Towards an Automated Approach to Apply the 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 the idle-time analysis, has proven successful in simplifying the identification of performance issues and their root causes, hence reducing the dependency to expert knowledge and increasing the productivity. However WAIT has some usability limitations that prevent its efficient usage in performance testing. This paper presents an light-weight 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 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

Quality plays a major role in the successful adoption of any software. Moreover it has a direct impact in the total cost of ownership. For example, a 2008 Quality Assurance study [1] documented that achieving high quality generates cost savings of around 40%. It is also an accepted fact 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 software projects. For example, a 2007 survey applied to information technology executives [2] 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 [3–5], is that current performance tools heavily rely on expert knowledge to understand their output. It is also common that multiple data sources and tools 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. Similarly a tester could use jhat [3] to review the heap dumps while jconsole [4] to check memory and threads usages. 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 those limited experts, hence impacting the overall productivity of large testing teams.

In addition to the previous challenges, the overhead generated by any technique should be low to minimize the impact its usage has in the tested environment in terms of response time and throughput, otherwise the technique would not be suitable for performance testing. Similarly, if a tool requires heavy human effort to be used effectively, 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 of [6], this strategy has proven successful in perform testing activities.

To ensure that our research work is helpful for solving real-life problems in the software industry, we have also carried out regular meetings with 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 performance of the testing teams by allowing testers with less expertise to carry out complex analysis tasks in less time.

WAIT [5] (an expert system that performs idle-time analysis) has proven successful in simplifying the detection of performance issues and their root causes in Java environments [7, 8]. WAIT is an attractive candidate to the performance

[3] http://docs.oracle.com/javase/6/docs/technotes/tools/share/jhat.html

[4] http://docs.oracle.com/javase/1.5.0/docs/guide/management/jconsole.html

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

testing domain because it has minimal impact in the monitored environment by using a light-weight 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, especially if the data needs to be refreshed periodically during the test execution to have an incremental view of the results. This is highly desirable in long runs (i.e. 5 days or more) so that a tester does not need to wait until the end of a test run to know if any performance issues exist. Even though these limitations might be manageable in small testing environments or short test runs, they prevent WAIT to be effectively use in bigger testing environments as the time and effort required to synchronize WAIT execution would overcome the possible benefits of its usage, especially considering that this process would be very error-prone. As a highly distributed environment is precisely the scenario where WAIT's analysis capabilities would be more valuable to simplify the identification and diagnosis of performance issues, 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, 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 remained 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 *Sample Interval*). Regarding time savings, using the automated approach added practically no extra effort to the tester and the time required by a tester to analyze WAIT's outputs was reduced. A tester now only needs to review a single WAIT report instead of multiple individual reports on a node basis. Moreover the consolidated WAIT report allowed to quickly identify the presence of performance issues, also also pinpointing the responsible classes and method calls. In conclusion, 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, including an empirical validation through a case study on real-world applications.

The rest of this paper is structured as follows: Section 2 discusses the background. Section 3 shows the related work. Section 4 summarizes the problem definition. Section 5 describes 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 an analysis methodology that pursues to explain the root causes that lead to under-utilized resources. This approach, proposed by [7], is based on the observed behavior that in multi-tier applications performance problems usually manifest as idle time indicating a lack of forward motion. This can be caused by multiple factors, such as resource constraints, locking problems or excessive system-level activities.

WAIT is a tool that implements the idle-time analysis and does performance triage, identifying the main performance inhibitors that exist on a system. WAIT is based on non-intrusive sampling mechanisms available at the 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* [6], (diagnostic feature of Java to get a quick snapshot of the JVM state, offering information such as threads, locks, monitors, and memory). The fact that WAIT uses these standard data sources makes it non-disruptive, as no special flags, service restart or instrumentation are required to use it. Moreover WAIT requires infrequent samples to perform its diagnosis, so it also has low overhead. This characteristic is especially relevant considering that the JVM has to pause its running applications whenever a *javacore* is generated. Even though a JVM can produce *javacores* with a fairly low perturbation, these pauses might go up to several hundred milliseconds if the application has thousands of concurrent threads and very deep call stack traces.

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. 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). It also shows the number and type of threads per sample in a timeline. The bottom section shows all the performance inhibitors that have been identified, ranking them by frequency and impact. 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 was caused by database readings.

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 [9–16]. Regarding research in performance analysis, a high percentage of the proposed techniques require instrumentation. For example, the authors of [17] 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 [18, 19] which rely on instrumentation to dynamically inferred invariants and detect programming errors; or the approach proposed by [20] which uses instrumentation to capture execution paths to determine the distributions of normal paths

