Towards an Automated Approach to Apply Idle-time Analysis in Performance Testing

A. Omar Portillo-Dominguez[1], Miao Wang[1], Philip Perry[1], John Murphy[1], Nick Mitchell[2], and Peter F. Sweeney[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}@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 the expertise of the engineers. WAIT, a tool that implements idle-time analysis, has proven successful in simplifying the above tasks, hence increasing the productivity and reducing the dependency on expert knowledge. However WAIT has some usability limitations that prevent its efficient usage in performance testing. This paper presents a lightweight approach that addresses those limitations and automate WAIT's usage, making it useful in performance testing. This work was validated through two case studies with real-life applications, assessing the proposed approach in terms of overhead costs and time savings in the analysis of performance issues. The current results have proven the benefits of the approach by achieving a good decrement in the time invested in performance analysis while only generating 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 performance plays a central role in usability. However it is not uncommon that performance issues occur and materialize into serious problems (i.e. outages on production environments, or even cancellation of projects) in a significant percentage of 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 single aspect of the design, code, and execution environment of an application. Latest trends in the information technology world (such as Service Oriented Architecture and Cloud Computing) 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. It is also common that multiple sources are required to diagnose performance problems, especially in highly distributed environments. For instance in Java thread dumps, garbage collection logs, heap dumps, CPU utilization and JVM memory usage are a few examples of the information that a tester could need to understand the performance of an application. This situation increases the expertise required to do performance analysis, which is usually held by only a small number of testers within an organization. This could potentially lead to bottlenecks where some activities can only be done by experts, hence impacting the productivity of large testing teams.

In addition to the previous challenges, the overhead generated by any technique should be low to minimize the impact it has in the tested environment, otherwise the technique would not be suitable for performance testing. 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 [5], this strategy has proven successful in performance testing activities.

To ensure that our research work is helpful for solving real-life problems in the software industry, we have carried out regular meetings with our industrial partner, the IBM System Verification Test (SVT), to discuss the challenges that they experience in their day-to-day testing activities. The received feedback confirms that there is a real need to have tools that can help to improve the productivity of the testing teams by allowing testers with less expertise to carry out analysis tasks in less time.

The Whole-system Analysis of Idle Time[3] tool (WAIT) has proven successful in simplifying the detection of performance issues and their root causes in Java environments [6, 7]. WAIT is an attractive candidate to the performance testing domain because it has minimal impact in the monitored environment by using a lightweight monitoring approach that does not require instrumentation. However WAIT has usability limitations that prevent its effective usage in performance testing: The overall data collection process needs to be controlled manually, which in a highly distributed environment (composed of multiple nodes to monitor and coordinate simultaneously) would be very time-consuming and error-prone, specially if the data needs to be updated periodically during the test execution to have an incremental view of the results. The same situation occurs with the output of WAIT, where a tester needs to review multiple reports, one for each node per update. Even though these limitations might be manageable in small testing environments, they prevent WAIT to be effectively use in big-

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

ger testing environments as the time and effort required to synchronize WAIT execution and analyze its outputs would overcome the possible benefits of its usage. As a highly distributed environment is precisely the scenario where WAIT's analysis capabilities to diagnosis performance issues would be more valuable, these usage limitations must be addressed.

This paper proposes a lightweight automated approach that addresses the above limitations to use WAIT effectively in the performance testing domain, while keeping WAIT's strengths of low overhead and minimal intrusion. This work was achieved through a two-phase experiment using two real-world applications. The first phase concentrated in validating that the overhead introduced by our approach was low. The second phase assessed the productivity benefits that an automated WAIT bring to the performance testing process. The current results have provided evidence about the benefits of the approach: The overhead in the monitored system was low (between 0% to 3% when using a common industry *Sampling Interval*). Regarding time savings, using the automated approach did not add extra effort to the tester and the time she required to analyze WAIT's outputs was reduced. Now a tester only needs to review a single WAIT report instead of multiple individual reports on a node basis. Moreover this consolidated WAIT report allowed to quickly identify the performance issues, also pinpointing the responsible classes and method calls. In summary, the main contribution of this paper is a lightweight approach to automate the data gathering and execution of WAIT in sync with a performance test loader to make WAIT usable in performance testing.

The rest of this paper is structured as follows: Section 2 discusses the background. Section 3 shows the related work. Section 4 describes the problem definition. Section 5 explains the proposed approach, while Section 6 explores the experimental evaluation and results. Finally Section 7 presents the conclusions and future work.

