Towards an Automated Approach to Use Expert Systems in the Performance Testing of Distributed Systems

A. Omar Portillo-Dominguez, Miao Wang, John Murphy, Damien Magoni, Nick Mitchell, Peter Sweeney, Erik Altman

Towards an Automated Approach to Use Expert Systems in Performance Testing

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

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

Nick Mitchell,Peter F. Sweeney,Erik Altman
IBM T.J. Watson Research Center, New York, USA
{nickm,pfs,ealtman}@ucd.ie

ABSTRACT

Performance testing in highly distributed environments is very challenging. Specifically, the identification of performance issues and the diagnosis of their root causes are time-consuming and complex tasks which usually require multiple tools and heavily rely on expertise. To simplify these tasks, hence increasing the productivity and reducing the dependency on human experts, many researchers have been developing tools with built-in expertise for non-expert users. However various limitations exist in these tools, such as managing huge volumes of generated data, that prevent their efficient usage in the performance testing of highly distributed environments. To address these limitations, this paper presents an adaptive framework to automate the usage of expert tools in performance testing. In this paper, we use a tool named Whole-system Analysis of Idle Time to demonstrate how our research work solves this problem. The validation involved two experiments which assessed the accuracy of the proposed adaptive framework and the time savings that it can bring to the analysis of performance issues. The results proved the benefits of the framework by achieving a significant decrease in the time invested in performance analysis.

Categories and Subject Descriptors

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

General Terms

Algorithms, Measurement, Performance

Keywords

Performance Testing, Automation, Performance Analysis, Expert Tools, Distributed Systems

1. INTRODUCTION

It is an accepted fact in the industry that performance is a critical dimension of quality and should be a major concern of any software project. This is especially true at enterprise-level, where the system performance plays a central role in using the software to achieve business goals. However it is not uncommon that performance issues occur and materialize into serious problems in a significant percentage of applications (i.e. outages on production environments or even cancellation of software projects). For example, a 2007 survey applied to information technology executives [10] reported that 50% of them had faced performance problems in at least 20% of their deployed applications. Similarly, many research studies have documented the magnitude of this problem. For example, in [12] authors found 332 previously unknown performance problems in the latest versions of five mature open-source software suites.

This situation is partially explained by the pervasive nature of performance, which makes it hard to assess because performance is practically influenced by every aspect of the design, code, and execution environment of an application. The latest trends in information technology (such as Service Oriented Architecture[1] and Cloud Computing[2]) have also augmented the complexity of applications further complicating all 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 [30, 23, 4], is that current performance tools heavily rely on human experts to be configured properly and to interpret their outputs. Also multiple sources are commonly required to diagnose performance problems, especially in highly distributed environments. For instance in Java: thread dumps, garbage collection logs, heap dumps, CPU utilization and memory usage, are a few examples of the information that a tester could need to understand the performance of an application. This problem increases the expertise required to do performance analysis, which is usually held by only a small number of experts inside an organization[27]. Therefore this issue could potentially lead to bottlenecks where certain activities can

[1]http://msdn.microsoft.com/en-us/library/aa480021.aspx
[2]http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf

only be done by these experts, impacting the productivity of the testing teams[4].

To simplify the performance analysis and diagnosis, hence increasing the productivity and reducing the dependency on human experts, many researchers have been developing tools with built-in expertise [2, 3, 4]. However, various limitations exist in these tools that prevent their efficient usage in the performance testing of highly distributed environments. Firstly, these tools still need to be manually configured, situation which might impact their accuracy, as their outputs are commonly sensitive to the used configuration. Therefore, if an inappropriate configuration is used, the tools might fail to obtain the desired outputs. Sequentially, the data collection usually needs to be controlled manually which, in an environment composed of multiple nodes to monitor and coordinate simultaneously, is very time-consuming and error-prone due to the vast amount of data to collect and consolidate. This challenge is more complex if the data needs to be processed periodically during a test execution to get incremental results. A similar problem occurs with the outputs, where a tester commonly gets multiple reports, one for each monitored node per data processing cycle.

Even though these limitations might be manageable in small testing environments, they prevent the efficient usage of these tools in bigger environments. To exemplify this problem, let's use the Eclipse Memory Analyzer Tool[3] (MAT), which is a popular open source tool to identify memory consumption issues in Java. If a tester wants to use MAT to monitor an environment composed of 100 nodes during a 24-hour test run and get incremental results every hour, she would need to manually coordinate the data gathering of memory snapshots and the generation of the tool's reports. These steps conducted periodically for the 100 nodes every hour, which yields a total of 2400 iterations. Moreover the tester would have to review the multiple reports she would get per hour to evaluate if any memory issues exist. As an alternative, she may concentrate the analysis on a single node, assuming it is representative of the whole system. However it generates the risk of potentially overlooking issues in the tested application. An additional assumption in the above scenario is that an appropriate configuration was used. Otherwise, the tester would have also experienced the risk of potentially overlooking issues.

In addition to these challenges, the overhead generated by any technique should be low to minimize the impact it has in the tested environment (i.e. inaccurate results or abnormal behavior). Otherwise the technique would not be suitable for performance testing. For example, instrumentation[4] is currently a common approach used in performance analysis to gather input data [34, 25, 7, 8]. However, it has the downside of obscuring the performance of the instrumented applications, hence compromising the results of performance testing. Similarly, if a tool requires heavy human effort, this might limit its applicability. On the contrary, automation could encourage the adoption of a technique. As documented by the authors in [26, 11], this strategy has proven successful in performance testing.

[3]http://www.eclipse.org/mat/

[4]http://msdn.microsoft.com/en-us/library/aa983649(VS.71).aspx

Finally, to ensure that our research is helpful to solve real-life problems in the software industry, we have been working with our industrial partner, IBM System Verification Test team (SVT), to understand the challenges in their testing activities. Their feedback confirms that there is a real need to automate the usage of expert tools so that testers can do analysis tasks in less time and get more meaningful results that can be conveyed with the development teams.

This paper proposes an adaptive automation framework that addresses the common usage limitations of an expert system in performance testing. During our research development work we have successfully applied our approach to the IBM Whole-system Analysis of Idle Time tool (WAIT)[5]. This publicly available tool is a lightweight expert system that helps to identify the main performance inhibitors in Java systems. Our work was validated through two experiments. The first experiment evaluated the accuracy of the proposed adaptive framework. The results demonstrated that the adaptive framework is able to configure WAIT, based on a desired usage scenario, without the need of manually tuning the tool. The second experiment assessed the productivity gains that our framework brought to the performance testing process. The results provided evidence about the benefits of the framework, as the effort required by a tester to use and analyze the outputs of the selected expert tool (WAIT) was reduced in 68%. This usage simplification translated into a quicker identification of performance issues; which leads to a reduction in the duration of the performance testing activities of 27%.

