increasing the productivity and reducing the dependency on expert knowledge, many researchers have been developing tools with built-in expertise knowledge for non-expert users [6, 7, 4]. However, various limitations exist in these tools that prevent their usage in performance testing of highly distributed environments. The data collection usually needs to be controlled manually, which in a highly distributed environment (composed of multiple nodes to monitor and coordinate simultaneously) is very time-consuming and error-prone as this process produces a vast amount of data that needs to be collected and consolidated. This challenge is more complex when the data also needs to be processed periodically during the test execution to get an incremental view of the results (as a performance test usually lasts several days and a tester should not wait until the end of the test to determine if any performance issues occurred). The same situation occurs with the outputs, where a tester commonly needs to review multiple reports (possible one for each monitored node per data processing cycle). This situation reduces the usefulness of the reports, as a tester must manually correlate their findings.

Even though these limitations might be manageable in small testing environments, they prevent the 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

[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/

to manually coordinate the data gathering of memory snapshots and the generation of the tool's reports. These steps conducted periodically for all the nodes every hour, which produces a total of 2400 iterations. Moreover the tester would have to review the multiple reports she would get every hour to evaluate if any memory issues exist. As an alternative, she may focus the analysis 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 the previous challenges, the overhead generated by any technique should be low and steady through time 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. Moreover it usually requires the modification of the source code, which might not always be available such as in third-party components. Similarly, if a tool requires heavy human effort to be used, this might limit the applicability of that tool. On the contrary, automation could play a key role to encourage the adoption of a technique. As documented by the authors in [12], this strategy has proven successful in performance testing activities.

Finally, to ensure that our research work is helpful for solving real-life problems in the software industry, we have been working with our industrial partner, IBM System Verification Test (SVT), to understand the challenges that they experience in their day-to-day testing activities. The received feedback confirms that there is a real need to simplify the usage of expert tools so that testers can carry out analysis tasks in less time.

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 IBM Whole-system Analysis of Idle Time tool (WAIT)[7]. This publicly available tool is a lightweight expert system that helps to identify the main performance inhibitors that exist in Java systems. Our work was validated through two case studies using real-world applications. The first case study concentrated in evaluating the overhead introduced by our approach. The second assessed the productivity gains that the approach can bring to the performance testing process. The results provided evidence about the benefits of this 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. Also the introduced overhead was low (between 0% to 3% when using a common industry *Sampling Interval*).

The main contributions of this paper are:

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

1. A novel lightweight approach to automate the usage of expert systems in performance testing of distributed software systems.
2. A practical validation of the approach consisting of an implementation around the WAIT tool and two case studies using real-life applications.
3. Analysis of the overhead the approach has in the monitored environment.

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 that lead to 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 indicating a lack of motion. 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 or instrumentation are required to use it. Furthermore WAIT requires infrequent samples to perform its diagnosis, so it also 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 and obtain 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 that 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 number and type of threads. The bottom section shows all the performance inhibitors that have been identified, ranked by frequency and impact. Also each color indicates a different category of problem. 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 (yellow color) and was caused by database readings.

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

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

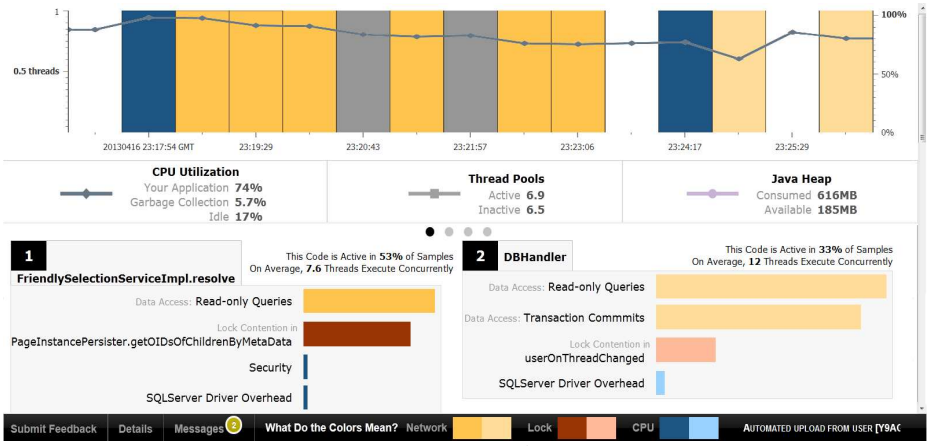

Fig. 1. Example of WAIT Report

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 expert system to get a consolidated output. This scenario is depicted in Fig. 2 where the automated logic shields the tester from the complexities of using the ES, so that she only needs to interact with her load testing tool during the whole duration of a performance test.