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

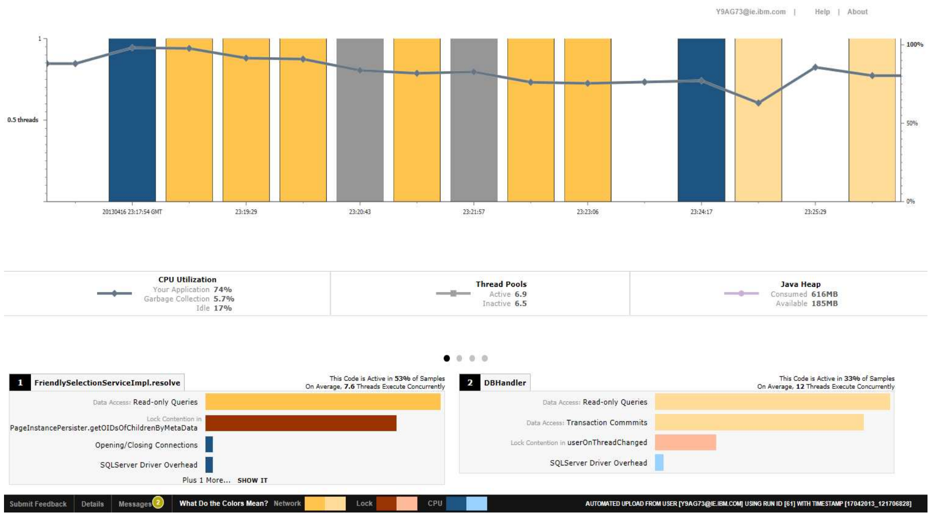

Y9AG73@ie.ibm.com | Help | About
1
0.5 threads
100%
50%
0%
20130416 23:17:54 GMT
23:19:29
23:20:43
23:21:57
23:23:06
23:24:17
23:25:29
CPU Utilization
Your Application 74%
Garbage Collection 5.7%
Idle 17%
Thread Pools
Active 6.9
Inactive 6.5
Java Heap
Consumed 616MB
Available 185MB
1 FriendlySelectionServiceImpl.resolve
This Code is Active in 53% of Samples
On Average, 7.6 Threads Execute Concurrently
Data Access: Read-only Queries
Lock Contention in PageInstancePersister.getOIDsOfChildrenByMetaData
Opening/Closing Connections
SQLServer Driver Overhead
Plus 1 More... SHOW IT
2 DBHandler
This Code is Active in 33% of Samples
On Average, 12 Threads Execute Concurrently
Data Access: Read-only Queries
Data Access: Transaction Commmits
Lock Contention in userOnThreadChanged
SQLServer Driver Overhead
Submit Feedback
Details
Messages 2
What Do the Colors Mean? Network Lock CPU
AUTOMATED UPLOAD FROM USER [Y9AG73@IE.IBM.COM] USING RUN ID [61] WITH TIMESTAMP [17042013_121706828]

Fig. 1. Example of WAIT Report

and look for any statistically 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 [21]. 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. The other work which is closest to ours is [22] which pursues to offer information about the causes behind the issues. However it can only determine which subsystem is 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. Finally, to the best of our knowledge our approach is the first to propose the applicability, through automation means, of the idle-time analysis in the performance testing domain to exploit its performance analysis capabilities.

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, which tend to rely on the expertise of the involved engineers. Due to its strengths, WAIT is a promising candidate that can help 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 usage 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 application nodes and the frequency with which the data needs to be refreshed. For example, assuming a relatively small environment composed of 10 application nodes, a 24-hour test run and a refresh frequency of 1 hour, a tester would need to coordinate 10 simultaneous data collection and upload cycles per hour (for a total of 240 cycles), doing them as fast as possible to minimize the time gaps between the end of a cycle and the start of the next. Additionally, the tester should review the 10 different WAIT reports he would get per hour to see if there are any performance issues. As it can be inferred from the above example, the cost of using WAIT in a highly distributed environment would overcome 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 the usage of WAIT be automated to minimize the effort required to use it in sync with a performance test?
Q2. Can the overhead be kept low during a performance test execution to avoid compromising the results?
Q3. What benefits in productivity can a tester achieve if the previous two questions are answered?

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 is depicted in the Algorithm 1 and requires a few inputs: The list of nodes that will be monitored (usually all the nodes that composed the tested system); a *Sampling Interval* to control how often the samples will be collected; a *Time Threshold* which indicates the maximum time without refreshing the results; a *Hard Disk Threshold* which indicates the maximum space to store collected data per node; 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 uniquely identifies the test run and its results. This value is then propagated to all the nodes. On each node, the *Hard Disk Usage* and the *Next Time Threshold* are initialized. Then each node starts the following loop until the performance test finishes: First 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 occurs where the data is uploaded to the WAIT server (labeling the upload with the *RunId* so that all uploads are identified as part of the same test run). Besides 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*

Algorithm 1: Proposed Approach

Input: A finite set $A = \{a_1, a_2, \ldots, a_n\}$ of application nodes, Sampling Interval *Sint*, Time Threshold *tThres*, Hard Disk Threshold *hdThres*, Backup Flag *bFlag*. If *bFlag* = true, Backup path *bPath*.

Output: Incremental WAIT report for all Nodes

```
   // Initialization
1  $rundId$ ←new RunId
2  share $rundId$ with all nodes
3  for $i \leftarrow 1$ to $n$ nodes do
4      $hdUsage \leftarrow 0$
5      $nextTimeThreshold$ ←current time from the Operating System + $tThres$
       // Main Process
6      while Performance Testing is executing do
           // Data gathering
7          Collect new set of samples (process and machine utilization as well as a
           new javacore per JVM process in the node)
           // Refresh performance results
8          $currentTime$ ←current time from the Operating System
9          $hdCurrentUsage$ ←calculate Hard Disk space of collected data
10         if $currentTime > nextTimeThreshold$ or $hdCurrentUsage >$
           $hdThres$ then
11             update locally collected data indicating it as part of test $rundId$
12             if $bFlag = true$ then
13                 copy locally collected data to $bPath$ indicating it as part of test
                   $rundId$
14             deleted locally collected data
15             retrieve updated performance results
16             $nextTimeThreshold$ ←current time from the Operating System +
               $tThres$
17         Wait $Sint$ before performing next iteration of the process
       // Closure
18     update remnant locally collected data indicating it as part of test $rundId$
19     if $bFlag = true$ then
20         copy remnant locally collected data to $bPath$ indicating it as part of
           test $rundId$
21     deleted remnant locally collected data
22 Retrieve final performance results
```

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 upload any remnant collected data. When completed this data is also cleared and the **final performance results report** is obtained.

Even though the above approach has been defined to address the usability needs of WAIT, it should be noted that its structure is flexible enough to be easily adjusted to fit automation scenarios of similar characteristics.