The main contributions of this paper are:

1. A novel adaptive framework to automate the usage of expert systems in performance testing.
2. Two novel adaptive policies to self-configure an expert system in performance testing.
3. A practical validation of the approach consisting of the implementation around the WAIT tool and two experiments. The first experiment demonstrates that the accuracy of our framework, and the second experiment demonstrates its productivity benefits.

The rest of this paper is structured as follows: Section 2 discusses the background and related work. Section 3 explains the proposed framework, and the prototype. Section 4 describes the assessment performed to identify suitable adaptive policies; while Section 5 presents the proposed adaptive policies. Section 6 describes the performed experiments. Finally, Section 7 shows the conclusions and future work.

2. BACKGROUND AND RELATED WORK

Idle-time analysis is a methodology to identify the root causes of under-utilized resources. This approach, proposed in [2], is based on the observed behavior that performance problems in multi-tier applications usually manifest as idle time of waiting threads. WAIT is an expert system that implements the idle-time analysis and identifies the main performance inhibitors that exist on a system. Moreover it has proven successful in simplifying the detection of performance issues and their root causes in Java systems [2, 31].

WAIT is based on non-intrusive sampling mechanisms available at Operating System level (i.e. "ps" command in a

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

Unix environment) and the Java Virtual Machine (JVM), in the form of *Javacores* [6] (diagnostic feature to get a snapshot of the JVM state, offering information such as threads, locks and memory). The fact that WAIT uses standard data sources makes it non-disruptive, as no special flags, restart or instrumentation are required to use it. It also requires infrequent samples to perform its diagnosis, so it has low overhead. From an end-user perspective, WAIT is simple: A user only needs to collect as much data as desired, upload it to a public web page and get a report with the findings. This process can be repeated multiple times to monitor a system through time. Internally, WAIT uses an engine built on top of a set of expert rules to perform the analysis.

Given its strengths, WAIT is a promising candidate to reduce the dependence on a human expert and reduce the time required for performance analysis. However, as with many expert systems that could be used for testing distributed software systems, the volume of data generated can be difficult to manage and efficiently process this data can be an impediment to their adoption. Similarly, the accuracy of WAIT depends on its configuration and the preferable configuration might vary depending on the application and usage scenario. These characteristics make WAIT a good candidate to apply our proposed framework.

Automation in Testing. 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[1, 20, 16, 18, 9, 28, 33]. For example [20] proposes an approach to automate the generation of test cases based on specified levels of load and combinations of resources. Similarly, [9] presents an automation framework that separates the application logic from the performance testing scripts to increase the re-usability of the test scripts. Meanwhile [33] presents a framework designed to automate the performance testing of web applications and which internally utilizes two usage models to simulate the users' behaviors more realistically.

Other research efforts have concentrated on automating specific analysis techniques. For example, the authors of [35] present a combination of coverage analysis and debugging techniques to automatically isolate failure-inducing changes. Similarly, the authors of [19] developed a technique to reduce the number of false memory leak warnings generated by static analysis techniques by automatically validating and categorizing those warnings.

Finally, other researchers have proposed frameworks to support different software engineering processes. For example, the authors of [14, 5] present frameworks to monitor software services. Both frameworks monitor the resource utilisation and the component interactions within a system, but target different technologies ([14] focuses on Java and [5] on Microsoft technologies). Unlike these works, which have been designed to assist on operational support activities, our proposed framework has been designed to address the specific needs of a tester in the performance testing, isolating her from the complexities of an expert system.

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

Performance Analysis. A major research trend has focused on identifying performance bugs and their root causes. For example, the work on [32] proposes an approach to predict the workload-dependent performance bottlenecks (WDPBs) through complexity models. The work on [36] presents a technique to detect processes accessing a shared resource without proper synchronization; while the authors of [6] analysed the memory heaps of real-world object-oriented programs to provide insights to improve memory allocation and program analysis techniques.

A high percentage of the proposed performance analysis techniques require some type of instrumentation. For example, the authors in [34] instrument the source code of the monitored applications to mine the sequences of call graphs under normal operation, information which is later used to infer any relevant error patterns. A similar case occurs with the works presented in [25, 7] which rely on instrumentation to dynamically infer invariants within the applications and detect programming errors; or the approach proposed by [8] which uses instrumentation to capture execution paths to determine the distributions of normal paths and look for any significant deviations in order to detect errors. In all these cases, the instrumentation would obscure the performance of an application during performance testing hence discouraging their usage. On the contrary, our proposed framework does not require to instrument the tested applications.

Furthermore, the authors of [17] present a non-intrusive approach which automatically analyzes the execution logs of a load test to identify performance problems. As this approach only relies on load testing results, it can not determine root causes. A similar approach is presented in [13] which aims to offer information about the causes behind the issues. However it can only provide the subsystem responsible of the performance deviation. On the contrary, our approach allows the applicability of the idle-time analysis in the performance testing domain through automation, which allows the identification of the classes and methods responsible for the performance issues.

3. ADAPTIVE AUTOMATED FRAMEWORK

The objective of this work was to automate the manual processes involved in the usage of an expert system to improve a tester productivity by decreasing the effort and expertise needed to use an expert system.

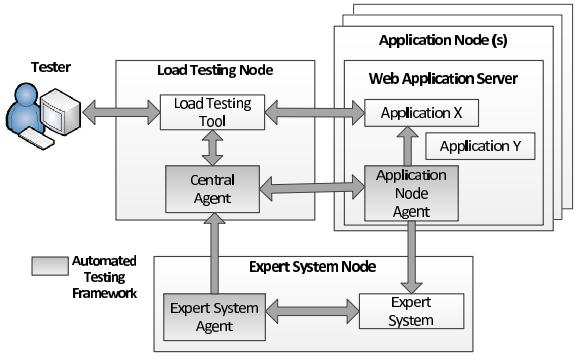

Figure 1: Proposed framework - Contextual view

This scenario is depicted in Figure 1, where our proposed framework executes concurrently with a performance test, shields the tester from the complexities of properly configuring and using the expert system, so that she only needs to interact with the load testing tool.

The following sections describe our proposed framework, its supporting architecture, and our implemented prototype.

Adaptive Framework. Our proposed adaptive framework is depicted in Figure 2. As a self-adaptive system is normally composed of a managed system and an autonomic manager [21], our framework plays the role of the autonomic manager. Therefore, it controls the feedback loops which adapts the managed system according to a set of goals. Meanwhile, the expert system and the application nodes under test play the role of the managed systems.

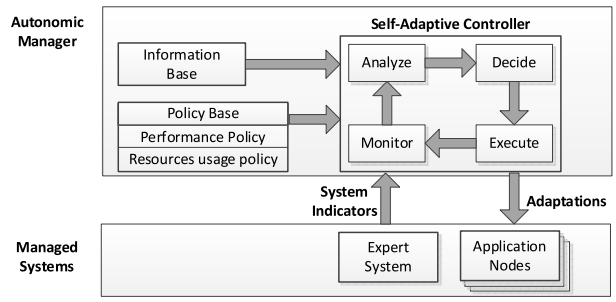

Figure 2: Adaptive automation framework

As defined by multiple authors [22, 29], self-adaptation endows a system with the capability to adapt itself autonomously to internal changes and dynamics in the environment to achieve particular quality goals in the face of uncertainty. In order to achieve that goal, our framework follows the well-known MAPE-K adaptive model [15]. This model is composed of 5 components: A *Monitoring* component to obtain information from the managed systems; an *Analysis* component to evaluate if any adaptation is required; then a component to *Plan* the adaptation, and a component to *Execute* it. Finally a *Knowledge* component supports the others in their respective tasks.

