PHOEBE: an automation framework for the effective usage of diagnosis tools in the performance testing of clustered systems

A Omar Portillo-Dominguez, Philip Perry, Damien Magoni, John Murphy

PHOEBE: An Automation Framework for the Effective Usage of Diagnosis Tools in the Performance Testing of Clustered Systems

A. Omar Portillo-Dominguez[1,*], Philip Perry[1], Damien Magoni[2] and John Murphy[1]

[1]*Lero, School of Computer Science, University College Dublin, Ireland*
[2]*LaBRI, University of Bordeaux, France*

SUMMARY

Nowadays, clustered environments are commonly used in enterprise-level applications to achieve faster response time and higher throughput than single machine environments. However, this shift from a monolithic architecture to a distributed one has augmented the complexity of these applications, considerably complicating all activities related to the performance testing of such clustered systems. 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, many researchers have been developing tools with built-in expertise for practitioners. However, various limitations exist in these tools that prevent their efficient usage in the performance testing of clusters (e.g., the need of manually analysing huge volumes of distributed results). To address these limitations, our previous work introduced a policy-based adaptive framework (PHOEBE) which automates the usage of diagnosis tools in the performance testing of clustered systems, in order to improve a tester's productivity by decreasing the effort and expertise needed to effectively use such tools. The aim of this paper is to extend our previous work by broadening the set of policies available in PHOEBE, as well as by performing a comprehensive assessment of PHOEBE in terms of its benefits, costs and generality (with respect to the used diagnosis tool). The performed evaluation involved a set of experiments to assess the different trade-offs commonly experienced by a tester when using a performance diagnosis tool, as well as the time savings that PHOEBE can bring to the performance testing and analysis processes. Our results have shown that PHOEBE can drastically reduce the effort required by a tester to do performance testing and analysis in a cluster. PHOEBE also experienced a consistent behaviour, when applied to a set of commonly used diagnosis tools, demonstrating its generality. Finally, PHOEBE proved to be capable of simplifying the configuration of a diagnosis tool. This was achieved by addressing the identified trade-offs without the need for manual intervention from the tester. These results offer practitioners a valuable reference regarding the benefits that an automation framework, focused on effectively addressing the common usage limitations experienced by a diagnosis tool, can bring to the performance testing of clustered systems.

KEY WORDS: Performance Testing; Performance Analysis; Cluster Computing; System Performance

1. INTRODUCTION

Performance is a critical dimension of quality and a major concern of any software project. However, it is not uncommon that performance issues occur and materialise into serious problems in a significant percentage of applications (e.g., outages on production environments or even cancellation of software projects). For example, a 2007 survey conducted with information technology executives [1] reported that 50% of them had faced performance problems in at least 20% of their deployed applications. This situation is partially explained by the pervasive nature of

performance, which makes it hard to assess because performance is practically influenced by every aspect of the design, code, and execution environment of an application.

In recent years, cluster computing has gained popularity as a powerful and cost-effective solution for parallel and distributed processing [2]. Thus, the usage of clusters is becoming ubiquitous: Modern high-assurance systems and enterprise-level applications, which usually require both fast response time and high throughput on a constant basis, are commonly deployed in clustered instances to fulfil such stringent performance requirements. However, this shift from a monolithic architecture to a distributed one has also augmented the complexity of these applications, further complicating all activities related to performance [3].

Under these conditions, it is not surprising that doing performance testing is complex and time-consuming. A special challenge, documented by multiple authors [4–7], is that current performance diagnosis 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 utilisation and memory usage, are a few examples of the information that a tester needs 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 organisation [8]. Therefore, this issue could potentially lead to bottlenecks where certain activities can only be done by these experts, impacting the productivity of the testing teams [4].

To simplify the performance analysis and diagnosis, many researchers have been developing tools with built-in expertise [4, 9, 10]. However, limitations exist in these diagnosis tools that prevent their efficient usage on highly distributed environments. Firstly, these tools still need to be manually configured, according to various sensitive parameters which need to be tweaked to avoid bad impacts on the accuracy of the tools' outputs. If an inappropriate configuration is used, the tools might fail to obtain the desired outputs, resulting in significant time wasted. In addition, to use these tools, testers need to manually carry out data collections. In a clustered environment, where multiple nodes need to be monitored and coordinated, such a manual process can be very time-consuming and error-prone due to the vast amount of data to collect and consolidate. In a long running performance test scenario, such a manual usage of diagnosis tools is more difficult due to the many periodical data collection processes. Similarly, excessive amounts of outputs produced by the tools can overwhelm a tester due to the time required to correlate and analyse the results. This problem is caused by the multiple reports which are commonly produced per monitored application node, information which needs to be manually correlated and analysed.

To ensure that our research can be usefully applied to solve real-life problems in the software industry, a research collaboration has been carried out with one industrial partner, the IBM System Verification Team, in order to identify and understand the challenges in their day-to-day activities. Their feedback confirms that there is a real need for techniques that facilitate the identification of performance issues, especially in the testing of large-scale environments.