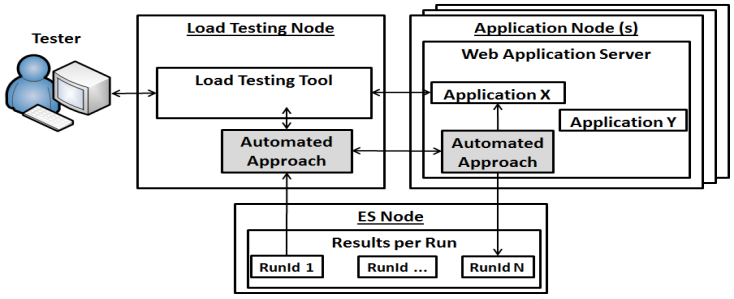

Fig. 2. Contextual View of the Proposed Approach

Moreover the detailed approach is depicted in the Fig. 3. It will start right after the performance test has started, requiring some additional inputs:

- The list of nodes to be monitored.
- The *Sampling Interval* to control how often the samples will be collected.
- The *Backup* flag to indicate if the collected data should be backed up before any cleaning occurs.

– The *Time Threshold* to define the maximum time between data uploads.
– The *Hard Disk Threshold* to define the maximum storage quota for collected data (to prevent its uncontrolled growth).

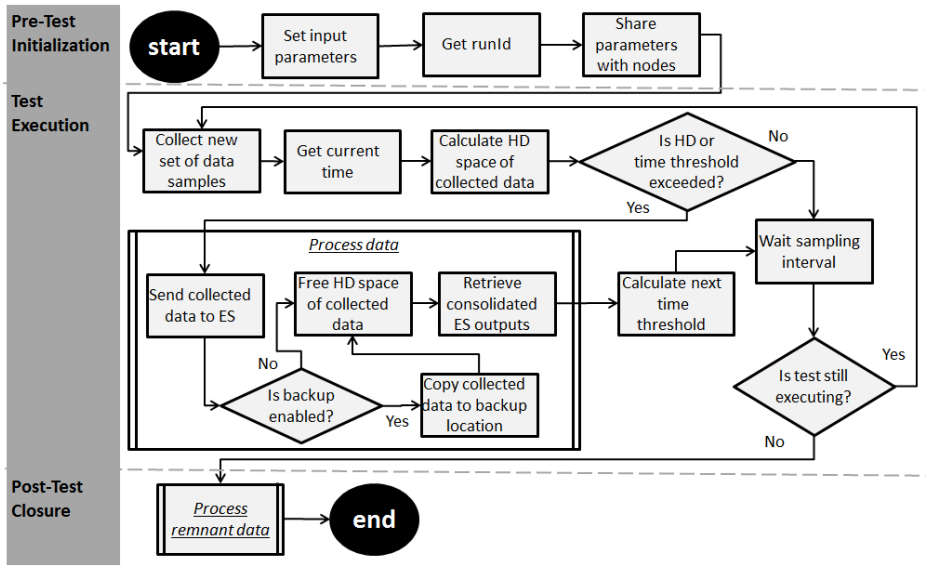

Fig. 3. Process Flow - Automation approach

The process starts, in parallel to the performance test, by initializing the configured parameters. Then it gets a new *RunId*, value which will uniquely identify this test run and all its collected data and reports. 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 make the process adaptable to different usages, by allowing a tester to control the pace of data collection and data feeding to an ES. For example, one tester might prefer to get updated results as soon as new data is available. In this case, she could define a very low *Hard Disk Usage Threshold* or a very small *Next Time Threshold*. Moreover it is not uncommon that an ES require some processing time to generate its outputs. In this scenario, bigger thresholds would be preferable.