The key element of our proposed framework is its policy base, which fulfills the role of the *Knowledge* component and defines the pool of available adaptive policies. Each expert system requires to have at least two policies: A data gathering policy (to control the collection of samples), and an upload policy (to control when the samples are processed by the expert system). Additionally, an expert system might have other policies available (i.e. to back up the obtained samples, or correlate the expert system outputs).

From a configuration perspective, the tester needs to provide the *Information Base*, which is composed of all the input parameters required by the chosen policies. For example, an upload policy might require a *Time Threshold* to process the samples periodically, or a data gathering policy could use a *Sampling Interval* to coordinate the collection of samples.

From a process perspective, the autonomic manager has a

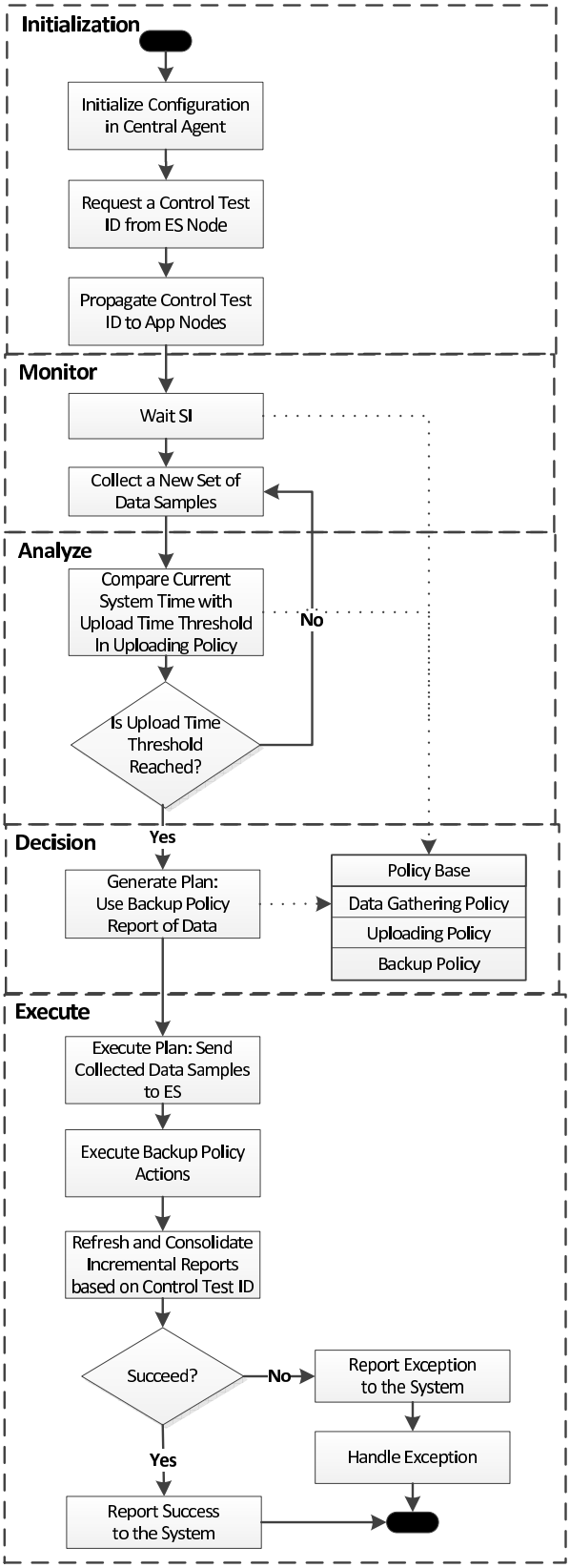

Figure 3: Automation framework - Core Process

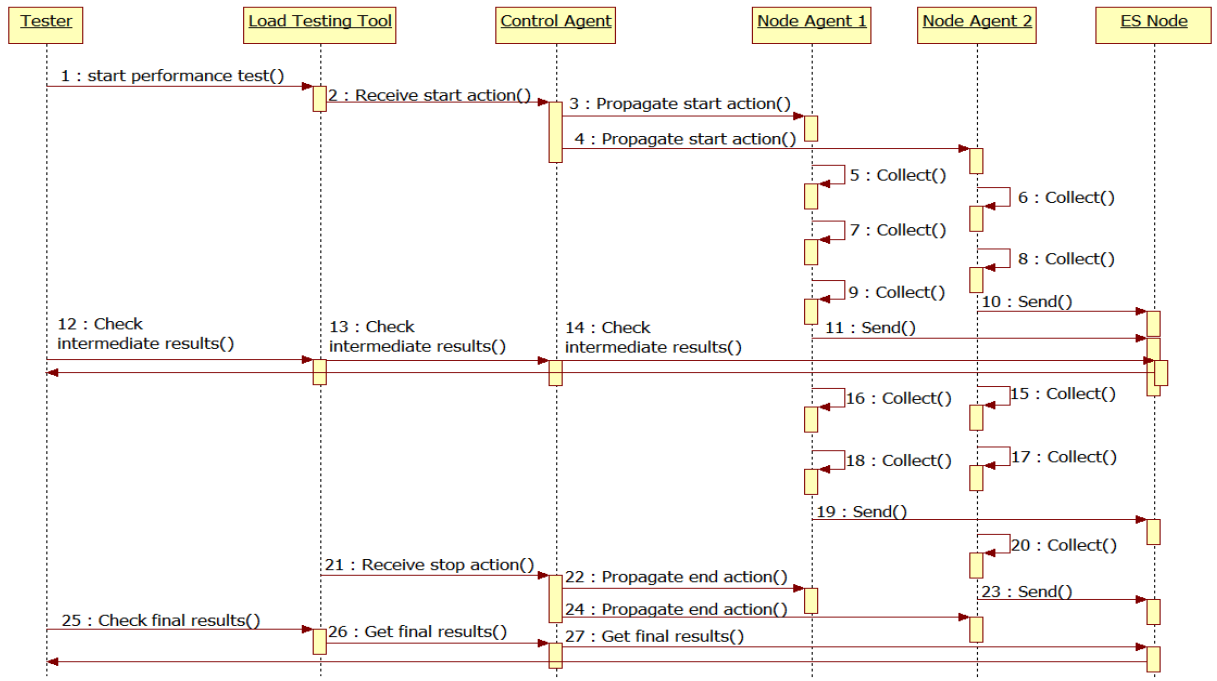

Figure 4: Automated framework - Sequence diagram

core process which coordinates the other MAPE-K components. The process is depicted in Figure 3. It is triggered when the performance test starts. As an initial step, it gets a new *Control Test Id*, value which will uniquely identify the test run and its collected data. This value is propagated to all the nodes. Next all application nodes start the following loop (in parallel) until the performance test finishes: A new set of data samples is collected following a data gathering policy. After the collection finishes, the process checks if any upload policy has been fulfilled. If it does, the data is sent to the expert system (labeling the data with the *Control Test Id* so that information from different nodes can be identified as part of the same test run). Similarly, updated results are retrieved from the expert system in order to be consolidated. Additional policies might also be executed depending on the configuration. For example, as certain data collections can be costly (i.e. the generation of a memory dump in Java can take minutes and require hundreds of megabytes of hard disk), a back-up policy could be used to enable further off-line analysis of the collected data. This core process continues iteratively until the performance test finishes. When that occurs, all applicable policies are evaluated one final time before the process ends. Furthermore, any exceptions are internally handled and then reported.

Architecture. The proposed framework is implemented with the multi-agent architecture presented in Figure 1. This is composed of three types of agents: The *Control Agent* is responsible of interacting with the load testing tool to know when the test starts and ends. It is also responsible of evaluating the adaptive policies and propagate the decisions to the other nodes. The second component is the *Application Node Agent* which is responsible to perform any required tasks in each application node. Similarly, the *Expert System Agent* is responsible of interfacing with the expert system.

These components communicate through commands, following the *Command*[7] Design Pattern: The *Control Agent* invokes the commands, while the other agents implement the logic in charge of executing each concrete command. An example of these interactions is depicted in Figure 4: Once a tester has started a performance test (step 1), the *Control Agent* propagates this action to all the *Application Node Agents* (steps 2 to 4). Then each *Application Node Agent* performs its periodic tasks (steps 5 to 9) until any of the configured upload policies is fulfilled and the data is sent to the expert system (steps 10 and 11). These steps continue iteratively until the test ends. At that moment, the *Control Agent* propagates the stop action (steps 21,22 and 24). At any time, the tester might choose to review the intermediate results of the expert system (steps 12 to 14) until getting the final results (steps 25 to 27).

Prototype. Based on our proposed framework, a prototype has been developed in conjunction with our industrial partner IBM. The *Control Agent* was implemented as a plugin for the Rational Performance Tester (RPT) [8], which is a common load testing tool in the industry; the *Application Node Agent* was implemented as a Java Web Application; WAIT was the selected expert system due to its analysis capabilities (discussed in Section 2). Finally, the *Expert System Agent* was implemented in PHP in order to extend the web interface of WAIT (which is developed in that technology).