This discussion motivates the core research question here: "What techniques can be developed to automatically conduct the performance analysis and bug diagnosis tasks in a cluster for improving its performance testing, while avoiding errors and saving time?". To address this challenge, our research work has centred on addressing the common usage limitations experienced by a diagnosis tool in order to be effectively used in the performance testing of clustered applications. The aim has been to improve a tester's productivity by decreasing the effort and expertise needed to use diagnosis tools. In our previous work [11] we presented PHOEBE, an adaptive framework that automates the configuration and usage of a diagnosis tool in a clustered testing environment (typically located within a data centre). PHOEBE is shown in Figure 1. There it can be noticed how PHOEBE executes concurrently with a performance test run, shielding the tester from the complexities of properly configuring and using the diagnosis tool, so that the tester only needs to interact with the load testing tool. Internally, PHOEBE leverages on policies to automatically control the different processes commonly involved in the usage of a performance diagnosis tool.

In this paper, we extend our previous work by broadening the set of policies available in PHOEBE to cover the whole spectrum of processes (i.e. sample gathering, sample processing and results'

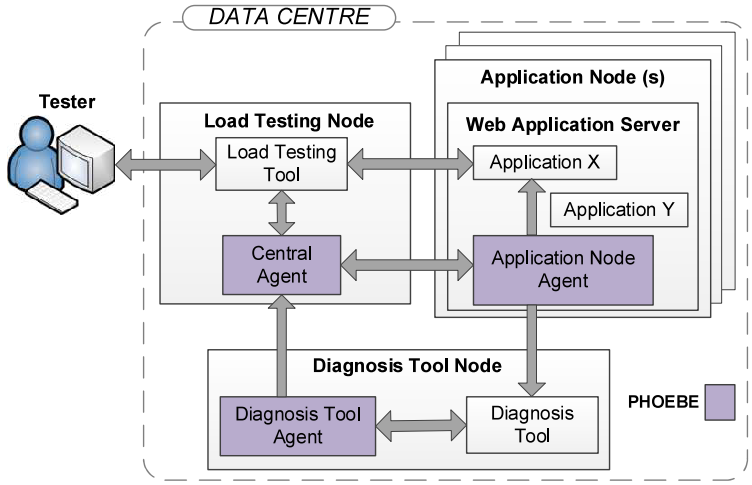

Figure 1. PHOEBE: An Automation Framework for Performance Testing.

consolidation) normally involved on the usage of a diagnosis tool in the performance testing of a clustered application (as previously we had only covered the sample gathering). Furthermore, we performed a comprehensive assessment of PHOEBE in terms of its benefits, costs and generality (with respect to the used diagnosis tool). For this purpose, PHOEBE was applied to five well-known diagnosis tools (as previously we had only applied it to one tool). The performed experimental evaluation was composed of five experiments. In the first three experiments, the different trade-offs that are commonly experienced when using a diagnosis tool (in terms of bug finding accuracy, testers' effort and resource utilisations) were evaluated. Then, the other two experiments focused on assessing the time/effort savings (in the processes of performance testing and analysis) obtained by a tester through the usage of PHOEBE and its set of policies.

The contributions of this paper are the following:

1. An extended description of our policy-based adaptive automation framework (PHOEBE), whose goal is to address the common usage limitations experienced by a diagnosis tool to be effectively used in the performance testing of clustered applications.
2. Two policies to self-configure the gathering and processing of data samples in a diagnosis tool. They are based on a set of configurable thresholds to control the performance trade-offs of using a diagnosis tool. Additionally, one policy to automatically consolidate and analyse the results produced by a diagnosis tool. It is based on a set of configurable assessments to customise how the frequency and severity of the identified issues are evaluated.
3. A comprehensive practical evaluation of PHOEBE and its policies, consisting of a prototype implementation around a set of five well-known performance diagnosis tools and a set of five experiments. These experiments demonstrate the accuracy of the framework as well as its productivity benefits.
4. Key findings that could serve as guidelines for practitioners to know the conditions under which the framework can be useful as well as when each policy can be more suitable.

The remainder of the paper is structured as follows: Sections 2 and 3 present the pertinent state of the art, in terms of background information and related work, respectively. Section 4 explains the internal workings of PHOEBE; while Section 5 discusses the performed experiments and their results. Finally, Section 6 presents the conclusions of this research work and provides pointers to future work in this area.

2. BACKGROUND

This section presents the main characteristics of a typical performance testing process, as well as the set of chosen performance diagnosis tools, which are necessary to understand the rest of the paper.

2.1. Performance Testing

When an application is tested during development, it is important not only to test its ability to perform the desired business functions through functional testing, but also to evaluate how well the application performs those functions when multiple concurrent users are accessing it. This is the goal of the performance testing, which aims to evaluate the behaviour of an application under a given workload [12]. In a traditional software development process (i.e. waterfall), performance testing is usually performed towards the end of the process and repeated for each new built version of the application (as shown in Figure 2). Meanwhile, in an agile software development process, performance testing is normally performed at the end of each iteration (or group of iterations).