2 Background

Idle-time analysis is a methodology that pursues to explain 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. WAIT is based on non-intrusive sampling mechanisms available at Operating System level (i.e. "ps" and "vmstat" commands in a Unix/Linux environment) and the Java Virtual Machine (JVM), in the form of *Javacores* [4] (diagnostic feature of Java 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. Moreover WAIT requires infrequent samples to perform its diagnosis, so it also has low overhead.

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

From a usability perspective, WAIT is simple: A user only needs to collect as much data as desired, upload it to a 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, thread pools and Java Heap) and the number and type of threads per sample. The bottom section shows all the performance inhibitors that have been identified, ranked by frequency and impact. Moreover each category of problem 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. The affected resource was the network (highlighted in yellow) and was caused by database readings.

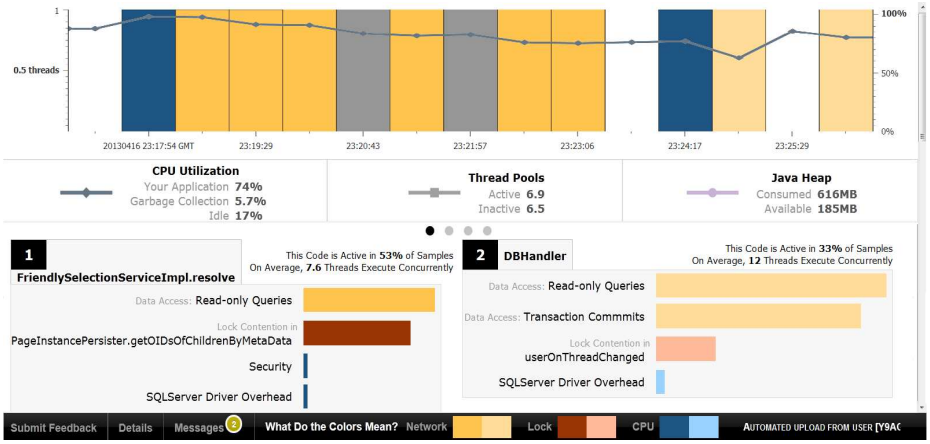

Fig. 1. Example of WAIT Report

3 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 [8–15]. Regarding performance analysis, a high percentage of the proposed techniques require instrumentation. For example, the authors in [16] instrument the source code to mine the sequences of call graphs to infer any relevant error patterns. A similar case occurs with the work presented in [17, 18] which rely on instrumentation to dynamically infer invariants and detect programming errors; or the approach proposed by [19] 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 cases, instrumentation would obscure the performance of an application during performance testing hence discouraging their usage. On the contrary, our proposed approach does not require instrumentation.

One of the closest works to ours is [20]. Here the authors present a non-intrusive approach which automatically analyzes the execution logs of a load test

to identify performance problems. As this approach only analyzes load testing results, it can not determine root causes. Another approach related to our work is presented in [21] which aims to offer information about the causes behind the issues. However it only limits to provide the subsystem responsible of the performance deviation. In both cases, the techniques also require information from previous runs as baseline to do their analysis, information which might not always be available. To the best of our knowledge, our approach is the first to propose the applicability of the idle-time analysis in the performance testing domain through automation means.

4 Problem Definition

While doing performance testing in highly distributed environments, the identification of performance problems and the diagnosis of their root causes are very complex and time-consuming activities. Furthermore they tend to rely on the expertise of the involved engineers. Given its strengths, WAIT is a promising candidate to improve this scenario by reducing the expert knowledge and time required to do performance analysis. However, WAIT has some usability limitations that prevent its effective applicability in performance testing: The effort required to do manual data collection to feed WAIT and the number of WAIT reports a tester would need to review are practically lineal with respect of the number of nodes in the application and the frequency with which the data is refreshed. For example, consider a relatively small environment composed of 10 nodes, a 24-hour test run and an uploading interval of 1 hour. A tester would need to coordinate to manually stop the data collection process, upload the data to generate the WAIT report and then start the data collection again. These steps conducted for the 10 nodes in cycles of 1 hour, which throws a total of 240 cycles. Additionally, these steps would need to be carried out as fast as possible to minimize the time gaps between the end of a cycle and the start of the next. Lastly, the tester would have to review the 10 different WAIT reports she would get every hour and compare them with the previous ones to evaluate if there are any performance issues. As it can be inferred from this example, the current cost of using WAIT in a highly distributed environment would outweigh the benefits, as it would be a very effort-intensive and error-prone process.