Additionally, two policies were implemented: One data gathering policy which uses a constant *Sample Interval* (SI) during the complete test execution. Similarly, an upload policy which uses a constant *Upload Time Threshold*(UTT).

[7]http://www.oodesign.com/command-pattern.html
[8]http://www-03.ibm.com/software/products/us/en/performance

4. ASSESSMENTS OF WAIT TRADE-OFFS

To understand which adaptive policies would be more adequate for WAIT, an assessment of its performance trade-offs were done. It involved two tests. The first one evaluated the overhead introduced by the data gathering process of WAIT in the application nodes. The second one measured the resource utilisation in the WAIT server during the processing of those samples.

All tests were done in an environment composed of eight VMs: Five application nodes, one WAIT server, one load balancer and one load tester (using RPT 8.5). All VMs had the following characteristics: 2 virtual CPUs, 3GB of RAM, and 50GB of HD; running Linux Ubuntu 12.04L 64-bit, and Oracle Hotspot JVM version 7 with a heap of 2GB. The application nodes also ran an Apache Tomcat 6.0.35.

The DaCapo[9] benchmark 9.12 was chosen because it offers a wide range of different application behaviours to test. For each benchmark, its smallest *Sample Size* was used. In order to invoke the Dacapo benchmarks from within a RPT HTTP test script, a wrapper JSP was developed. It allowed the execution of any DaCapo benchmark via an input parameter. Finally, a 24-hour test duration was chosen to reflect more realistic test conditions.

The next sections describe the results of the performed tests.

Overhead in the Application Nodes. This test involved the assessment of the throughput (transactions per second) and the identified bugs during the data gathering process of WAIT. These metrics were collected through RPT and the WAIT report, respectively.

As the SI controls the frequency of samples collection from the monitored application (which is the main potential cause of overhead introduced by WAIT), a broad range of values was tested (7, 15, 30, 60, 120, 240, 360, 480, 720 and 960 seconds). Purposely, the smaller value in the range (7 seconds) was chosen to be smaller than the minimum recommended value for WAIT (30 seconds). Similarly, the largest value in the range (960 seconds) was chosen to be larger than 480 seconds (a SI commonly used in the industry). As the UTT has minimum impact in the performance of the application nodes (only controlling when the samples are sent to the WAIT server), a constant value of 30 minutes was used.

The obtained results showed that there is a relationship between the selection of the SI and the performance cost of using WAIT. This behaviour is depicted in Figure 5, which summarizes the results of the tested configurations. It can be noticed how the throughput decreases when the SI decreases. This performance impact is mainly caused by the *Javacore* generation process which pauses the JVM during its execution. Even though its cost was minimum when using higher SIs, its performance impact gradually became visible (especially when using SIs below 30 seconds, which is the minimum recommended SI for WAIT). On the contrary, the number of identified bug increases when the SI decreases. This positive impact is a direct consequence of feeding more samples to the WAIT server, situation which allows WAIT to do a more detailed analysis of the monitored application. A second round of analysis concentrated on the most critical issues identified by the WAIT report (those ranked with a frequency above 80% and which would be the most relevant for a tester) to assess if the previously described behaviours were also observed there. That analysis confirmed the presence of similar behaviours, as depicted in Figure 6.

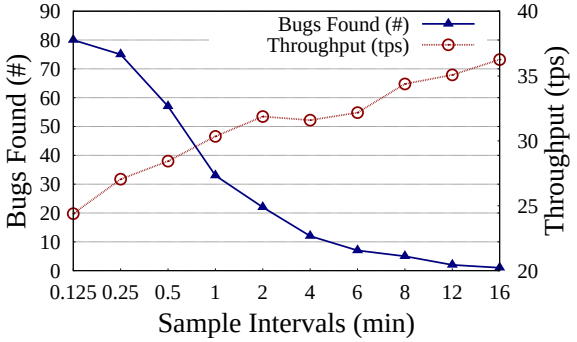

Figure 5: Bug Coverage vs. Throughput

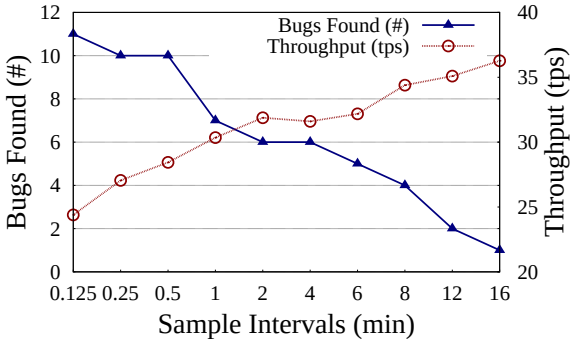

Figure 6: Critical Bug Coverage vs. Throughput

Overhead in WAIT Server. This test involved the assessment of the CPU (%) and memory (MB) utilization in the WAIT Server during the processing of the samples. These metrics were collected using the "top" command.