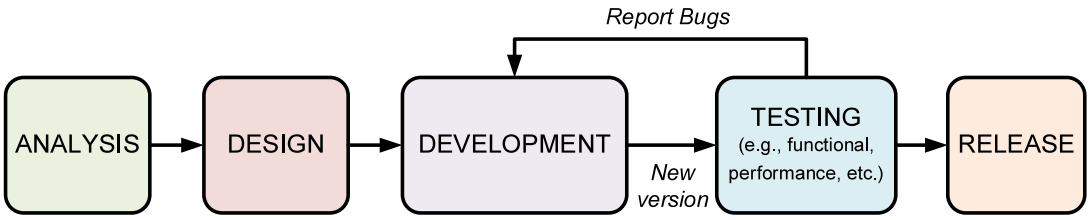

Figure 2. Traditional Software Development Process.

A performance test run involves exposing an application to a workload that resembles its expected real-life conditions. This is achieved by using a load generator (e.g., IBM Rational Performance Tester [13], Apache JMeter [14] or HP LoadRunner [15]) for an extended period of time (e.g., 24-hours or even longer durations) to simulate the desired set of concurrent users interacting with the application.

During the execution of a performance testing run, testers usually collect performance-related counters. These counters are normally of two main types [16]: performance metrics (i.e. response time and throughput) and resource metrics (e.g., CPU or memory utilisations). The objective is to analyse the behaviour of the monitored counters through time, as well as compare the counters against a define performance baseline (e.g., a target Service Level Agreement), in order to identify performance anomalies. Additionally, testers usually use some type of performance diagnosis tool to further investigate the collected performance counters in order to look for anomalous behaviours and their potential root causes.

From an end-user perspective, using a performance diagnosis tool is normally simple: A tester only needs to collect as much data as desired, process it on the chosen diagnosis tool and get a report (or set of reports, depending on the tool) with the identified findings. This process can be repeated multiple times to monitor a system through time.

Given their capabilities, these diagnosis tools are normally seen as promising candidates to reduce the dependence on the human expertise and time required for performance analysis. However, the volume of data generated can be difficult to manage and the effort required to efficiently process this data can be an impediment to their adoption. Moreover, the effort required to manually collect data to feed the tools, as well as the number of reports a tester gets from the tools, are commonly linear with respect to the number of nodes and the update frequency of the obtained results. These excessive amounts of outputs produced by diagnosis tools can easily overwhelm a tester due to the time required to correlate and analyse the results. Finally, the accuracy of the tools depend on their configuration, where the preferable configuration might vary depending on the application and usage scenario. All these usage limitations make the diagnosis tools good candidates for automation, such as the automation framework proposed in this paper.

2.2. Performance Diagnosis Tools

For the purpose of this work, we have focused on diagnosis tools applicable to Java. This is because, with an estimated business impact of a hundred billion dollars yearly, Java is a predominant player at enterprise level [17, 18]. Therefore, this technology is commonly used to build clustered systems.

The range of existing performance diagnosis tools in Java is broad. Nowadays, five tools frequently used in the industry are: Eclipse Memory Analyser, IBM Garbage Collection Lite, IBM Garbage Collection and Memory Visualiser, IBM Health Center, and IBM Whole Analysis Idle Time. The following paragraphs briefly describe them.

Eclipse Memory Analyser (EMAT) [19]. It is a diagnosis tool that helps to detect memory-related performance issues. EMAT is based on non-intrusive sampling mechanisms available at the Java Virtual Machine (JVM) [20], in the form of *heapdumps* [21] (detailed snapshots of the JVM memory, offering information such as used memory per object type, object references and current locks). From a report perspective, EMAT generates one HTML report per sample. Each report presents the identified memory leaks as well as a detailed quantitative description of the objects, classes and classpaths currently used in memory.

IBM Garbage Collection Lite (GCLITE) [4]. It is a diagnosis tool that helps to detect garbage collection (GC) [22] performance issues. GC is a key feature of Java which automates most of the tasks related to memory management. However, it comes with a cost: Whenever the GC is triggered, it has an impact on the system performance by pausing the involved programs [23]. GCLITE is based on the tracing mechanisms available at the JVM, in the form of the *GC verbose* [24] (complete GC logs containing information such as the total size of the heap - i.e. the memory area -, the size of the generations -i.e. the memory sub-regions - before and after each GC, and the time taken). From a report perspective, GCLITE generates one HTML report per processed GC verbose. Each report presents the identified performance issues, classified in five categories.

IBM Garbage Collection and Memory Visualiser (GCMV) [25]. It is a diagnosis tool that helps to detect GC and memory-related performance issues. GCMV is based on the tracing mechanisms available at the JVM, in the form of the *GC verbose* [24]. From a report perspective, GCMV generates one HTML report per processed GC verbose. Each report presents the identified performance tuning recommendations, providing guidance on improvements in areas such as memory leak detection, GC performance optimisation, and heap size tuning.

IBM Health Center (HC) [26]. It is a diagnosis tool that helps to detect performance issues in Java systems by providing recommendations to improve the performance and efficiency of the system. HC is based on its own sampling mechanism that produces HCD files (detailed snapshots of the JVM state, offering information on a wide range of areas such as memory, GC, method profiling and threads). From a report perspective, HC generates one per processed sample. Each report presents the identified performance issues classified in five categories and four severity levels.