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 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 free space and keep the HD usage below the threshold. As some data collection processes could be very time-consuming (i.e. the generation of a memory dump in Java can take several minutes and take hundreds of megabytes of HD), this step could be very useful as it might allow further off-line analysis of the collected data without affecting the actual test

run. Then updated outputs from the expert system are retrieved and the *Next 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 this information is also processed by the expert system. Lastly this data is also cleared and the final consolidated 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 to know when the performance 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 steps in each application node.

¡¡ - Change architecture overview to component/class diagram? (similar to Kikier) - Does the Load Testing Tool need to understand the outputs of the ES? No ... clarify the role of the Control Agent: maybe describe that Control would have interfaces for LTT, while Node per ES. - Probably dig a little deeper in the interfaces (only required to interact with the LTT); which usually have extension points to integrate additional resources. ¿¿

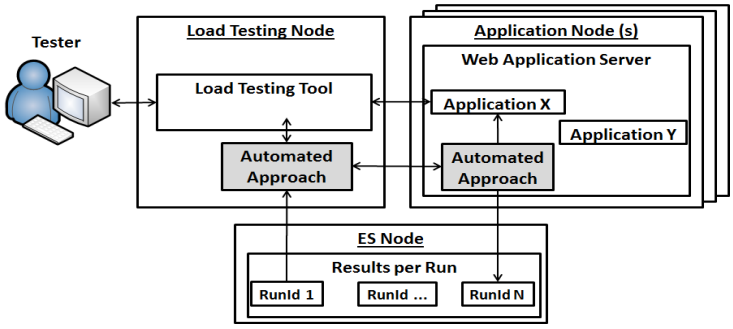

Fig. 4. High-level architecture of automated approach

These components communicate through commands, following the *Command* Design Pattern[9]. Here the *Control Agent* invokes the start and stop commands based on the changes of status in the Load Testing tool, while the *Node Agent* implements the logic in charge of executing each concrete command. This logic includes sending back the result of the performed operation to the *Control Agent*.

An example of the distinct interactions that occur between these components are depicted in Fig. 5. Once a tester has started a performance test (step 1), the *Control Agent* propagates the action to each of 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).

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

These processes continue iteratively until the test ends. At that moment, the *Control Agent* propagates the stop action to all *Node Agents* (steps 21,22 and 24). At any time during the test execution, the tester might choose to review the intermediate results obtained from the expert system (steps 12 to 14) until getting the final results (steps 25 to 27).

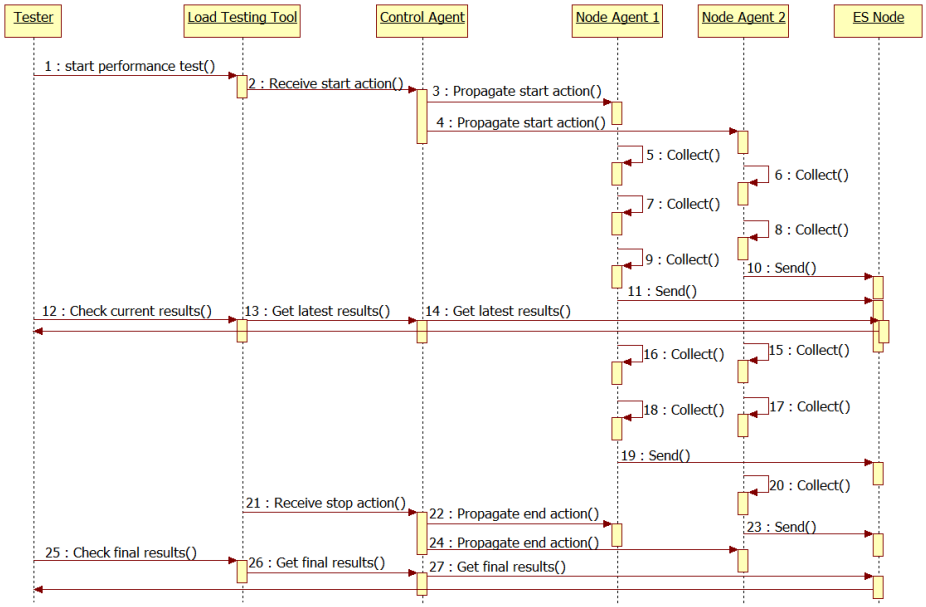

Fig. 5. Sequence diagram of the automated approach

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 an Eclipse Plugin for the Rational Performance Tester (RPT) [10], which is a load testing tool commonly used 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 previously discussed in Section 2.