As both SI and UTT influence the number of samples that are sent to the WAIT server to be processed (as the lower the SI, the more samples to be processed. Similarly, the higher the UTT, the more samples to be processed), a range was chosen for each parameter. For the SI, the following 3 values were used: 30, 240 and 480 seconds. For the UTT, the following 3 values were used: 5, 30 and 60 minutes.

The results showed how the resource utilisation in the WAIT server is related to the number of samples processed in parallel; which is a function of both the SI and UTT. For example, the experimental configuration which used a SI of 240 seconds and an UTT of 30 minutes, reported similar resource utilisations than the configuration which used a SI of 480 seconds and an UTT of 60 minutes, as both combinations fed the same number of samples (per upload iteration) to the WAIT server.

Even though the CPU and memory utilisations showed simi-

[9]http://dacapobench.org/

lar trends, the WAIT server proved to be more CPU-intensive (with its CPU_{AVG} and CPU_{MAX} exceeding the 90% utilisation in most of the tested configurations). On the contrary, the WAIT server was considerable less memory-intensive (with its maximum MEM_{AVG} around 23MB, and the maximum MEM_{MAX} around 28MB). These results are presented in Figures 7 (CPU) and 8 (memory).

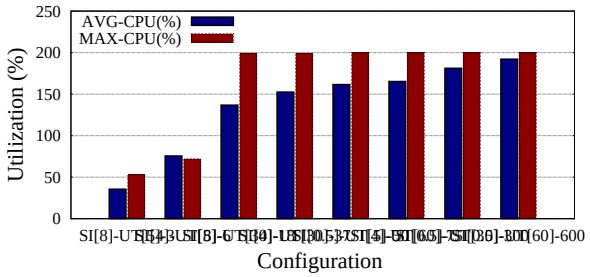

Figure 7: WAIT Server - CPU utilisations

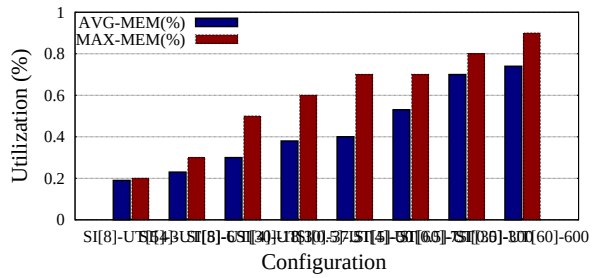

Figure 8: WAIT Server - Memory utilisations

Conclusions. The performed tests proved that the selection of SI influences the performance impact that using WAIT might provoke in the monitored application. Similarly, the selection of UTT influences the resource utilisation in the WAIT server. These behaviours make the selection of both settings good candidates to become adaptive policies.

5. ADAPTIVE POLICIES

The following sections describe our proposed policies, which are based on the outcomes of the previously described assessment.

Accuracy-Target Data Gathering Policy. This adaptive policy was designed to balance the performance trade-offs (between precision and performance) caused by the selection of the SI.

The detailed policy is depicted in Figure 9, where each step is mapped to the corresponding MAPE-K element. This policy requires two user inputs: The response time threshold, which is the maximum acceptable impact to the response time (express as a percentage); and the warm-up period, which is the time after which all the types of transactions have been executed at least one time (hence contributing at least once to the average response time of the test run). Two additional parameters are required. As their values are specific for each expert system, they would be retrieved from our policy base: The minimum SI that should be used for

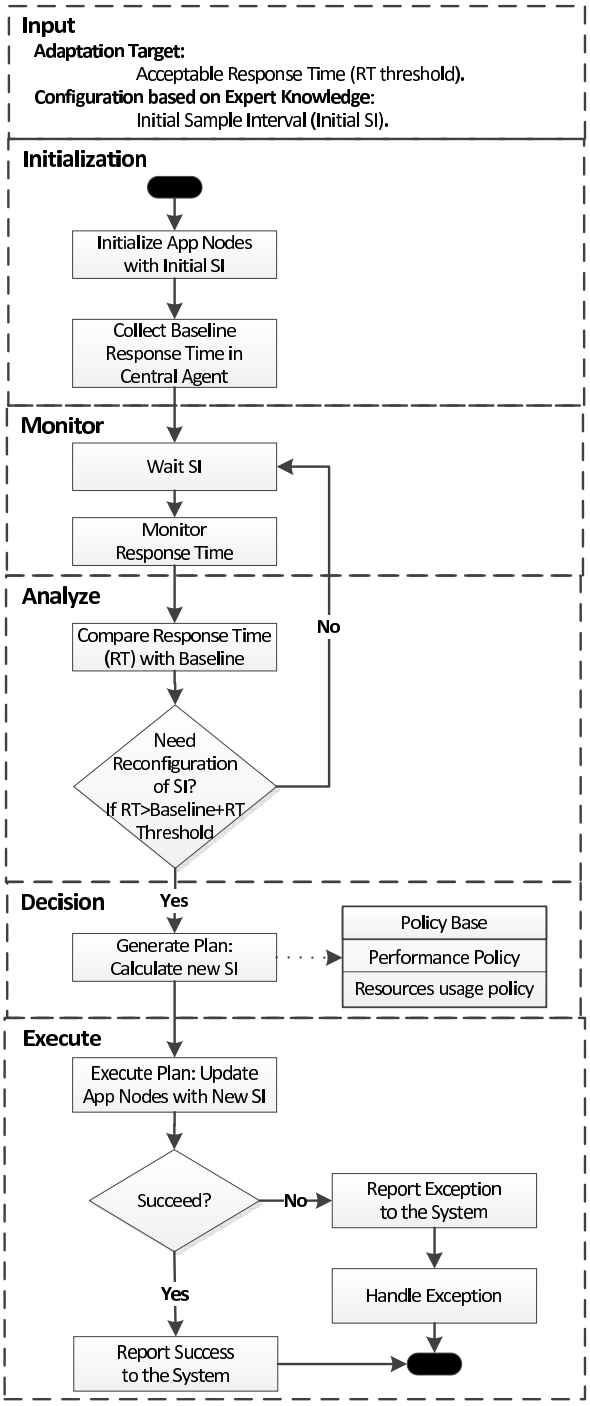

Figure 9: Accuracy-Target Data Gathering Policy

collection; and the ΔSI, which indicates how much the SI should change in case of adjustment.