IBM Whole Analysis Idle Time (WAIT) [27]. Idle-time analysis is a methodology to identify the root causes of under-utilised resources. It is based on the observed behaviour that performance issues in multi-tier applications usually manifest as idle-time of waiting threads [9]. WAIT is a diagnosis tool that implements this methodology and has proven to simplify the detection of performance issues in Java systems [6, 9, 28]. WAIT is based on non-intrusive sampling mechanisms available at Operating System level (e.g., the "ps" command in Unix) and the JVM, in the form of *Javacores* [29] (snapshots of the JVM state, offering information such as threads, locks and memory). From a report perspective, WAIT generates one HTML report per set of processed samples. The report presents the identified performance issues, sorted by frequency and impact, as well as classified in four categories.

3. RELATED WORK

In this section, we first review the state-of-the art work in automation in testing. Then, we discuss the related work in the area of performance analysis.

3.1. 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 [30–41]. For example, the authors in [33] propose an approach to automate the generation of test cases based on specified

levels of load and combinations of resources. Similarly, [31] presents an automation framework that separates the application logic from the performance testing scripts to increase the re-usability of the test scripts. Also, the work on [42] presented a test case prioritisation approach which improves the diagnostic information of a test by minimising the loss of diagnostic quality in the prioritised test suite. Meanwhile, [35] presents a framework designed to automate the performance testing of web applications and which internally utilises two usage models to simulate the users' behaviours more realistically. Finally, the authors of [43] introduced an approach, based on a novel dynamic stochastic model, to automatically generate well-distributed grammar-based test cases.

Other research efforts have concentrated on automating specific analysis techniques. For example, [44] presents a combination of coverage analysis and debugging techniques to automatically isolate failure-inducing changes. Similarly, the authors of [45] developed a technique to reduce the number of false memory leak warnings generated by static analysis techniques by automatically validating and categorizing those warnings. Moreover, the work on [46] presented an automatic technique to assess the robustness of a piece of Java code. This is done by automatically testing all the public methods of the code (using random data) in order to try to crash the program (i.e. to throw an undeclared runtime exception).

Finally, other researchers have proposed frameworks to support different software engineering processes. For example, the authors of [47, 48] present frameworks to monitor software services. Both frameworks monitor the resource utilisation and the component interactions within a system. One focuses on Java [48] and the other focuses on Microsoft technologies [47]. Likewise, the authors of [49] proposed a framework for monitoring, managing, controlling and optimising a distributed system. It relies on an agent-based architecture in which uses real-time information to perform the supported tasks. 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 the tester from the complexities of using and configuring a performance diagnosis tool.

3.2. Performance Analysis

Multiple research efforts have aimed to improve the performance analysis processes. A major research trend has focused on identifying performance bugs and their root causes. For example, the work on [50] proposes an approach to predict the workload-dependent performance bottlenecks (WDPBs) through complexity models that infer the iteration counts of those potential WDPBs. Similarly, the work on [51] presents a technique to detect processes accessing a shared resource without proper synchronisation, and which are a common cause of problems; while the authors of [52] analysed the memory heaps of several real-world object-oriented programs and provided insights to improve memory allocation and program analysis techniques. Furthermore, the authors of [53] proposed a methodology to enhance the process of selecting the appropriate analysis tools for a particular task by defining a list of comparison criteria as well as a list of usage profiles.

A high percentage of the proposed performance analysis techniques require some type of instrumentation [54]. For example, the authors in [55] 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 [56, 57] which rely on instrumentation to dynamically infer invariants within the applications and detect programming errors; or the approach proposed by [58] 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 instrumenting the tested applications.

Furthermore, the authors of [59] present a non-intrusive approach which automatically analyses the execution logs of a load test to identify performance problems. As this approach only relies on load testing results, it cannot determine root causes. A similar approach is presented in [60] 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 proposed framework

enables the effective utilisation of performance diagnosis tools in the performance testing domain through automation, so that expediting the process of identifying performance issues and their root causes. Moreover the techniques presented in [59, 60] require information from previous runs to baseline their analysis, information which might not always be available.

4. PHOEBE: AN AUTOMATION FRAMEWORK FOR PERFORMANCE TESTING

In this section, we describe PHOEBE. First, we provide the context of our solution. Next, we describe the internal workings of PHOEBE. Finally, we conclude the section with a discussion of the proposed policies.

4.1. PHOEBE Overview

The objective of our research work was to develop a framework (PHOEBE) to automate the processes involved in the usage of a performance diagnosis tool during the performance testing of a clustered application. Such framework would improve the testers' productivity, as well as the performance testing process, by decreasing the effort and expertise needed to use the diagnosis tool.

A self-adaptive system is normally composed of a managed system and an autonomic manager [61]. In this context, PHOEBE plays the role of the autonomic manager. Therefore, it controls the feedback loop which adapts the managed system according to a set of goals. Meanwhile, the diagnosis tool and the application nodes play the role of the managed systems. This is depicted in Figure 3, which shows the conceptual view of PHOEBE.