To achieve a lightweight automation, the previous approach was complemented with the architecture presented in the Fig. 2. It is composed of two main components. The *WAIT Control Agent* will be placed in the machine where the Load Testing tool is installed. This component will be responsible of interacting with the Load Testing tool to know when to start and stop the overall process. It is also responsible of getting the runId and propagate it to all the nodes. The second component is the *WAIT Node Agent*, which will be in each application node. It will be responsible of the collection, upload, backup and cleanup steps. Two key assumptions were considered when defining the above design. As the

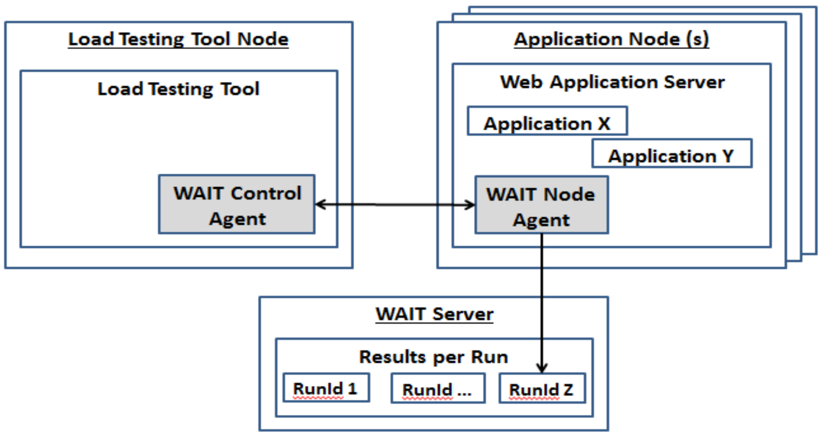
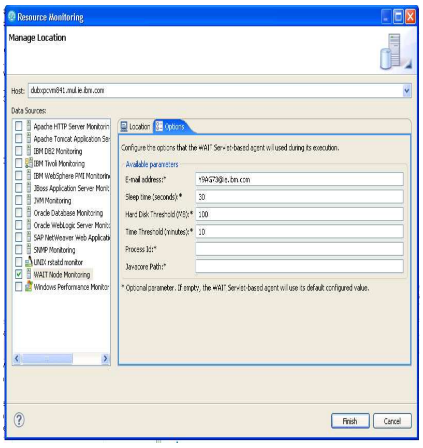


Fig. 2. High-level Architecture of the solution

scope of this work is the performance testing domain, it is assumed that a Load Testing tool will always be present. Moreover this work has also focused on Web Applications, which is a traditional Java business niche. For this reason, it is assumed that there will always be a Web Application Server in each application node. This assumption allows the *WAIT Node Agent* to be a Web Application. As it uses plain HTTP request to interact with the *WAIT Control Agent*, a single version of *WAIT Node Agent* could easily interact with multiple types of *WAIT Control Agent* (as it will be very likely to have one per type of Load Testing Tool) or even used independently.

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 a Eclipse Plugin for the Rational Performance Tester (RPT) [7], while the *WAIT Control Agent* was implemented as a Java Web Application. In both cases, the involved technologies (Eclipse Plugin and Web Application) were selected because they are simply to install.

Once installed, WAIT can now be configured as any other resource in RPT. This is shown in Fig. 3. Similarly, once the performance test has started, WAIT can now be monitored as any other resource in the *Performance Report* of RPT under the *Resource View*. This is shown in the Fig. 4. Finally, the WAIT report (which is initially created once the first upload is received and then updated after every data upload) is also accessible within RPT, so that the tester does not need to leave RPT during the whole duration of the performance test. This is shown in Fig. 5.



Manage Location

Host: dubxpcvm841.mul.ie.ibm.com

Data Sources:

- [ ] Apache HTTP Server Monitorin
- [ ] Apache Tomcat Application Ser
- [ ] IBM DB2 Monitoring
- [ ] IBM Tivoli Monitoring
- [ ] IBM WebSphere PMI Monitorin
- [ ] JBoss Application Server Monit
- [ ] JVM Monitoring
- [ ] Oracle Database Monitoring
- [ ] Oracle WebLogic Server Monit
- [ ] SAP NetWeaver Web Applicati
- [ ] SNMP Monitoring
- [ ] UNIX rstatd monitor
- [✓] WAIT Node Monitoring
- [ ] Windows Performance Monitor

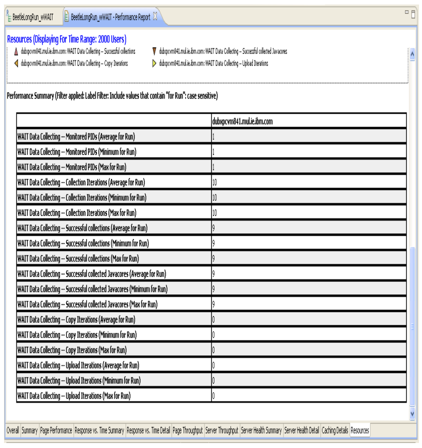
Configure the options that the WAIT Servlet-based agent will used during its execution.

Available parameters

E-mail address:*	Y9AG73@ie.ibm.com
Sleep time (seconds):*	30
Hard Disk Threshold (MB):*	100
Time Threshold (minutes):*	10
Process Id:*	
Javacore Path:*	

* Optional parameter. If empty, the WAIT Servlet-based agent will use its default configured value.

Fig. 3. WAIT configuration in RPT



Resources (Displaying For Time Range: 2000 Users)

dubxpcvm841.mul.ie.ibm.com: WAIT Data Collecting -- Successful collections
dubxpcvm841.mul.ie.ibm.com: WAIT Data Collecting -- Copy Iterations
dubxpcvm841.mul.ie.ibm.com: WAIT Data Collecting -- Successful collected Javacores
dubxpcvm841.mul.ie.ibm.com: WAIT Data Collecting -- Upload Iterations