The process starts by waiting the configured warm-up period. Then it retrieves the average response time (RT_{AVG}) from the load testing tool. This value becomes the response time baseline (RT_{BL}). After that, the process initialises the SI of all the application nodes with the minimum SI. This strategy allows to collect as many samples as possible, un-

less the performance degrades below the desired threshold. Next, an iteratively monitoring process starts (which lasts until the performance testing finishes): First, the process waits the current SI (as no performance impact caused by the expert system might occur until the data gathering occurs). Then, the new RT_{AVG} is retrieved and compared against the RT_{BL} to check if the threshold has been exceeded. If so, it means that the current SI is too small to keep the overhead below the configured threshold. In this case, the SI is increased by the value configured as ΔSI. Finally, the new SI is propagated to all the application nodes, which start using it since their next data gathering iteration.

Finally, it is worth mentioning that this policy was inspired on the scenario (commonly experienced in the industry) where a test team aims to minimise the number of required performance tests due to budget or schedule constraints. This might be achieved by allowing certain level of known overhead in the tested application in order to identify as many performance bugs as possible, in addition to the normal results obtained from a performance test.

Efficiency-Target Upload Policy. To minimise the resource utilisation caused by the selection of UTT, the following adaptive policy was designed. The detailed policy is depicted in Figure 10, where each step is mapped to the corresponding MAPE-K element. It requires two user inputs: The initial UTT to be used; and the ΔUTT, which indicates how much the UTT should change when an adjustment is required. An additional parameter would be retrieved from our policy base (as its value is specific to each type of resource): The target of maximum utilisation (ToMU). As documented in [24], the objective of this target is to retains certain unused capacity to provide a soft assurance for quality of service. In the case of the CPU, this target is 90%.

The process starts by initialising the UTT of all the application nodes with the initial UTT. Then the process awaits until all the application nodes have uploaded their samples one time. This step is done to make sure that the UTT is only modified if required. After all the nodes have uploaded their results, the process retrieves the CPU_{AVG} of the shared service during the processing of those samples, as well as the average duration of the CPU peak usage ($CPU-D_{AVG}$). Then the CPU_{AVG} is compared with the ToMU. If the ToMU has been exceeded, new UTTs are calculated.

As a first strategy, a different UTT is calculated for each node to prevent that all the nodes upload their results at the same time. To respect as much as possible the initial UTT, the calculation of the new UTTs is based on the current UTT (by iteratively subtracting or adding the $CPU_{AVG}-D$ from the current UTT until all nodes have a new UTT). For example, if we have 5 nodes, a current UTT of 30 minutes and a $CPU-D_{AVG}$ of 1 minute, the new UTTs of the nodes would be 28,29,30,31 and 32. Finally, the new UTTs are propagated to all the application nodes, which start using them since the next upload iteration.

In case a subsequent adjustment is required (meaning that only splitting the UTT was not enough to bring the CPU_{AVG} below the ToMU), the current UTT is decreased (by the value configured as ΔUTT), before the calculation of the new UTTs is done. This second strategy aims to reduce the number of samples sent (by each node) per upload iteration.

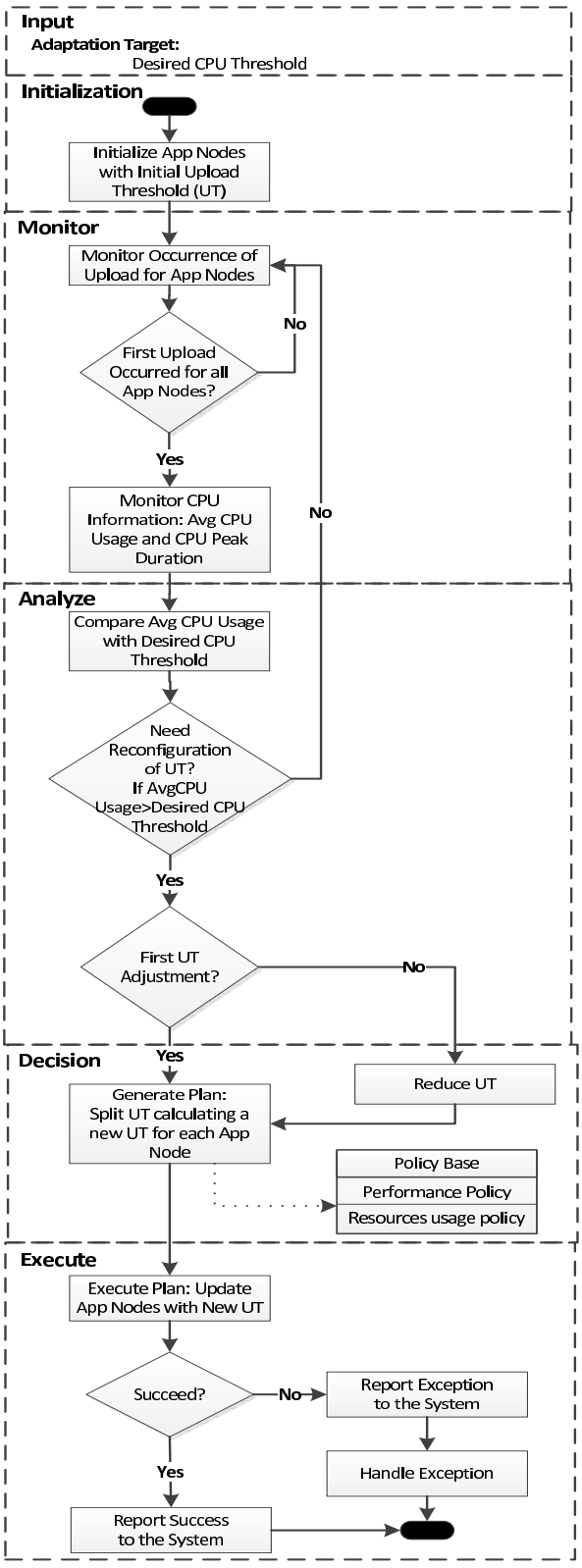

Figure 10: Efficiency-Target Upload Policy

6. EXPERIMENTAL EVALUATION

Two additional experiments were performed to evaluate the performance of our proposed adaptive framework. The first experiment evaluated the accuracy of the implemented adaptive policies, while the second experiment assessed the productivity gains that our framework brought to the performance testing process. The next sections describe these experiments, their results, and the main threats of validity.

Assessment of Adaptive Policies. The objective of this experiment was to evaluate if the adaptive policies fulfilled their purpose of addressing the identified trade-offs without the need of manual intervention from the tester's side. The experimental set-up was equal to the one used in the assessments of the WAIT trade-offs (Section 4) with except of the selection of SI and UTT, as the adaptive policies took the place of the manual configurations of these parameters.