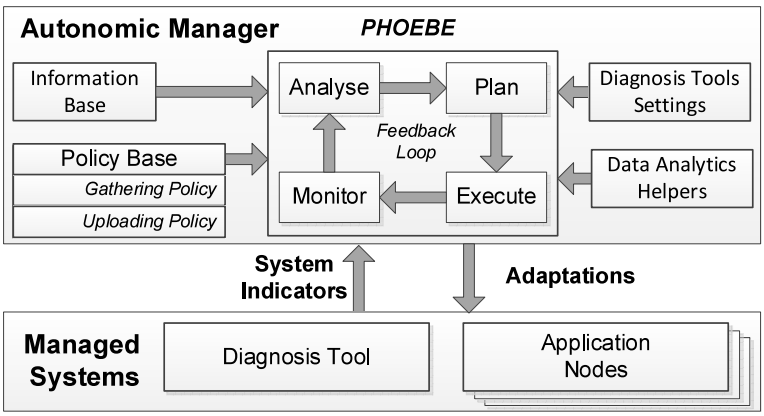

Figure 3. PHOEBE - Conceptual View.

As defined by multiple authors [62, 63], self-adaptation endows a system to adapt itself autonomously to internal and external changes to achieve particular quality goals in the face of uncertainty. In the context of PHOEBE, it means balancing the different trade-offs that exist when using a diagnosis tool (e.g., the amount of overhead introduced in the application nodes with respect to the bug accuracy achieved by the tool, as both factors depend on the frequency of the sampling and the amount of sampled data). To incorporate self-adaptation to PHOEBE, We have followed the well-known MAPE-K adaptive model [64]. This model is composed of 5 elements (depicted in Figure 3): A *Monitoring* element to obtain information from the managed systems; an *Analysis* element to evaluate if any adaptation is required; an element to *Plan* the adaptation; an element to *Execute* it; and a *Knowledge* element to support the others in their respective tasks.

The key aspect of PHOEBE is its *policy base*, which fulfils the role of the *Knowledge* element (within the MAPE-K model), and defines the pool of available policies. The encapsulation of the knowledge into policies allows PHOEBE to be easily extensible and capable of incorporating multiple policies, which might be suitable to different scenarios and diagnosis tools. In this context, a policy defines the set of interrelated tasks needed to perform one of the processes involved on the usage of a diagnosis tool within a performance test run. Each diagnosis tool requires three policies:

A data gathering policy (to control the collection of samples in the application nodes), an upload policy (to control when the samples are sent to the diagnosis tool for processing), and a consolidation policy (to integrate all the results obtained from the diagnosis tool for the different application nodes). Additionally, a diagnosis tool might have other policies available (e.g., to back up the obtained samples, or to trigger actions based on the tool's outputs). These policies can also make use of the available set of *data analytics helpers* (supporting logic which provides miscellaneous services to further customise the behaviour of a policy). For instance, a policy might focus on assessing the severity of the bugs identified by the tool. In this example, several helpers can be defined in order to offer different sets of severity levels for categorising the bugs (as the appropriate severity levels might vary depending on the usage scenario). In case a tool requires any particular settings to work properly (e.g., its range of applicable sample intervals), this information can also be captured by the framework (as a *diagnosis tool setting*).

From a configuration perspective, the tester needs to provide the *information base* (as shown in Figure 3), which is composed of all the input parameters required by the chosen policies. For example, an upload policy might require a *time interval* to know when to send the samples for processing, or a data gathering policy could use a *sampling interval* to know the frequency for the collection of samples. Likewise, a consolidation policy might require as input the *topology* of the clustered system, so that it can differentiate the application nodes which compose the cluster.

4.2. Core Process

From a process perspective, PHOEBE has a core process which coordinates the MAPE-K elements. The process is depicted in Figure 4. It is triggered when the performance test run starts. As an initial step, it gets a new *control test id*, a value that will uniquely identify the test run and its collected data. This value is propagated to all the nodes. Next all application nodes start (in parallel) the loop specified in the monitor and analyse phases, until the test run finishes: A new set of data samples is collected following a data gathering policy. After the collection finishes, the analyser process