Performance Summary (Filter applied: Label Filter: Include values that contain "for Run": case sensitive)

	dubxpcvm841.mul.ie.ibm.com
WAIT Data Collecting -- Monitored PIDs (Average for Run)	1
WAIT Data Collecting -- Monitored PIDs (Minimum for Run)	1
WAIT Data Collecting -- Monitored PIDs (Max for Run)	1
WAIT Data Collecting -- Collection Iterations (Average for Run)	10
WAIT Data Collecting -- Collection Iterations (Minimum for Run)	10
WAIT Data Collecting -- Collection Iterations (Max for Run)	10
WAIT Data Collecting -- Successful collections (Average for Run)	9
WAIT Data Collecting -- Successful collections (Minimum for Run)	9
WAIT Data Collecting -- Successful collections (Max for Run)	9
WAIT Data Collecting -- Successful collected Javacores (Average for Run)	9
WAIT Data Collecting -- Successful collected Javacores (Minimum for Run)	9
WAIT Data Collecting -- Successful collected Javacores (Max for Run)	9
WAIT Data Collecting -- Copy Iterations (Average for Run)	0
WAIT Data Collecting -- Copy Iterations (Minimum for Run)	0
WAIT Data Collecting -- Copy Iterations (Max for Run)	0
WAIT Data Collecting -- Upload Iterations (Average for Run)	0
WAIT Data Collecting -- Upload Iterations (Minimum for Run)	0
WAIT Data Collecting -- Upload Iterations (Max for Run)	0

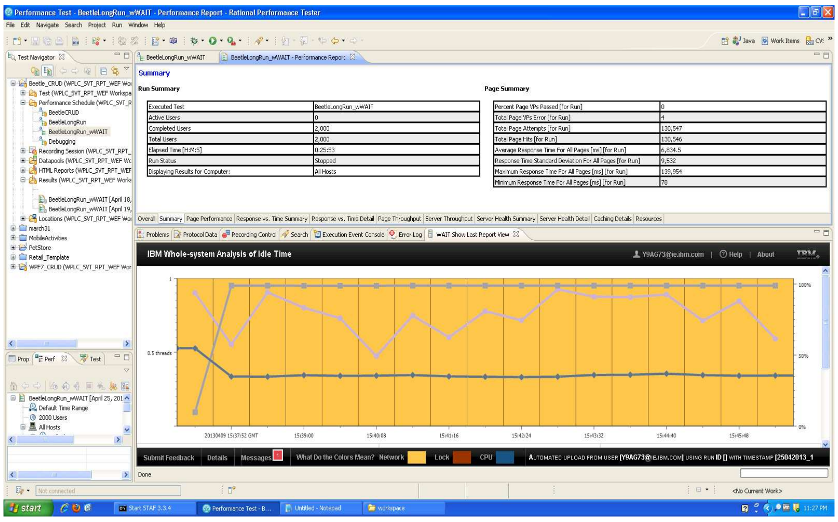
Fig. 4. WAIT monitoring in RPT

6 Experimental Evaluation and Results

Two experiments were performed. The first one pursed to answer the research question Q2 (Can the overhead be kept low during a performance test execution to avoid compromising the results?), while the second one pursued to answer the research question Q3 (What benefits in productivity can a tester achieve if the previous two questions are answered?). Finally, the combined outcome of the experiments pursued to answer the research question Q1 (How can the usage of WAIT be automated to minimize the effort required to use it in sync with a performance test?).

Moreover two environments were used: One was composed of one RPT node, one application node and one WAIT Server node; the other was composed of

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



Summary

Run Summary

Executed Test	BeetleLongRun_wWAIT
Active Users	0
Completed Users	2,000
Total Users	2,000
Elapsed Time [H:M:S]	0:25:53
Run Status	Stopped
Displaying Results for Computer:	All Hosts

Page Summary

Percent Page VPs Passed [for Run]	0
Total Page VPs Error [for Run]	4
Total Page Attempts [for Run]	130,547
Total Page Hits [for Run]	130,546
Average Response Time For All Pages [ms] [for Run]	6,834.5
Response Time Standard Deviation For All Pages [for Run]	9,532
Maximum Response Time For All Pages [ms] [for Run]	139,954
Minimum Response Time For All Pages [ms] [for Run]	78

Performance Test - BeetleLongRun_wWAIT - Performance Report - Rational Performance Tester
File Edit Navigate Search Project Run Window Help
BeetleLongRun_wWAIT
BeetleLongRun_wWAIT - Performance Report
Overall Summary Page Performance Response vs. Time Summary Response vs. Time Detail Page Throughput Server Throughput Server Health Summary Server Health Detail Caching Details Resources
Problems Protocol Data Recording Control Search Execution Event Console Error Log WAIT Show Last Report View
IBM Whole-system Analysis of Idle Time
Help About
100%
50%
0%
0.5 threads
Submit Feedback Details Messages
What Do the Colors Mean? Network Lock CPU

Fig. 5. WAIT Report accessible in RPT

one RPT node, one load balance node, two application nodes and one WAIT Server node. All nodes connected by a 10-GBit LAN. These set-ups are shown in Fig. 6. The RPT node was a machine running Windows XP with an Intel

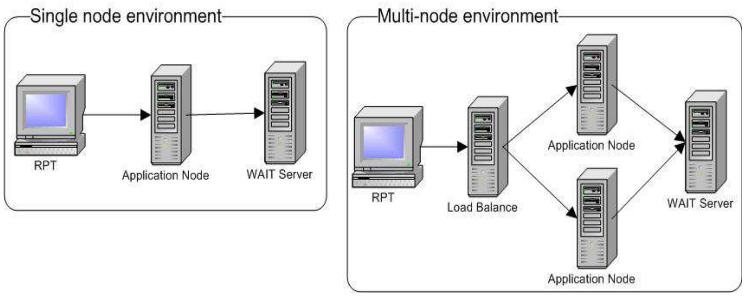


Fig. 6. Environment Set-ups used in the Experiments

Xeon processor at 2.67 Ghz and 3GB of RAM using RPT 8.2.1.3. The WAIT Server was a machine running Red Hat Enterprise Linux Server 5.9, with an Intel Xeon processor 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 (4 cores) at 2.26 GHz and 4GB of RAM running Java 1.6.0 IBM J9 VM (build 2.6). Finally, the load balance node had the same characteristics of the WAIT Server node and used mod_jk and a round robin strategy for load balance.