The objective of this paper is to address the usability limitations of WAIT to be used effectively in performance testing. To achieve this, the following research questions have been formulated:
Q1. How can we minimize the effort required to use WAIT in performance testing?
Q2. Can the overhead, introduced by the proposed approach, be low enough such that it does not compromise the results during a performance test execution?
Q3. What are the productivity benefits that a tester can gain by using the proposed approach?

The following sections show how these questions were answered by our approach and validated through a series of experiments.

5 Proposed Approach and Implementation

Our approach proposes the automation of the manual processes involved in the usage of WAIT. It will execute jointly to a performance test, periodically uploading the collected samples to get a consolidated WAIT report.

The approach is depicted in the Algorithm 1 and requires a few inputs: The list of nodes to be monitored by WAIT; a *Sampling Interval* to control how often the samples will be collected; a *Time Threshold* to define the maximum time between uploads; a *Hard Disk Threshold* to define the maximum storage quota for collected data (to prevent its uncontrolled growth); and a *Backup* flag that indicates if the collected data should be backed up before any cleaning occurs.

The process starts by getting a new *RunId*, value which will uniquely identify the test run and its reports. This value is propagated to all the nodes. On each node, the *Hard Disk Usage* and the *Next Time Threshold* are initialized. Then each node starts in parallel the following loop until the performance test finishes: A new set of data samples is collected (composed of machine and process utilization and a *Javacore* from each running JVM). After the collection finishes, it is assessed if any of the two thresholds has been reached (either the *Hard Disk Usage* has exceeded the *Hard Disk Threshold* or the *Next Time Threshold* has been reached). If any of the conditions has occurred, a new upload round occurs where the data is uploaded to the *WAIT Server* (labeling the data with the *RunId* so that uploads of different nodes are 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. Then a reference to the updated WAIT report is obtained and the *Next Time Threshold* is calculated. Finally, the logic awaits the configured *Sampling Interval* before a new iteration starts.

Once the performance test finishes, a final upload round (and backup if configured) is done to integrate any remnant collected data. Then this data is also cleared and the final consolidated WAIT report is obtained.

The previous approach was complemented with the proposed architecture shown in Fig. 2. It is composed of two main components: The *WAIT 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 *WAIT Node Agent* which is responsible of the collection, upload, backup and cleanup steps in each application node.

Based on the concepts presented here, a prototype has been developed in conjunction with our industrial partner IBM. The *WAIT Control Agent* was implemented as an Eclipse Plugin for the Rational Performance Tester (RPT) [5] because it is a tool that IBM SVT testers use regularly in their daily activities. This decision facilitates the adoption of the proposed approach. Similarly, the *WAIT Node Agent* was implemented as a Java Web Application because this is another skill set well known within IBM SVT. Once agents have been installed, WAIT can now be configured as any other resource in RPT as shown in Fig. 3.

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

Algorithm 1: Automated WAIT

Input: A set AN of n application nodes, Sampling Interval $Sint$, Time Threshold $tThres$, Hard Disk Threshold $hdThres$, Backup Flag $bFlag$. If $bFlag$ = true, Backup destination path $bPath$.
Output: Consolidated WAIT report

```
// Initialization
```
obtain $runId$ ←new RunId
share $runId$ with all nodes
foreach *node in AN (in parallel)* **do**
    $hdUsage \leftarrow 0$
    $nextTimeThreshold$ ←current time from the Operating System + $tThres$
    `// Main Process`
    while *performance test is executing* **do**
        `// Data gathering`
        collect new set of samples (process and machine utilization as well as a new javacore per JVM process in the node)
        `// Update performance results`
        $currentTime$ ←current time from the Operating System
        $hdCurrentUsage$ ←calculate Hard Disk space of collected data
        if $currentTime \geq nextTimeThreshold$ *or* $hdCurrentUsage \geq hdThres$ **then**
            upload locally collected data indicating it as part of test $runId$
            if $bFlag = true$ **then**
                copy locally collected data to $bPath$ indicating it as part of test $runId$
            delete locally collected data
            retrieve consolidated WAIT report
            $nextTimeThreshold$ ←current time from the Operating System + $tThres$
        wait $Sint$ before performing next iteration of the process
    `// Closure`
    upload remnant locally collected data indicating it as part of test $runId$
    if $bFlag = true$ **then**
        copy remnant locally collected data to $bPath$ indicating it as part of test $runId$
    delete remnant locally collected data
retrieve final consolidated WAIT report