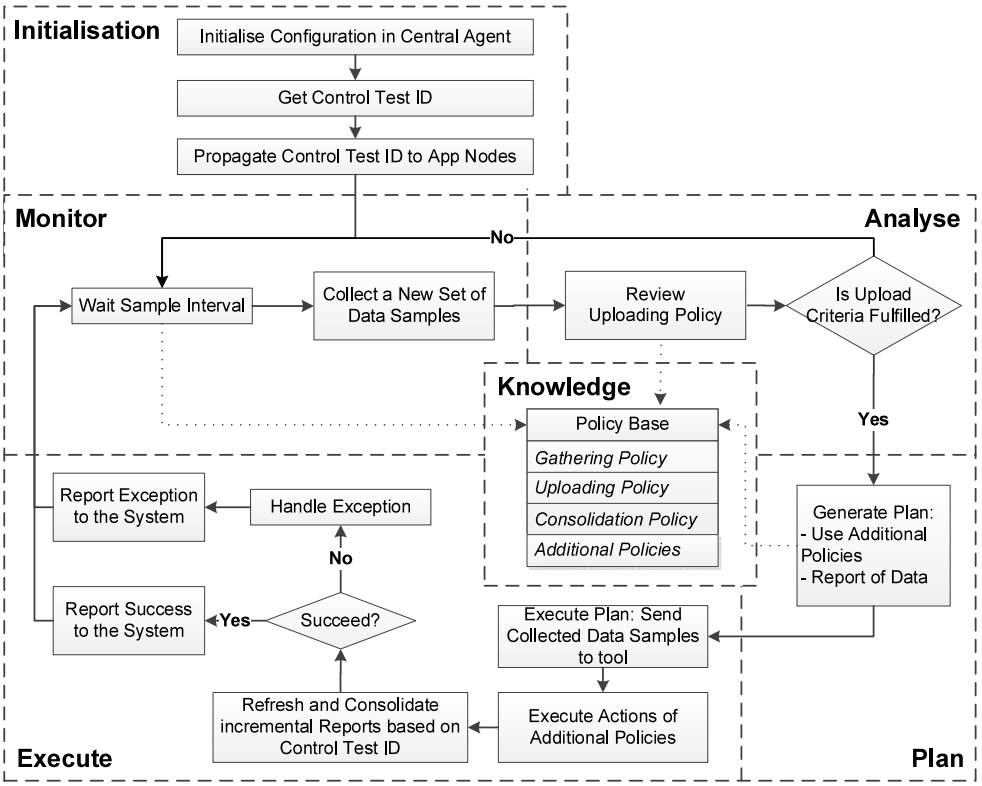

Figure 4. PHOEBE - Core Process.

checks the upload policies. If any upload policy has been fulfilled, the data is sent to the diagnosis tool (labelling 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 diagnosis tool to be consolidated. Additional policies might also be executed depending on the user's input configuration. For example, as certain data collections can be costly (e.g., the generation of a single 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 run finishes (or an alternative exit condition defined by a policy is fulfilled). When that occurs, all applicable policies are evaluated one final time before the process ends. Furthermore, any exceptions are internally handled and reported.

4.3. Architecture

PHOEBE is implemented with the multi-agent architecture depicted in Figure 1. There it can be seen how PHOEBE 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 policies and propagating the decisions to the other nodes. Meanwhile, the *application node agent* is responsible of performing the required tasks in each application node (e.g., sampling collection or sending the collected samples to the diagnosis tool). Finally, the *diagnosis tool agent* is responsible of interfacing with the diagnosis tool (e.g., feeding it or post-processing its generated reports).

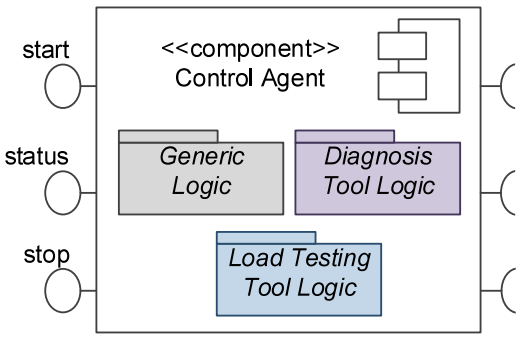

Figure 5. Control Agent - Component Diagram.

Internally, each agent is comprised of different components. This is exemplified in Figure 5, which presents the component diagram [65] of the control agent. There, it can be noticed how the agent has three main components: The generic component contains the control logic and all supporting functionality which is independent of the target diagnosis and load testing tools (e.g., the analysis and planning tasks of the policies). Regarding the logic that interfaces with the target tools, it needs to be customised per tool. Therefore, this logic is encapsulated in their respective components to minimise the required code changes. To complement this design strategy, the components are only accessed through interfaces. This is exemplified in Figure 6, which presents the high-level structure of the diagnosis tool component. It contains a main interface *IDiagnosisTool* to expose all required actions and an abstract class for all the common functionality. This hierarchy can then be extended to support specific diagnosis tools (e.g., WAIT) on different operating systems. Internally, the required class (supporting a particular tool, and possible a specific operating system) is automatically selected. This is achieved by following a *Factory* design pattern [66].

Finally, the agents communicate through commands, following the *Command* design pattern [67]: 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 7: Once a tester has started a performance test run (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 diagnosis tool for processing (steps 10 and 11). These steps continue iteratively until the test ends.

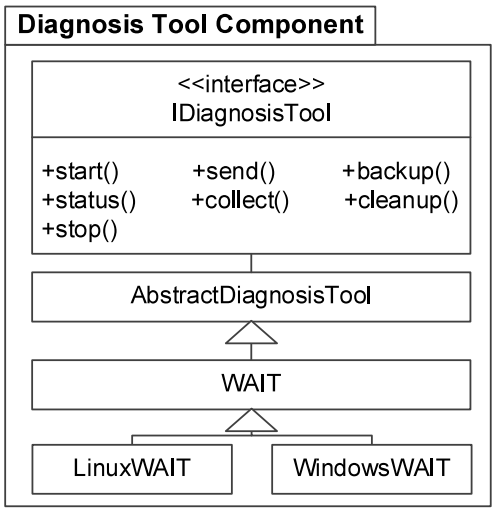

Figure 6. Diagnosis Tool Component - Class Diagram.