6.1 Experiment #1

Its objective was to validate that the proposed approach had a low overhead. It involved the assessment of four metrics: Throughput and response time (classic metrics in performance testing) as well as CPU and memory utilization (to assess how much additional resources the automation required). All metrics were collected through RPT. Also two real-world applications were used in this case study: One was iBatis JPetStore 4.0 [8] which is an upgraded version of Sun's original J2EE Pet Store, an e-commerce shopping cart application used to teach J2EE. Unlike Suns, JPetStore is a reimplementation with a more efficient design [9]. This application was run over an Apache Tomcat 6.0.35 Application Server. The other application was IBM WebSphere Portal Server 8.0.1 [10], a leader solution in the enterprise portal market [23]. This application was run over an IBM WebSphere Application Server 8.0.0.5.

The experiment involved two tests. First, the behaviour of the approach was tested in a single-node environment to assess how much overhead it brought under those conditions. For each application, 3 combinations of WAIT were run: First the application was executed without WAIT to get a baseline; then a manual WAIT execution (data collection without upload) was introduced to measure how much overhead WAIT added. Finally, the automated approach was introduced to measure how much overhead it added. For each combination in which WAIT was present, two *Sampling Interval* were tested: One value (480 seconds), suggested by our industrial partner and commonly used in the industry. The other selected value (30 seconds) is the minimum recommended *Sampling Interval* for WAIT. The remaining parameters involved in the performance test were suggested by our industrial partner: 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, for each of the 5 combinations per application, 3 runs were performed.

For JPetStore, each test run produced around 500,000 transactions. The results showed that using WAIT with a *Sample Interval* of 480 seconds had practically no impact in terms of response time and throughput. Furthermore the difference in resource consumption between the manual and automated approaches was around 1%. As the uploaded data was very small in this case (around 200KB every 10 minutes), the difference in resource consumption was related to the presence of the WebAgent. When WAIT used a *Sample Interval* of 30 seconds, the impacts in response time and throughput appeared. As the impact in throughput was similar between approaches, this should had been caused by the *Javacore* generation (step shared between approaches). In average, the generation of a *Javacore* took around 1 second. Even though this cost was neglectable in the higher *Sample 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

[8] http://sourceforge.net/projects/ibatisjpetstore/
[9] www.clintonbegin.com/downloads/JPetStore-1-2-0.pdf
[10] http://www-03.ibm.com/software/products/us/en/portalserver

data), as the cost of the WebAgent had been previously measured. In terms of resource consumption, the differences between approaches were similar to the *Sample Interval* of 480 seconds. These results are shown in the Table 1 where the first row shows the baseline and the rest the percentage differences of each tested combination.

Table 1. PetStore - 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 showed similar trends in terms of achieving lower overheads using the *Sampling Interval* of 480 seconds, a few key differences were identified: First, the impact in response time and throughput was 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 difference in functionality between the tested applications. This was confirmed after analyzing the *Javacores* generated by the Portal, which allowed to quantify the difference in behavior of Portal: The average size of a *Javacore* was 5.5MB (450% bigger than JPetStore's), its average generation time was 2s (100% bigger than JPetStore's), with a maximum generation time of 3s (also 100% bigger than JPetStore's). The results are shown in the Table 2.

Table 2. Portal - 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

[11], using a significant level of p¡0.1, was done to evaluate if the differences in response time and throughput were statistically significant. For JPetStore, the results showed that the differences were only significant in the average response time using the *Sample Interval* of 30 seconds, and in the average throughput using the *Sample Interval* if 30 seconds with the automated approach. Moreover the results of Portal were similar. This analysis reinforced the conclusion that the overhead was low as well as the observation that the usage of the *Sample Interval* of 480 seconds was preferable in terms of overhead. These results are shown in the Table 3.

Table 3. Paired t-Test Results

Application	**WAIT Modality**	**Avg Response Time (ms)**	**Max Response Time (ms)**	**Avg Throughput (hps)**
PetStore	Manual, 480s	0.470	0.143	0.206
PetStore	Automated, 480s	0.342	0.297	0.472
PetStore	Manual, 30s	0.089	0.241	0.154
PetStore	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 and a longer test run. JPetStore was used for this test. First it was executed alone to have a baseline; then the automated WAIT approach was introduced using a *Sampling Interval* of 480 seconds. The rest of the set-up was identical to the previous tests with two exemptions: The workload was doubled to compensate the additional application node and the test duration 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% 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 (and 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, 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 this when using WAIT. In our case, the *Sampling Interval* of 480 seconds proved efficient in terms of overhead for the two tested applications.

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

6.2 Experiment #2

Here the objective was to assess the benefits the approach brings to a performance tester. For this experiment the automated approach was used to monitor a modified version of the iBatis PetStore 4.0 application. The source code was modified to inject some performance issues to assess how WAIT was able to identify them and estimate the corresponding time savings in performance analysis. Specifically, 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 [12]. All other set-up parameters were identical to the multi-node test previously performed with exception of the duration which was 1-hour. To save space, only the most relevant sections of the GUI are presented.

Surprisingly the first ranked issue was none of the injected bugs but a method named "McastServiceImpl.receive". Even though further analysis determined this method call was benigne (is used by the clustering functionality of Tomcat), its presence in the results was appropriate. As it appeared in practically all the samples, it was worth an investigation. The secondly ranked issue was the lock contention. A relevant point to highlight is that these two issues were detected since the early versions of the report. Based on their high frequency (above 96% of the samples), this information could have lead a tester to raise bugs and 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 ranks. These additional information could also be shared with the development team. Fig. 7.a shows the results of the early report, while 7.b shows the results of the final report.