Once the agents are installed in an environment, WAIT can be configured as any other resource in RPT as shown in Fig. 6. Similarly, during a performance test WAIT can be monitored as any other resource in the *Performance Report* of RPT under the *Resource View* as depicted in Fig. 7, which shows some of the metrics that a tester can check: The number of monitored processes, the number of WAIT data collections that have been triggered and the collections that have occurred. Finally, the consolidated WAIT report is also accessible within RPT, so a tester does not need to leave RPT during the whole performance test. This is shown in Fig. 8.

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

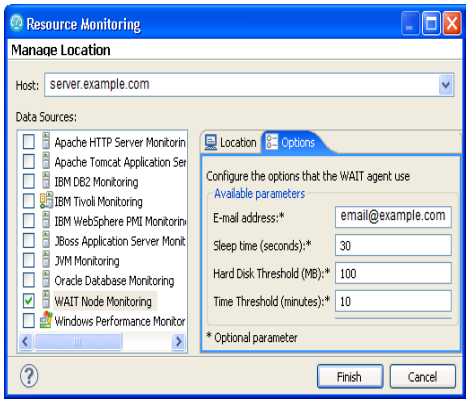

Fig. 6. WAIT configuration in RPT

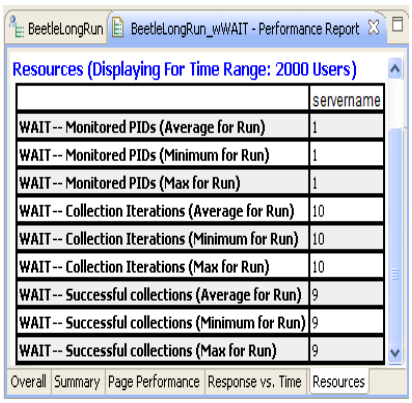

Fig. 7. WAIT monitored in RPT

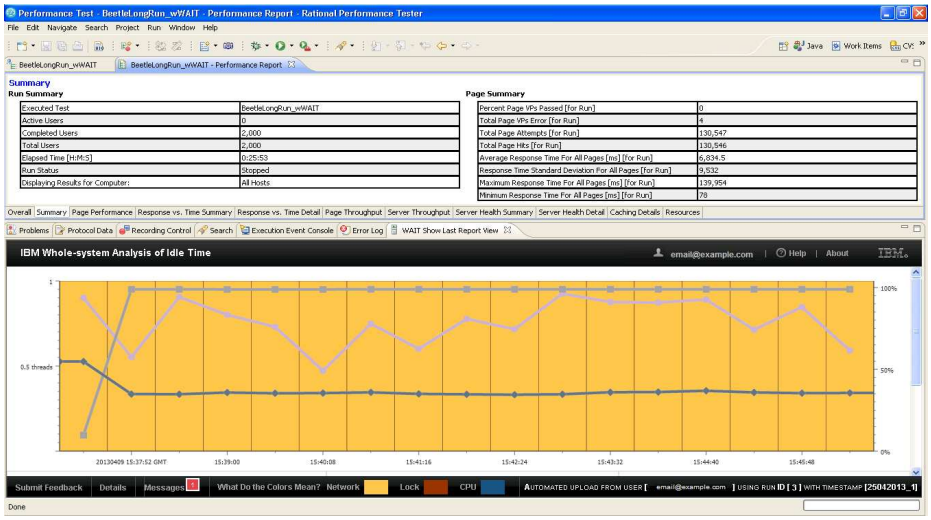

Fig. 8. WAIT Report accessible in RPT

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. 9. 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.