Similarly, during a performance test WAIT can be monitored as any other resource in the *Performance Report* of RPT under the *Resource View* depicted in Fig. 4. Finally, the consolidated WAIT report is also accessible within RPT, so a tester does not need to leave RPT during the whole duration of the performance test. This is shown in Fig. 5.

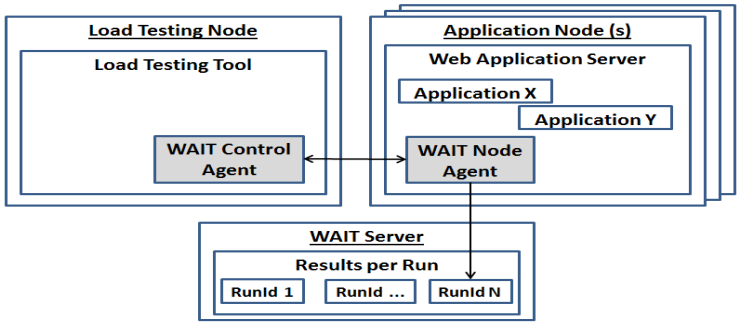

Fig. 2. High-level Architecture of automated WAIT

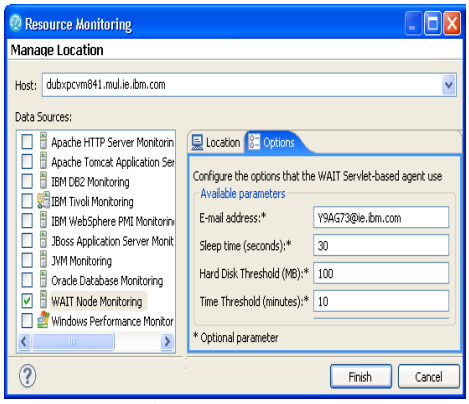

Fig. 3. WAIT configuration in RPT

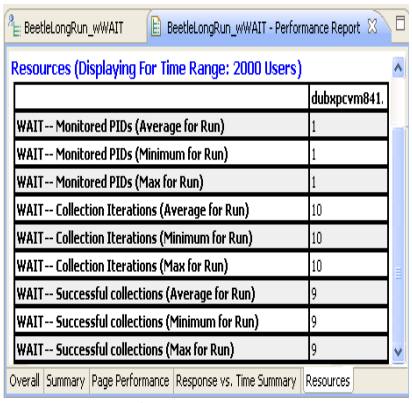

Fig. 4. WAIT monitored in RPT

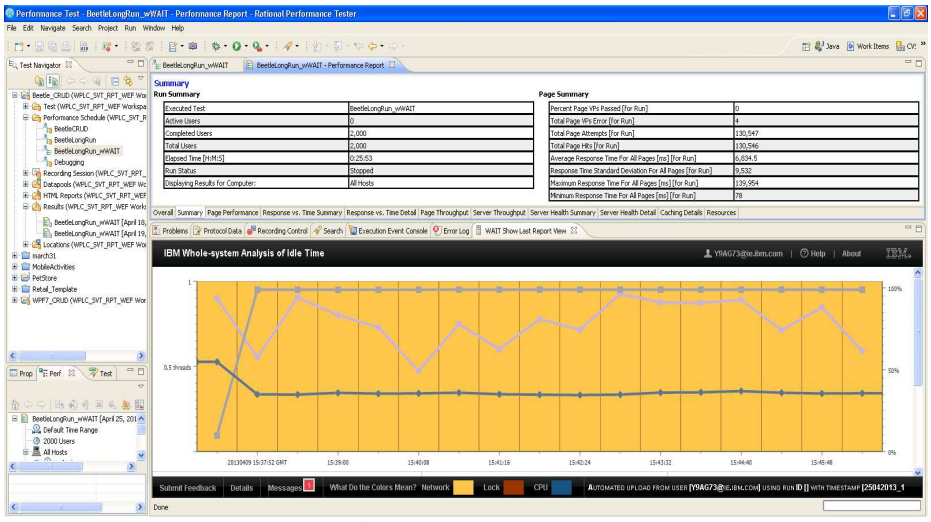

Fig. 5. WAIT Report accessible in RPT

6 Experimental Evaluation and Results

Two experiments were performed. The first one aimed to answer the research question Q2 (Can the overhead, introduced by the proposed approach, be low