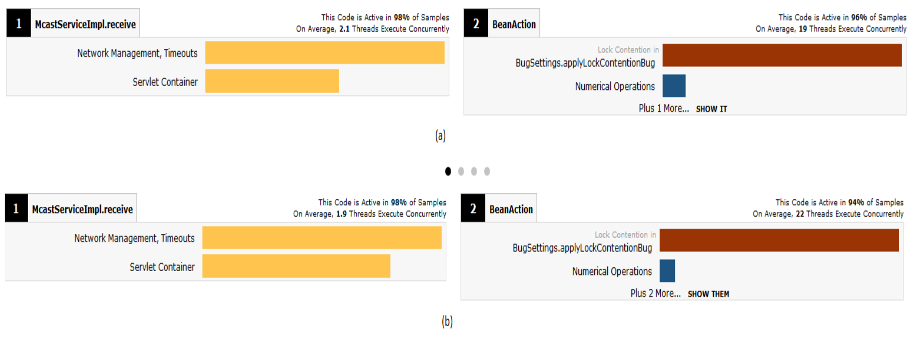
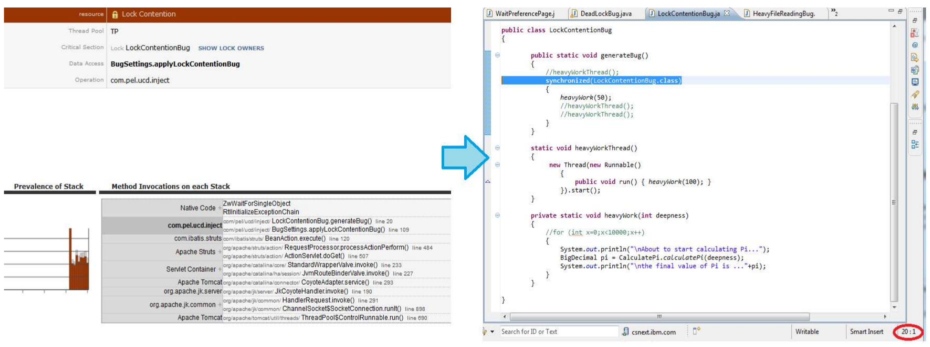

1 McastServiceImpl.receive
This Code is Active in 98% of Samples
On Average, 2.1 Threads Execute Concurrently
Network Management, Timeouts
Servlet Container
2 BeanAction
This Code is Active in 96% of Samples
On Average, 19 Threads Execute Concurrently
Lock Contention in
BugSettings.applyLockContentionBug
Numerical Operations
Plus 1 More... SHOW IT
(a)
1 McastServiceImpl.receive
This Code is Active in 98% of Samples
On Average, 1.9 Threads Execute Concurrently
Network Management, Timeouts
Servlet Container
2 BeanAction
This Code is Active in 94% of Samples
On Average, 22 Threads Execute Concurrently
Lock Contention in
BugSettings.applyLockContentionBug
Numerical Operations
Plus 2 More... SHOW THEM
(b)

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

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,

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

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). The report showed



resource	Lock Contention
Thread Pool	TP
Critical Section	Lock LockContentionBug SHOW LOCK OWNERS
Data Access	BugSettings.applyLockContentionBug
Operation	com.pel.ucd.inject

Method Invocations on each Stack

Native Code +	ZwWaitForSingleObject RtlInitializeExceptionChain
com.pel.ucd.inject	com/pel/ucd/inject/ LockContentionBug.generateBug() line 20 com/pel/ucd/inject/ BugSettings.applyLockContentionBug() line 109
com.ibatis.struts	com/ibatis/struts/ BeanAction.execute() line 120
Apache Struts +	org/apache/struts/action/ RequestProcessor.processActionPerform() line 484 org/apache/struts/action/ ActionServlet.doGet() line 507
Servlet Container +	org/apache/catalina/core/ StandardWrapperValve.invoke() line 233 org/apache/catalina/ha/session/ JvmRouteBinderValve.invoke() line 227
Apache Tomcat	org/apache/catalina/connector/ CoyoteAdapter.service() line 293
org.apache.jk.server	org/apache/jk/server/ JkCoyoteHandler.invoke() line 190
org.apache.jk.common +	org/apache/jk/common/ HandlerRequest.invoke() line 291 org/apache/jk/common/ ChannelSocket$SocketConnection.runIt() line 898
Apache Tomcat	org/apache/tomcat/util/threads/ ThreadPool$ControlRunnable.run() line 690

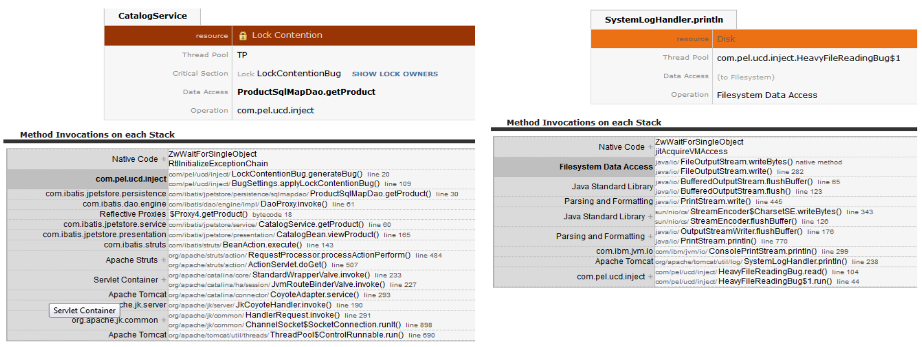
WaitPreferencePage.j | DeadLockBug.java | LockContentionBug.ja | HeavyFileReadingBug.

```java
public class LockContentionBug
{

        public static void generateBug()
        {
            //heavyWorkThread();
            synchronized(LockContentionBug.class)
            {
                heavyWork(50);
                //heavyWorkThread();
                //heavyWorkThread();
            }
        }

        static void heavyWorkThread()
        {
             new Thread(new Runnable()
                {
                    public void run() { heavyWork(100); }
                }).start();
        }

        private static void heavyWork(int deepness)
        {
            //for (int x=0;x<10000;x++)
            {
                System.out.println("\nAbout to start calculating Pi...");
                BigDecimal pi = CalculatePi.calculatePi(deepness);
                System.out.println("\nthe final value of Pi is ..."+pi);
            }
        }

}
```