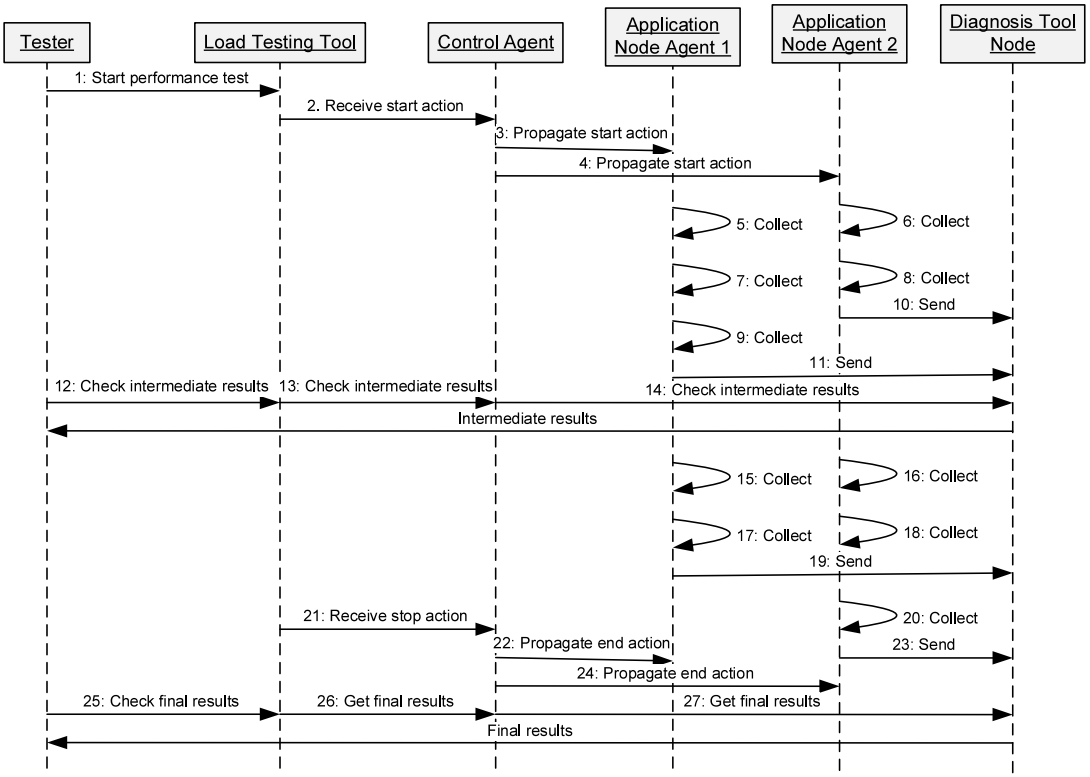

Figure 7. PHOEBE - Sequence diagram.

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 diagnosis tool (steps 12 to 14) until getting the final results (steps 25 to 27).

4.4. Adaptive and Data Analytics Policies

The following sections describe the set of policies that have been developed to work with PHOEBE. They are mainly based on the outcomes of the performed assessments of trade-offs (discussed in Sections 5.1, 5.2 and 5.3).

4.4.1. Accuracy-Target Data Gathering Policy. This adaptive policy was designed to balance the trade-off between the accuracy in the results of a diagnosis tool and the performance overhead introduced into the tested application. This is because both factors are influenced by the selection of the sample interval (SI).

The policy process is depicted in Figure 8, 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 (expressed as a percentage); and the warm-up period. Resembling its usage in performance testing [68], the warm-up period is the time after which all transactions have been executed at least once (hence contributing to the average response time of the test run). Two additional parameters are retrieved from the knowledge base, as their values are specific for each diagnosis tool: The minimum SI that should be used for 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 application nodes with the minimum SI. This strategy allows collecting as many samples as possible, unless the performance degrades below the desired threshold, thus violating the acceptable service level agreement (SLA). 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 diagnosis tool 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.

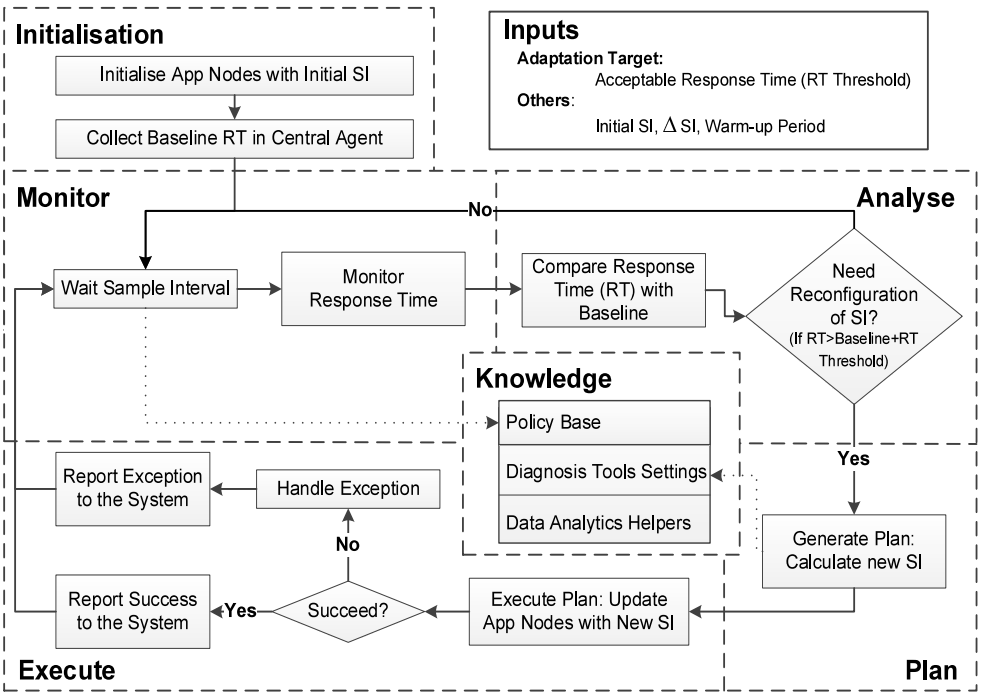

Figure 8. Accuracy-Target Data Gathering Policy.