The adaptive policies used the following configurations: For the accuracy policy, a 20% threshold was defined. This configuration was suggested by IBM SVT to reflect real-world conditions. Additionally, a warm-up period of 5 minutes was found to be enough for all the test transactions to be completed at least one. Finally, the minimum SI and the ΔSI were set to 30 seconds. Regarding the efficiency policy, the initial UTT was set to 30 minutes (as it is a time range commonly used in the industry to monitor performance tests); and the ΔUTT was set to 10 minutes. Finally, the CPU maximum utilisation threshold was set to 90%.

The obtained results were compared against the results from the previously performed assessment of the WAIT trade-offs (Section 4). The first part of our analysis focused on evaluating the accuracy policy. In terms of performance overhead, the results demonstrated that the accuracy policy worked, as it was possible to finish the test with the overhead caused by WAIT within the desired threshold. This was the result of increasing the SI once the threshold was exceeded (during the test execution) to reduce its performance impact. In our case, this adjustment involved that the SI increased two times, moving from its initial value of 30 seconds to a final value of 90 seconds. Regarding bug coverage, the number of bugs found with the adaptive policy was higher than the number of bugs found with the corresponding static SI of 90 seconds. This was the result of using other (smaller) SIs during the test, situation which provoked that the bug coverage was higher (compared to a SI of 90 seconds) during certain periods of the test. The same analysis was done considering only the critical bugs, and similar behaviours were observed. These results are presented in Figures 11 (for the overall bugs) and 12 (for the critical bugs).

The second part of our analysis concentrated on evaluating the efficiency policy. The obtained results showed that the efficiency policy achieved its goal of decreasing the utilisation in the shared service (the WAIT server in this scenario). In our case, the CPU_{AVG} and CPU_{MAX} of the first round of uploads (which occurred before any adjustment) were 90.7% and 95.0%, respectively. As these metrics exceeded the target of maximum utilisation (ToMU), the efficiency policy adjusted the UTTs of the nodes after the first round of uploads. After the UTT adjustment, the CPU_{AVG} and CPU_{MAX} decreased to 65.7% and 75.0% (remaining below the ToMU during the rest of the test). This behaviour is depicted in Figure 13 which visually shows the CPU utilisation for one of the performed experiment using our efficiency policy. The first peak is the CPU utilisation before the UTT adjustment, while the second set of peaks is the CPU utilisation after the adjustment.

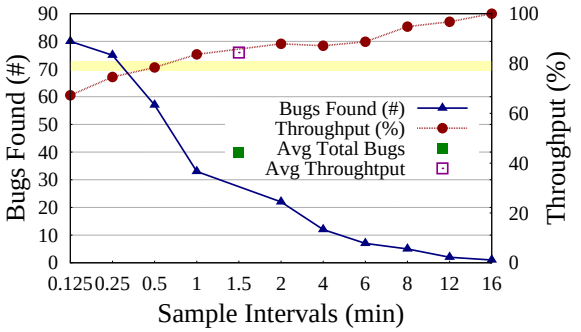

Figure 11: Overall Bug Coverage vs. Throughput

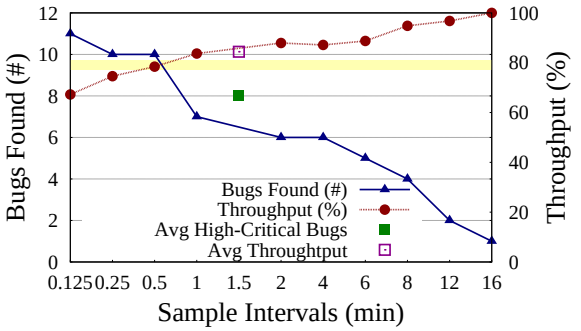

Figure 12: Critical Bug Coverage vs. Throughput

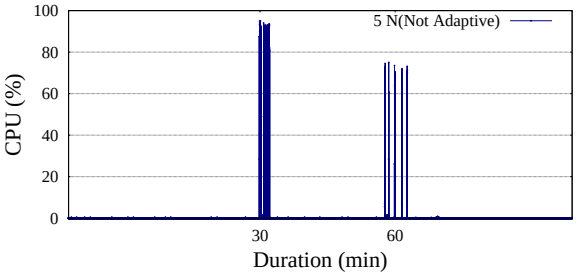

Figure 13: WAIT Server - CPU utilisation

In conclusion, the results of this experiment demonstrated that our proposed adaptive policies achieved their respective goals: The accuracy policy kept the performance trade-offs within the desired threshold. Meanwhile, the efficiency policy decreased the resource utilisation of the shared service, minimising the possibility of its saturation.

Assessment of benefits in testing productivity. Here the objective was to assess the benefits our framework brings to a performance tester in terms of reduced effort and time. The experimental set-up was similar to the one used in the previous experiment with three exceptions: First, the iBatis

JPetStore 4.0 [10] application was selected in order to test our framework with a different application behaviour. Second, the number of application nodes was increased (to 10 nodes) to test our framework in a bigger test environment.This experiment also involved modifying the source code of JPetStore to inject five performance issues.

Two types of runs were performed: The first type involved a tester trying to identify the injected bugs using WAIT manually (M-WAIT). A second type of run involved using WAIT through our automation framework (A-WAIT). In both cases, the tester did not know the number or characteristics of the injected bugs.

The results of these experiments are summarized in Table 1. After comparing the results of both runs, two time savings were documented when using the automated WAIT: First, the effort required to identify bugs was decreased (68% less than the manual WAIT). This time saving was the result of simplifying the analysis of the WAIT reports: Instead of having multiple reports (one per node) that needed to be analysed and manually correlated, the tester using the automated WAIT only needed to keep monitoring a single report which incrementally evolved. The second saving involved the time required by the tester to identify all the injected bugs. By using the automation framework, it was possible to feed WAIT incrementally during the performance test execution (in contrast to manual WAIT, where the tester needed to wait until the end of the performance test). This behaviour allowed the tester using the automated WAIT to easily get intermediate results during the test run. In our case, all the bugs were identified by the tester using the automated WAIT after the first hour of test execution. Therefore, she was able to start the analysis of those bugs in parallel to the rest of the performance test execution (which the tester kept monitoring). A direct consequence of this second time saving was that the overall duration of the performance testing activity decreased 27%. For the tester using the automated WAIT, the activity practically lasted only the planned 24-hour duration of the performance test, plus some additional time required to review the final consolidated WAIT report. It is also worth mentioning that both testers were able to identify all the injected bugs with the help of the WAIT reports.

Table 1: M-WAIT and A-WAIT Comparison

Metric	M-WAIT (hr)	A-WAIT (hr)	M-WAIT vs. A-WAIT (%)
a.Duration of Performance testing activity (b+c)	33.2	24.1	-27%
b.Duration of Performance Testing	24.0	24.0	0%
c.Effort of Performance Analysis (d+e)	8.5	4.8	-44%
d.Effort of Bug Identification	6.5	2.1	-68%
e.Effort of Root Cause Analysis	2.0	2.0	0%

An additional observation from this experiment is that the time savings gained by the automated framework are directly related to the duration of the test and the number of application nodes in the environment. This expected behaviour is especially valuable in long-term runs, which are common in performance testing and typically last several days. The same situation occurs with the performance testing of highly distributed environments, as the obtained time savings will be higher under those conditions.