enough such that it does not compromise the results during a performance test execution?), while the second one pursued to answer the research question Q3 (What are the productivity benefits that a tester can gain by using the proposed approach?). The combined outcome of the experiments pursued to answer the research question Q1 (How can we minimize the effort required to use WAIT in performance testing?).

Moreover two environment configurations were used: One was composed of one RPT node, one application node and one *WAIT Server* node; the other was composed of one RPT node, one load balancer node, two application nodes and one *WAIT Server* node. All connected by a 10-GBit LAN. These set-ups are shown in Fig. 6.

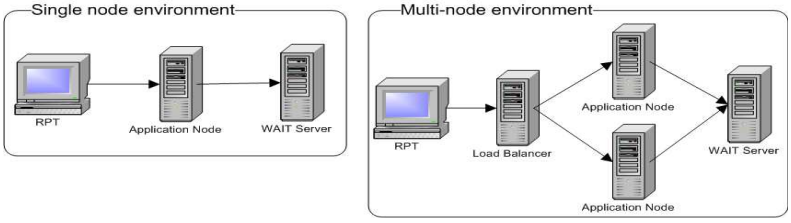

Fig. 6. Environment Configurations

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.

6.1 Experiment #1: Overhead Evaluation

Its objective was to validate that the proposed approach had low overhead 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, 2 real-world applications were used: iBatis JPetStore 4.0 [6] which is an upgraded version of Sun's Pet Store, an e-commerce shopping cart application. It ran over an Apache Tomcat 6.0.35 Application Server. The other application was IBM WebSphere Portal Server 8.0.1 [7], a leader solution in the enterprise portal market [22]. It ran over an IBM WebSphere Application Server 8.0.0.5.

Firstly, the overhead of the approach was measured in a single-node environment. For each application, 3 combinations of WAIT were evaluated: The application alone to get a baseline; the application with manual WAIT data collection; and the application with automated WAIT. For each combination

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

using WAIT, the *Sampling Intervals* were configured to 480 seconds (suggested by IBM SVT) and 30 seconds (minimum value recommended for WAIT). The remaining configuration parameters 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 3 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 *WAIT Node Agent* because the uploaded data was very small in this case (around 200KB every 10 minutes). When WAIT used a *Sampling Interval* of 30 seconds, the impacts in response time and throughput appeared. Considering the throughput was similar between the WAIT modalities, the impact was caused by the *Javacore* generation (step shared between the modalities). In 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 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 *WAIT 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. Even though the results presented in Table 2 showed similar trends in terms of having lower overheads using the *Sampling Interval* of 480 seconds, a few key differences were identified: First, the impacts in response time and throughput were visible since the *Sampling Interval* of 480 seconds. Besides, the differences between *Sampling Intervals* were bigger. As the experiment 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), 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%

Due to the small differences among the runs and the variations (presumable environmental) that were experienced during the experiments, a Paired t-Test [8], was done (using a significant level of $p<0.1$) to evaluate if the differences in response time and throughput were statistically significant. The results presented in Table 3 showed that for JPetStore the only significant differences existed in the average response time and the average throughput (automated WAIT) 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.

Table 3. Paired t-Test Results

Application	**WAIT Modality**	**Avg Response Time (ms)**	**Max Response Time (ms)**	**Avg Throughput (hps)**
JPetStore	Manual, 480s	0.470	0.143	0.206
JPetStore	Automated, 480s	0.342	0.297	0.472
JPetStore	Manual, 30s	0.089	0.241	0.154
JPetStore	Automated, 30s	0.019	0.334	0.078
Portal	Manual, 480s	0.140	0.263	0.496
Portal	Automated, 480s	0.040	0.189	0.131
Portal	Manual, 30s	0.001	0.158	0.167
Portal	Automated, 30s	0.013	0.105	0.072

A second test was done to validate that the overhead of the proposed approach remained low when used in a multi-node environment over a longer test run. This test used JPetStore and the automated WAIT 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 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

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

response time (0.5% in the average and 0.2% in the max) and throughput (1.4%). Moreover the consumption of resources behaved similar to the single-node test (an increment of 0.85% in CPU and 2.3% in Memory).