Search for ID or Text | csnext.ibm.com | Writable | Smart Insert | 20 : 1

Fig. 8. Comparison of lock contention issue in the WAIT report against the actual source code

in 3rd place a symptom of the lock contention issue (it was possible to relate the issues by comparing their detail information, where both issues pinpointed to the same class/method), suggesting this was a major issue. Moreover the I/O latency bug was identified in 4th place. Fig. 9 shows the details of these two issues. The deadlock issue did not appear in the run, somehow prevented by



CatalogService

resource	Lock Contention
Thread Pool	TP
Critical Section	Lock LockContentionBug SHOW LOCK OWNERS
Data Access	ProductSqlMapDao.getProduct
Operation	com.pel.ucd.inject

Method Invocations on each Stack

Native Code +	ZwWaitForSingleObject RtlInitializeExceptionChain
com.pel.ucd.inject	com/pel/ucd/inject/ LockContentionBug.generateBug() line 20 com/pel/ucd/inject/ BugSettings.applyLockContentionBug() line 109
com.ibatis.jpetstore.persistence	com/ibatis/jpetstore/persistence/sqlmapdao/ ProductSqlMapDao.getProduct() line 30
com.ibatis.dao.engine	com/ibatis/dao/engine/impl/ DaoProxy.invoke() line 61
Reflective Proxies	$Proxy4.getProduct() bytecode 18
com.ibatis.jpetstore.service	com/ibatis/jpetstore/service/ CatalogService.getProduct() line 60
com.ibatis.jpetstore.presentation	com/ibatis/jpetstore/presentation/ CatalogBean.viewProduct() line 165
com.ibatis.struts	com/ibatis/struts/ BeanAction.execute() line 143
Apache Struts +	org/apache/struts/action/ RequestProcessor.processActionPerform() line 484 org/apache/struts/action/ ActionServlet.doGet() line 507
Servlet Container +	org/apache/catalina/core/ StandardWrapperValve.invoke() line 233 org/apache/catalina/ha/session/ JvmRouteBinderValve.invoke() line 227
Apache Tomcat	org/apache/catalina/connector/ CoyoteAdapter.service() line 293
che.jk.server	org/apache/jk/server/ JkCoyoteHandler.invoke() line 190
org.apache.jk.common +	org/apache/jk/common/ HandlerRequest.invoke() line 291 org/apache/jk/common/ ChannelSocket$SocketConnection.runIt() line 898
Apache Tomcat	org/apache/tomcat/util/threads/ ThreadPool$ControlRunnable.run() line 690

Servlet Container

SystemLogHandler.println

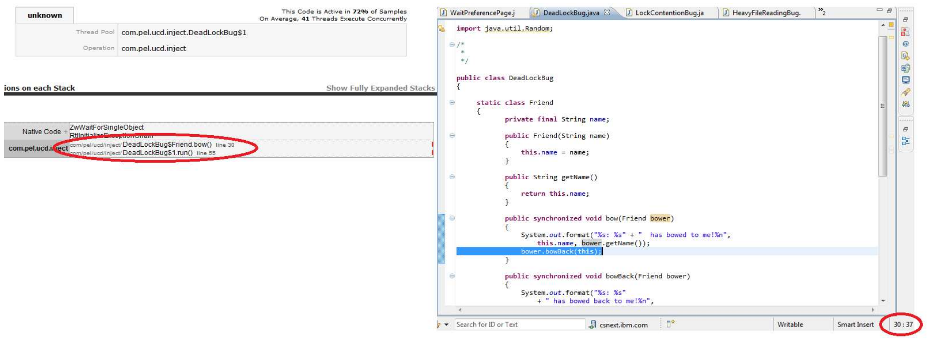
resource	Disk
Thread Pool	com.pel.ucd.inject.HeavyFileReadingBug$1
Data Access	(to Filesystem)
Operation	Filesystem Data Access

Method Invocations on each Stack

Native Code +	ZwWaitForSingleObject jitAcquireVMAccess
Filesystem Data Access	java/io/ FileOutputStream.writeBytes() native method java/io/ FileOutputStream.write() line 282
Java Standard Library	java/io/ BufferedOutputStream.flushBuffer() line 65 java/io/ BufferedOutputStream.flush() line 123
Parsing and Formatting	java/io/ PrintStream.write() line 445
Java Standard Library +	sun/nio/cs/ StreamEncoder$CharsetSE.writeBytes() line 343 sun/nio/cs/ StreamEncoder.flushBuffer() line 126
Parsing and Formatting +	java/io/ OutputStreamWriter.flushBuffer() line 176 java/io/ PrintStream.println() line 770
com.ibm.jvm.io	com/ibm/jvm/io/ ConsolePrintStream.println() line 299
Apache Tomcat	org/apache/tomcat/util/log/ SystemLogHandler.println() line 238
com.pel.ucd.inject +	com/pel/ucd/inject/ HeavyFileReadingBug.read() line 104 com/pel/ucd/inject/ HeavyFileReadingBug$1.run() line 44

Fig. 9. Details of issues ranked 3 and 4 in the first test run

the lock contention bug which had a major impact that planned. As per the results of the run, the two identified bugs were "fixed" from the code. As in any common test phase, an additional 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


unknown

This Code is Active in **72%** of Samples
On Average, **41** Threads Execute Concurrently

Thread Pool	com.pel.ucd.inject.DeadLockBug$1
Operation	com.pel.ucd.inject

ions on each Stack — Show Fully Expanded Stacks

```
Native Code        ZwWaitForSingleObject
                   RtlInitializeExceptionChain
com.pel.ucd.inject com/pel/ucd/inject/ DeadLockBug$Friend.bow()  line 30
                   com/pel/ucd/inject/ DeadLockBug$1.run()  line 55
```

```java
import java.util.Random;

/*
 *
 */

public class DeadLockBug
{

    static class Friend
    {
        private final String name;

        public Friend(String name)
        {
            this.name = name;
        }

        public String getName()
        {
            return this.name;
        }

        public synchronized void bow(Friend bower)
        {
            System.out.format("%s: %s" + "  has bowed to me!%n",
                this.name, bower.getName());
            bower.bowBack(this);
        }

        public synchronized void bowBack(Friend bower)
        {
            System.out.format("%s: %s"
                + " has bowed back to me!%n",
```