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

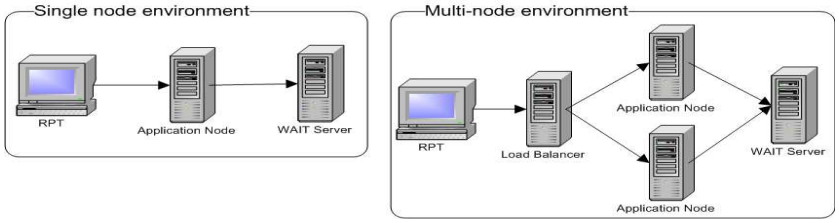

Fig. 9. Environment Configurations

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 Pet Store, 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. [14] 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.
- 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 configuration was 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.

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 different 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 in this case (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

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

cost was insignificant in the higher *Sampling Interval*, with 30 seconds the impact was visible. The difference in response times (2.8%, around 53 milliseconds) was caused by the upload and backup processes (around 4MB of data every 10 minutes), as the cost of the *Node Agent* presence had been previously measured. 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%	-4.0%	1.47%	4.1%
Automated, 30s	4.4%	0.5%	-3.1%	2.53%	4.4%

For Portal, each test run produced around 400,000 transactions and the results are presented in Table 2. These 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 the differences in behavior of Portal to be quantified: 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), with a maximum generation time of 3 sec (100% bigger than JPetStore's).

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%	1.13%	2.2%
Automated, 480s	3.4%	1.0%	-2.8%	0.63%	4.1%
Manual, 30s	14.9%	5.4%	-5.7%	2.97%	5.3%
Automated, 30s	16.8%	9.1%	-5.6%	2.23%	6.0%

¡¡ - Stress the overhead (manual WAIT) against which matters to measure? !!!- Regenerate the Pair t-Test using the real data*** (avg/max rt-¿rt, avg thr-¿thr) + 0.05 ¿¿

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 for JPetStore the only significant differences existed in the average response time and the average throughput when using a *Sampling Interval* of 30 seconds. Similar results were obtained from Portal. 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). A paired t-Test also indicated that the differences in response time and throughput between the runs were not significant.

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.
- 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 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.

Surprisingly the 1st ranked issue was none of the injected bugs but a method named "McastServiceImpl.receive" which appeared in practically all the samples.

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

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

Further analysis determined this method call was benign and related to the clustering functionality of Tomcat. The 2nd ranked issue was the lock contention. It is worth noting that both issues were detected in 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. 10.a shows the results of the early report, while 10.b shows the results of the final report.