The performed paired t-Test also indicated that the differences in response time and throughput between the test runs were not statistically significant.

In conclusion, the results of this first experiment showed that the overhead caused by the automated approach remained low. This answered our research question Q2 positively (Can the overhead, introduced by the proposed approach, be low enough such that it does not compromise the results during a performance test execution?), but with a side note: Due to the impact that the *Sampling Interval* and the application behavior could have in the overhead, it is important to consider these factors when using WAIT. In our case, a *Sampling Interval* of 480 seconds proved efficient in terms of overhead for the two tested applications.

6.2 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 3 common performance issues were injected: A lock contention bug, composed by a very heavy calculation within a synchronized block of code; a I/O latency bug, composed by a very expensive file reading method; and a deadlock bug, an implementation of the classic "friends bowing" deadlock example[9]. Then the automated WAIT was used to monitor this 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 performed with exception of the duration which was reduced to 1-hour. Due to space constraints, only the most relevant sections of the WAIT report 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. Further analysis determined this method call was benign and related to the clustering functionality of Tomcat. The 2nd ranked issue was the lock contention. A relevant point to highlight is that both issues were detected since the early versions of the report. Based on their high frequency (above 96% of the samples), this information 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 ranking. Fig. 7.a shows the results of the early report, while 7.b shows the results of the final report.

After identifying an issue, a tester can see more details, including the type of problem, involved class, method and method line. Fig. 8 shows the information of our Lock Contention bug, which was located in the LockContentionBug class, 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).

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

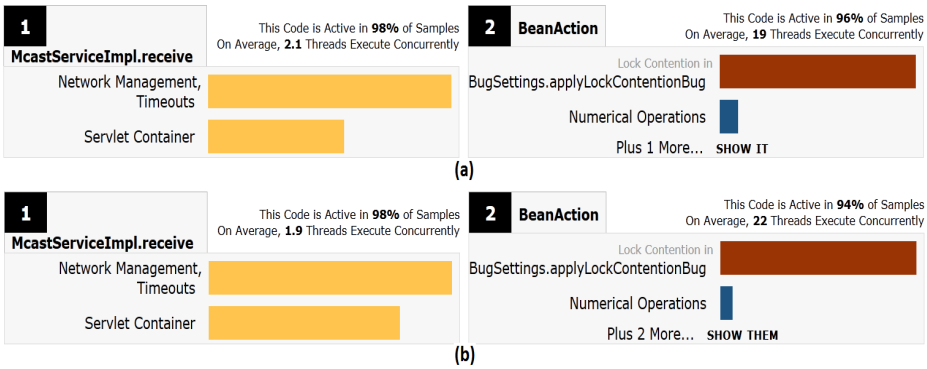

Fig. 7. Top detected performance issues in modified JPetStore application

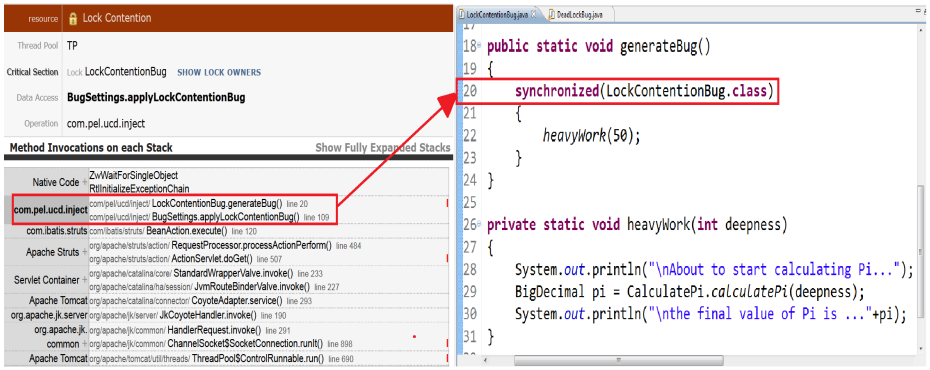

Fig. 8. Lock contention issue in the WAIT report and the actual source code

In 3rd place the report showed a symptom of the lock contention issue (the issues were correlated by comparing their detail information, which pinpointed to the same class/method), suggesting this was a major issue. The I/O latency bug was identified in 4th place. Fig. 9 shows the details of these issues.