To summarize the experimental results, they allowed to measure the productivity benefits that a tester can gain by using an expert system through our proposed automation framework. In particular, two time savings were documented: The effort required to identified bugs was reduced (68% in our case), as well as the total duration of the activity (27% in our case). A direct consequence of these time savings is the reduction in the dependence on human expert knowledge and a reduced effort required by a tester to identify performance issues, hence improving the productivity.

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. To mitigate this, all tests were done in an unloaded environment and multiple runs were executed for each identified combination. Another threat was the selection of the tested applications. Their limited number implies that not all types of applications have been tested and wider experiments might be needed to get more general conclusions. However, there is no reason to believe that the presented framework cannot be applied to other usage scenarios. Additionally, a key motivation to select DaCapo was that it offers a wide range of application behaviours, hence helping to mitigate this risk.

7. CONCLUSIONS AND FUTURE WORK

The identification of performance problems in highly distributed environments is complex and time-consuming. Even though researchers have been developing expert systems to simplify this task, various limitations exist in those tools that prevent their effective usage in performance testing. To address these limitations, this work proposed a novel adaptive framework to automate the usage of an expert system in a distributed testing environment. A prototype was developed around the WAIT tool and then its benefits were assessed. The results showed the time savings gained by applying the proposed framework. In our case, the effort required to identified bugs was reduced 68%, while the total duration of the performance testing activity was reduced 27%. The results also demonstrated that the adaptibility capabilities of our framework fulfilled their purpose of simplifying the configuration of the expert system. This was done by addressing the identified WAIT trade-offs without the need of manual intervention from the tester's side. Thus, the approach was shown to simplify the usage of an expert system and to reduce the time required to analyze performance issues, thereby reducing the costs associated with performance testing. Future work will focus on investigating how best to extend the adaptive capabilities of our framework (i.e. adding data analytic policies to post-process the outputs of the expert systems).

8. ACKNOWLEDGMENTS

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

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

9. REFERENCES

[1] J. Albert, Elvira, Miguel Gomez-Zamalloa. Resource-Driven CLP-Based test case generation. *Logic-Based Program Synthesis and Transformation*, 2012.

[2] E. Altman, M. Arnold, S. Fink, and N. Mitchell. Performance analysis of idle programs. *ACM SIGPLAN Notices*, 45(10):739, Oct. 2010.

[3] G. Ammons, J.-d. Choi, M. Gupta, and N. Swamy. Finding and Removing Performance Bottlenecks in Large Systems. In *ECOOP*, 2004.

[4] V. Angelopoulos, T. Parsons, J. Murphy, and P. O'Sullivan. GcLite: An Expert Tool for Analyzing Garbage Collection Behavior. *2012 IEEE 36th Annual Computer Software and Applications Conference Workshops*, pages 493–502, July 2012.

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

[6] M. Barr, E., Christian Bird. Collecting a Heap of Shapes. *ISSTA*, 2013.

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

[8] M. Y. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer. Path-based failure and evolution management. *NSDI*, 2004.

[9] S. Chen, D. Moreland, S. Nepal, and J. Zic. Yet Another Performance Testing Framework. *Australian Conference on Software Engineering*, 2008.

[10] Compuware. *Applied Performance Management Survey*. 2007.

[11] S. DÃűsinger, Stefan, Richard Mordinyi. Communicating continuous integration servers for increasing effectiveness of automated testing, 2012.

[12] e. G. Jin, L. Song. Understanding and detecting real-world performance bugs, 2012.

[13] A. Haroon Malik, Bram Adams. Pinpointing the subsys responsible for the performance deviations in a load test. *Software Reliability Engineering*, 2010.

[14] V. Hoorn, M. Rohr, W. Hasselbring, J. Waller, J. Ehlers, and D. Kieselhorst. ÃČâĂŽÃĆÂÍ INFORMATIK INSTITUT F UR Continuous Monitoring of Software Services : Design and Application of the Kieker Framework CHRISTIAN-ALBRECHTS-UNIVERSIT AT Continuous Monitoring of Software Services : Design and Application of the Kieker Framework. (0921), 2009.

[15] D. J. Kephart. The vision of autonomic computing, Jan. 2003.

[16] S. J. Zhang. Automated test case generation for the stress testing of multimedia systems. *Softw. Pract. Exper.*, 2002.

[17] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora. Automated performance analysis of load tests. In *IEEE International Conference on Software Maintenance*, pages 125–134. Ieee, Sept. 2009.

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

[19] e. Li, Mengchen. Dynamically Validating Static Memory Leak Warnings. *ISSTA*, 2013.

[20] J. M. S. Bayan. Automatic stress and load testing for embedded systems. *International Computer Software and Applications Conference*, 2006.

[21] L. M. Salehie. Self-adaptive software: Landscape and research challenges, 2009.

[22] H. P. Robertson, R. Laddaga. Introduction: the First International Workshop on Self-Adaptive Software. *IWSAS*, 2000.

[23] T. Parsons and J. Murphy. Detecting Performance Antipatterns in Component Based Enterprise Systems. In *2nd International Middleware Doctoral Symposium*, volume 7, pages 55–90, 2008.

[24] M. Rolia, Jerry, Artur Andrzejak. Automating enterprise application placement in resource utilities, 2003.

[25] M. S. Hangal. Tracking down software bugs using automatic anomaly detection. *International Conference on Software Engineering*, 2002.

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

[27] W. Spear, S. Shende, A. Malony, R. Portillo, P. J. Teller, D. Cronk, S. Moore, and D. Terpstra. Making Performance Analysis and Tuning Part of the Software Development Cycle. *2009 DoD High Performance Computing Modernization Program Users Group Conference*, pages 430–437, June 2009.

[28] Y. V. Garousi, L. C. Briand. Traffic-aware stress testing of distributed systems based on uml models. *International conference on Software engineering*, 2006.

[29] J. Weyns, Danny, M. Usman Iftikhar. Do external feedback loops improve the design of self-adaptive systems? a controlled experiment, 2013.

[30] M. Woodside, G. Franks, and D. C. Petriu. The Future of Software Performance Engineering. *Future of Software Engineering (FOSE '07)*, pages 171–187, May 2007.

[31] T. Wu, Haishan, Asser N. Tantawi. A Self-Optimizing Workload Management Solution for Cloud Applications. 2012.

[32] e. Xiao, Xusheng. Context-Sensitive Delta Inference for Identifying Workload-Dependent Performance Bottlenecks, 2013.

[33] W. Xingen Wang, Bo Zhou. Model-based load testing of web applications, 2013.

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

[35] e. Yu, Kai. Practical isolation of failure-inducing changes for debugging regression faults, 2012.

[36] G. Yu, Tingting, Witawas Srisa-an. SimRacer: an automated framework to support testing for process-level races., 2013.