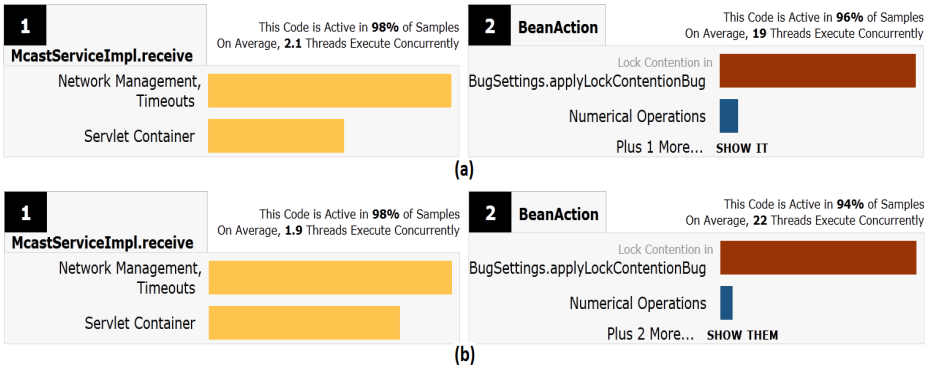

Fig. 10. Top detected performance issues in modified JPetStore application

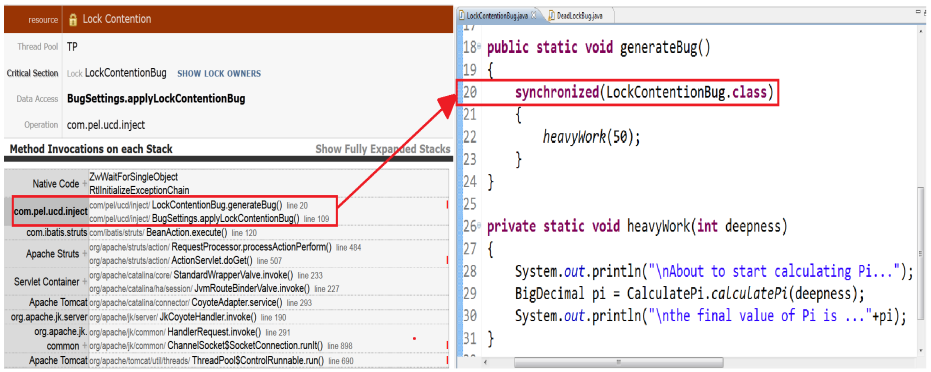

Fig. 11. Lock contention issue in the WAIT report and the actual source code

After identifying an issue, a tester can see more details, including the type of problem, involved class, method and method line. Fig. 11 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 that 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 this was a major problem (the issues were correlated by comparing their detailed information, which pinpointed to the same class and method). Finally, the I/O latency bug was identified in 4th place. Fig. 12 shows the details of these issues.

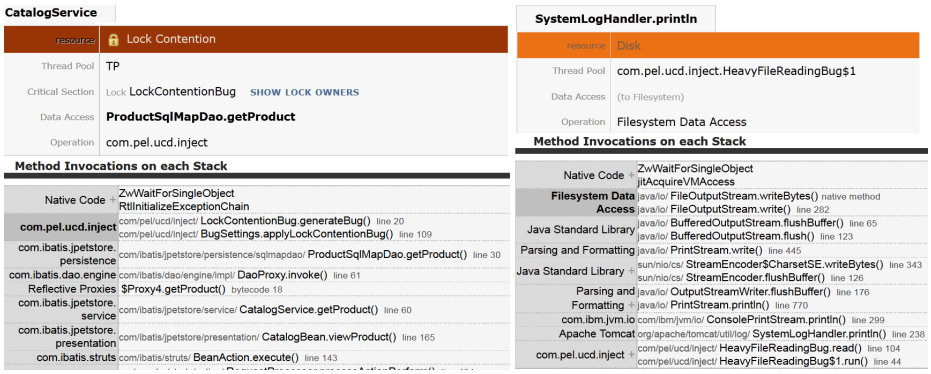

Fig. 12. Details of issues ranked 3rd and 4th

The deadlock issue did not appear in this test run, somehow prevented by the lock contention bug which had a bigger impact that 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. 13 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).

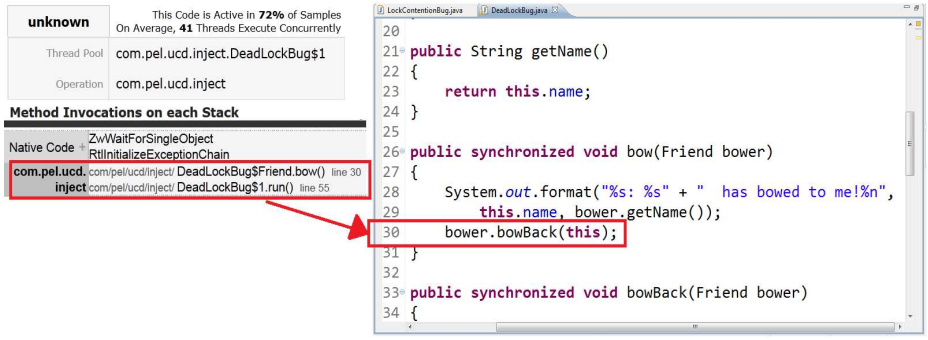

Fig. 13. Deadlock issue in the WAIT report and the actual source code

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 case study, 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 also 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 reduced the 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[15–22]. For example [16] proposes an approach to automate the generation of test cases, hence simplifying this time-consuming task, based on specified levels of load and combinations of resources. Similarly, [21] presents an automation framework that separates the application logic from the performance testing scripts.

Regarding performance analysis, a high percentage of the proposed techniques require instrumentation. For example, the authors in [8] instrument the source code to mine the sequences of call graphs 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 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 to detect errors. In all these cases, instrumentation would obscure the performance of an application during performance testing hence discouraging their usage. This is especially relevant in load and stress testings,

where it is common to use huge workloads, which might trigger non-lineal overheads in the instrumentation. On the contrary, our proposed approach does not require any instrumentation.