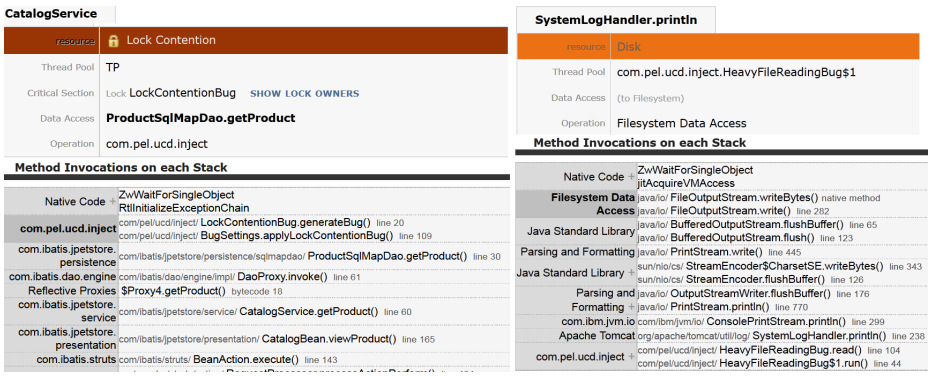

Fig. 9. Details of issues ranked 3rd and 4th

The deadlock issue did not appear in this test run, somehow prevented by the lock contention bug which had a major impact that planned. As in any common

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. 10 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).

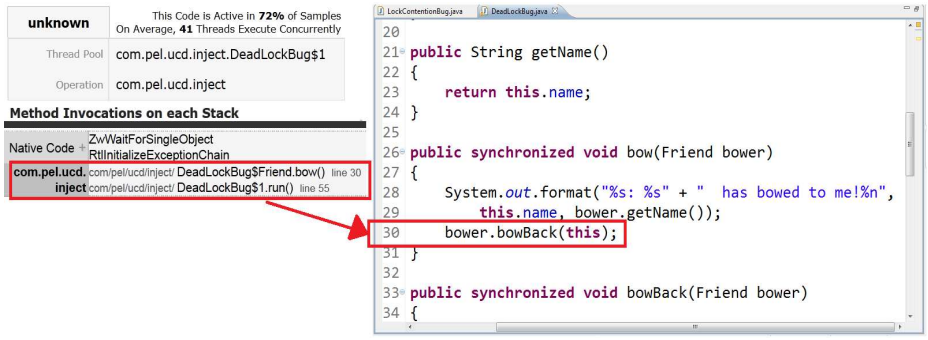

Fig. 10. Deadlock issue in the WAIT report and the actual source code

This experiment was considered successful as all injected bugs were identified, including the involved classes and methods. In terms of time, two main savings were documented. First, the automated approach practically reduced the effort of collecting data for WAIT to zero. After a one-time installation which took no more than 15 minutes for all nodes, the only additional time required to use the automate approach were a few seconds required to modify its configuration (i.e. to change the *Sampling Interval*). The second time saving was 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 usability constraints of WAIT also allow a tester 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 (i.e. 2 hours per bug) could help to quantify WAIT's savings. In our study case, instead of spending an estimate of 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 relevant issues are reported to developers in parallel to the test execution. This is especially valuable in long-term runs, common in performance testing and which usually last several days.

In conclusion, the results of this second experiment answered our research question Q3 (What are the productivity benefits that a tester can gain by using the proposed approach?), showing how a tester's productivity improves by using an automated WAIT.

To summarize the experimental results, they were satisfactory because it was possible to achieve the goal of automating the execution of WAIT, while keeping the overhead low (in the range of 0% to 3% using a *Sample Interval* of 480 seconds). Additionally the automated approach brought several time savings to

a tester: After a quick installation (around 5 minutes per node), the time required to use an automated WAIT was minimal. Also a tester now only needs to monitor a single WAIT report, which offers a consolidated view of the results. A direct result of these savings is the reduction in effort and expert knowledge required by tester to identify performance issues, hence improving the productivity.

6.3 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 approach is not applicable to other environments.