Fig. 10. Comparison of deadlock issue in the WAIT report against the actual source code

(as the deadlock occurs when the friends try to bow back to each other). As
the measurement of success for this experiment was to the ability of WAIT to identify performance problems, 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 identified. First, the automated approach practically reduced the overhead of collecting data for WAIT to zero. In parallel to the execution of both experiments (around 40 performance tests), the time spent in the automated approach was tracked. The initial installation took no more than 15 minutes; then its usage require no additional time over the regular usage of the RPT (barely a few seconds to change its settings whenever needed -i.e. to modify the *Sampling Interval*-).

The second time saving was in the analysis of the WAIT reports. Previously, a tester would have ended with multiple ones. Assuming an environment composed of X nodes, with a desired time window of 1-hour and a test duration of Y hours, a tester would have ended with X*Y different WAIT reports, having to review X reports per hour. With the automated approach, a tester only needs to review a single report which gets refreshed every hour.

Furthermore, overcoming the usability constraints of WAIT allow a tester to enjoy the benefits of WAIT and exploit its expert knowledge capabilities to identify performance issues. Even though it might be hard to define an average time spent identifying performance issues, a conservative estimate (i.e. 2 hours) could quantify WAIT's savings. In our study case, instead of spending 6 hours analyzing the issues, it was possible to identify them and their root causes with the information provided by the WAIT report. Moreover a tester does not need to wait until the end of the test to raise bugs if required. As seen in the experiment, additional time can be saved by reporting relevant issues in parallel to the execution of the test. This will be especially valuable in long-term runs, very common in performance testing, and usually lasting several days.

In conclusion, the results of this second experiment answered our research question Q3 (What benefits in productivity can a tester achieve if the previous two questions are answered?), showing the productivity gains that a tester achieves by using the automated approach of 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). Besides 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 practically zero. Previously the number of WAIT reports a tester needed to review was equal to the number of nodes multiplied by the times the collected data was uploaded; now, a tester only needs to monitor a single report, which offers an incremental view of the results. A direct benefit of these savings is the reduction of expert knowledge required by tester to do performance analysis and detect performance issues along with their root causes. Ultimately, this improves a tester productivity by reducing the effort required to locate defects.

6.3 Threads 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. Also to determine if the differences in the results were statistically significant (with 90% of certainty), a Paired t-Test was applied.

Another thread was the selection of the test parameters (i.e. workload and duration). This was addressed with the expert judgement of the SVT, which guided on their selection. Finally, the validity of the results are threatened by 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 effective used to reduce the expertise 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 light-weight, generating low overhead (in the range of 0% to 3% using a *Sample 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 SVT IBM, 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. Capers Jones: Applied Software Measurement: Global Analysis of Productivity and Quality. McGraw-Hill (2008)
2. Compuware: Applied Performance Management Survey. (2007)
3. Woodside, M., Franks, G., Petriu, D.C.: The Future of Software Performance Engineering. Future of Software Engineering (FOSE '07) (May 2007) 171–187
4. Parsons, T., Murphy, J.: Detecting Performance Antipatterns in Component Based Enterprise Systems. In: 2nd International Middleware Doctoral Symposium. Volume 7. (2008) 55–90
5. 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
6. Shahamiri, S.R., Kadir, W.M.N.W., M.: A Comparative Study on Automated Software Test Oracle Methods. ICSEA (2009)
7. Altman, E., Arnold, M., Fink, S., Mitchell, N.: Performance analysis of idle programs. ACM SIGPLAN Notices **45**(10) (October 2010) 739
8. Wu, Haishan, Asser N. Tantawi, T.: A Self-Optimizing Workload Management Solution for Cloud Applications. (2012)
9. Chen, S., Moreland, D., N.Z.: Yet Another Performance Testing Framework. Australian Conference on Software Engineering (ASWEC) (2008)
10. Albert, Elvira, Miguel Gmez-Zamalloa, J.: Resource-Driven CLP-Based test case generation. Logic-Based Program Synthesis and Transformation (2012)
11. J. Zhang, S.: Automated test case generation for the stress testing of multimedia systems. Softw. Pract. Exper. (2002)
12. 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)
13. M. S. Bayan, J.: Automatic stress and load testing for embedded systems. 30th Annual International Computer Software and Applications Conference (2006)
14. A. Avritzer, E.: Generating test suites for software load testing. ACM SIGSOFT international symposium on Software testing and analysis (1994)

15. A. Avritzer, E.: The automatic generation of load test suites and the assessment of the resulting software. IEEE Trans. Softw. Eng. (1995)
16. V. Garousi, L. C. Briand, Y.: Traffic-aware stress testing of distributed systems based on uml models. International conference on Software engineering (2006)
17. J. Yang, D. Evans, D.T.M.: Perracotta: mining temporal api rules from imperfect traces. International conference on Software engineering (2008)
18. S. Hangal, M.: Tracking down software bugs using automatic anomaly detection. International Conference on Software Engineering (2002)
19. C. Csallner, Y.: Dsd-crasher: a hybrid analysis tool for bug finding. International symposium on Software testing and analysis (2006)
20. M. Y. Chen, A. Accardi, E.J.A.E.: Path-based failure and evolution management. Symposium on Networked Systems Design and Implementation (2004)
21. 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
22. Haroon Malik, Bram Adams, A.: Pinpointing the subsys responsible for the performance deviations in a load test. Software Reliability Engineering (ISSRE) (2010)
23. Gootzit, David, Gene Phifer, R.: Magic Quadrant for Horizontal Portal Products. Technical report, Gartner Inc. (2008)