Moreover the authors of [23] 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 [24] 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 [23, 24] require information from previous runs to baseline their analysis, information which might not always be available. Finally, the authors of [25] presents a framework for monitoring software services. The main difference between this work and ours is that our proposed approach has been designed to address the specific needs of performance testing, isolating a tester from the complexities of an expert system, while [25] has been designed for operational support.

6 Conclusions and Future Work

The identification of performance problems and their root causes in highly distributed environments are complex and time-consuming tasks. Even though researchers have been developing expert systems to simplify these tasks, various limitations exist 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 test environment. A prototype was developed around the WAIT expert tool and then its benefits and overhead were assessed. The results showed that for a *Sampling Interval* of 480 seconds, the automation caused a negligible performance degradation of the JPetstore application. For the Portal, it caused a 3.4% increase in average response time and a 2.8% reduction in throughput. The results also showed the time savings gained by applying the approach to an expert tool. In our case, the effort to utilize WAIT, which used to depend on the number of nodes and the sampling frequency, was reduced to seconds regardless of the configuration. This optimization also simplified the identification of performance issues. In our case study, 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 provide an easily accessible summary of the system performance in a single report. It therefore has the possibility of reducing the time required to analyze performance issues, thereby reducing the effort and costs associated with performance testing.

Future work will concentrate on assessing the approach and its benefits through broader case studies 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 in-

formation that can now be obtained from a test environment (i.e. workload, response time, throughput) to improve the idle-time analysis.

¡¡ Possible future directions: - Make it open source to evaluate how useful it might become and/or apply it to MAT. - Take actions based on rules? (i.e. raise a bug or email if data is not flowing, adaptive thresholds). ¿¿

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., Terpstra, D.: Making Perf. Analysis and Tuning Part of Software Development Cycle. High Perf. Computing Modernization Program Users Group (June 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. J. Yang, D. Evans, D.T.M.: Perracotta: mining temporal api rules from imperfect traces. International conference on Software engineering (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. International symposium on Software testing and analysis (2006)
11. M. Y. Chen, A. Accardi, E.J.A.E.: Path-based failure and evolution management. Symposium on Networked Systems Design and Implementation (2004)
12. Shahamiri, S.R., Kadir, W.M.N.W., M.: 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. Gootzit, David, Gene Phifer, R.: Magic Quadrant for Horizontal Portal Products. Technical report, Gartner Inc. (2008)
15. Albert, Elvira, Miguel Gmez-Zamalloa, J.: Resource-Driven CLP-Based test case generation. Logic-Based Program Synthesis and Transformation (2012)
16. M. S. Bayan, J.: Automatic stress and load testing for embedded systems. 30th Annual International Computer Software and Applications Conference (2006)
17. J. Zhang, S.: Automated test case generation for the stress testing of multimedia systems. Softw. Pract. Exper. (2002)

18. L. C. Briand, Y. Labiche, M.: Using genetic algorithms for early schedulability analysis and ST in RT systems. Genetic Prog. and Evolvable Machines (2006)
19. A. Avritzer, E.: Generating test suites for software load testing. ACM SIGSOFT international symposium on Software testing and analysis (1994)
20. A. Avritzer, E.: The automatic generation of load test suites and the assessment of the resulting software. IEEE Trans. Softw. Eng. (1995)
21. Chen, S., Moreland, D., N.Z.: Yet Another Performance Testing Framework. Australian Conference on Software Engineering (2008)
22. V. Garousi, L. C. Briand, Y.: Traffic-aware stress testing of distributed systems based on uml models. International conference on Software engineering (2006)
23. Jiang, Z.M., Hassan, A.E., Hamann, G., Flora, P.: Automated performance analysis of load tests. In: International Conference on Software Maintenance. (2009)
24. Haroon Malik, Bram Adams, A.: Pinpointing the subsys responsible for the performance deviations in a load test. Software Reliability Engineering (2010)
25. Hoorn, V., Rohr, M., Hasselbring, W., Waller, J., Ehlers, J., Kieselhorst, D.: Continuous Monitoring of Software Services: Design and Application of the Kieker Framework. (2009)