7 Conclusions and Future Work

The identification of performance problems and the diagnosis of their root causes in highly distributed environments are complex and time-consuming tasks, which tend to rely on the expertise of the involved engineers. The objective of this work was to address the limitations that prevented the usage of WAIT in performance testing, so that it can be effectively used to reduce the expert knowledge and effort required to identify performance issues and their root causes. To achieve this goal the paper presented a novel automation approach and its validation, composed of a prototype and study cases with two real-life applications. The results are encouraging as they have proved that the approach addresses effectively the adoption barriers of WAIT in performance testing: The solution has proven lightweight, generating low overhead (in the range of 0% to 3% using a *Sampling Interval* commonly used in the industry). Moreover, there are also tangible time savings in terms of the effort required to detect performance issues.

Future work will concentrate on assessing the approach and its benefits through broader study cases with our industrial partner IBM with a special interest in understanding the trade-off between the *Sampling Interval* and the nature of the applications (represented by their *Javacores*). It will also be investigated how best to exploit the functional information that can now be obtained from a tested environment (i.e. workload, throughput or transactions) to improve the qualitative and quantitative capabilities of the idle-time analysis to identify more types of performance problems.

Acknowledgments

We would like to thanks Amarendra Darisa, from IBM SVT, as his 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 (FOSE '07) (May 2007) 171–187
3. Parsons, T., Murphy, J.: Detecting Performance Antipatterns in Component Based Enterprise Systems. In: 2nd International Middleware Doctoral Symposium. Volume 7. (2008) 55–90
4. Angelopoulos, V., Parsons, T., Murphy, J., O'Sullivan, P.: GcLite: An Expert Tool for Analyzing Garbage Collection Behavior. 2012 IEEE 36th Annual Computer Software and Applications Conference Workshops (July 2012) 493–502
5. Shahamiri, S.R., Kadir, W.M.N.W., M.: A Comparative Study on Automated Software Test Oracle Methods. ICSEA (2009)
6. Altman, E., Arnold, M., Fink, S., Mitchell, N.: Performance analysis of idle programs. ACM SIGPLAN Notices **45**(10) (October 2010) 739
7. Wu, Haishan, Asser N. Tantawi, T.: A Self-Optimizing Workload Management Solution for Cloud Applications. (2012)
8. Chen, S., Moreland, D., N.Z.: Yet Another Performance Testing Framework. Australian Conference on Software Engineering (ASWEC) (2008)
9. Albert, Elvira, Miguel Gmez-Zamalloa, J.: Resource-Driven CLP-Based test case generation. Logic-Based Program Synthesis and Transformation (2012)
10. J. Zhang, S.: Automated test case generation for the stress testing of multimedia systems. Softw. Pract. Exper. (2002)
11. L. C. Briand, Y. Labiche, M.: Using genetic algorithms for early schedulability analysis and stress testing in RT systems. Genetic Programming and Evolvable Machines (2006)
12. M. S. Bayan, J.: Automatic stress and load testing for embedded systems. 30th Annual International Computer Software and Applications Conference (2006)
13. A. Avritzer, E.: Generating test suites for software load testing. ACM SIGSOFT international symposium on Software testing and analysis (1994)
14. A. Avritzer, E.: The automatic generation of load test suites and the assessment of the resulting software. IEEE Trans. Softw. Eng. (1995)
15. V. Garousi, L. C. Briand, Y.: Traffic-aware stress testing of distributed systems based on uml models. International conference on Software engineering (2006)
16. J. Yang, D. Evans, D.T.M.: Perracotta: mining temporal api rules from imperfect traces. International conference on Software engineering (2008)
17. S. Hangal, M.: Tracking down software bugs using automatic anomaly detection. International Conference on Software Engineering (2002)
18. C. Csallner, Y.: Dsd-crasher: a hybrid analysis tool for bug finding. International symposium on Software testing and analysis (2006)
19. M. Y. Chen, A. Accardi, E.J.A.E.: Path-based failure and evolution management. Symposium on Networked Systems Design and Implementation (2004)
20. Jiang, Z.M., Hassan, A.E., Hamann, G., Flora, P.: Automated performance analysis of load tests. In: IEEE International Conference on Software Maintenance, Ieee (September 2009) 125–134
21. Haroon Malik, Bram Adams, A.: Pinpointing the subsys responsible for the performance deviations in a load test. Software Reliability Engineering (ISSRE) (2010)
22. Gootzit, David, Gene Phifer, R.: Magic Quadrant for Horizontal Portal Products. Technical report, Gartner Inc. (2008)