Finally, it is worth mentioning that this policy was inspired by the scenario (commonly experienced in the industry) where a test team aims to minimise the number of required performance test runs 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 run.

4.4.2. Efficiency-Target Upload Policy. This adaptive policy was designed to balance the trade-off between the resource utilisation levels required by a diagnosis tool and the amount of sampled data which is concurrently processed by the tool. This is because both factors are influenced by the selection of the upload interval (UI).

The policy process is depicted in Figure 9, where each step is mapped to the corresponding MAPE-K element. It requires two user inputs: The initial UI to be used; and the ΔUI, which indicates how much the UI should change when an adjustment is required. An additional parameter would be retrieved from the knowledge base (as its value is specific to each type of resource): The target of maximum utilisation (U_{MAX}). As documented in [69], the objective of this target is to retain certain unused capacity to provide a soft assurance for quality of service. For instance, in the case of the CPU, this target is 90%.

The process starts by initialising the application nodes with the initial UI. Then the process waits until all the application nodes have uploaded their samples once. This step is done to make sure that the UI is only modified if required. After all the nodes have uploaded their results, the process retrieves the average resource utilisation (RES_{AVG}) of the shared service (e.g., a WAIT server) during the processing of those samples, as well as the average duration of the resource usage ($DRES_{AVG}$). Then the RES_{AVG} is compared with the U_{MAX}. If the U_{MAX} has been exceeded, new UIs are calculated. As a first strategy, a different UI 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 current UI, the calculation of the new UIs is based on the current UI (by iteratively subtracting or adding the $DRES_{AVG}$ from the current UI until all nodes have a different UI). For example, if we have 5

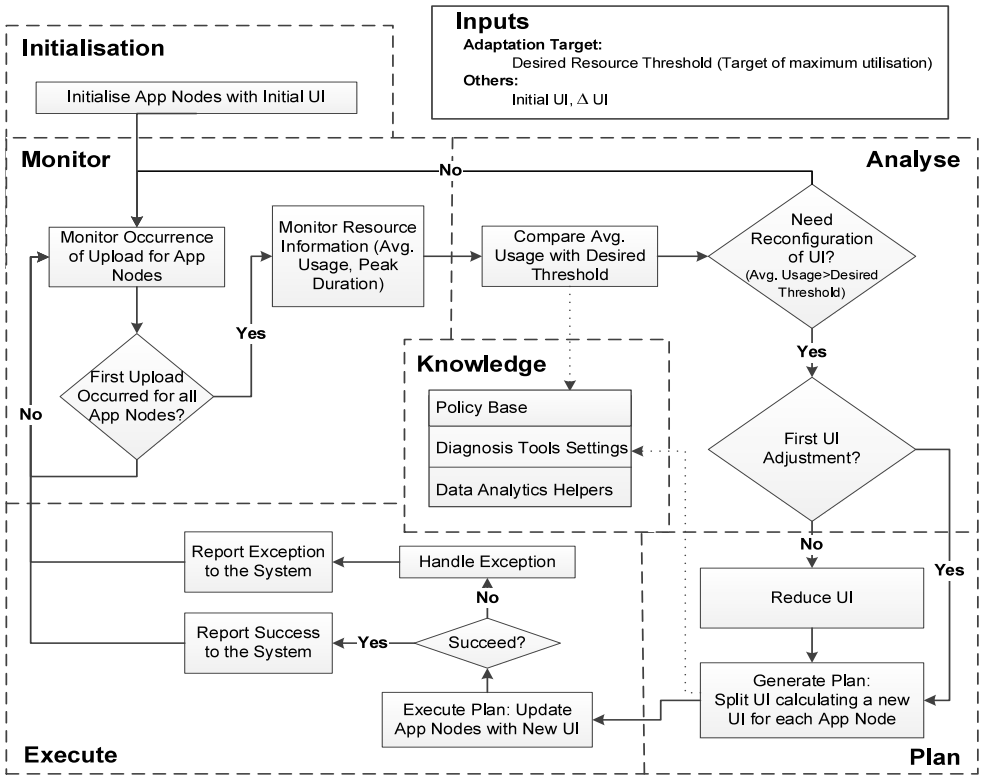

Figure 9. Efficiency-Target Upload Policy.

nodes, a current UI of 30 minutes and a $DRES_{AVG}$ of 1 minute, the new UIs of the nodes would be distributed as 28,29,30,31 and 32. Finally, the new UIs are propagated to their respective application nodes, which start using them since the next upload iteration. In case a subsequent adjustment is required (meaning that only splitting the UI was not enough to bring the RES_{AVG} below the U_{MAX}), the current UI is decreased (by the value configured as ΔUI), before the calculation of the new UIs is done. This is done to reduce the number of samples sent (per node) in each upload.

Finally, it is worth mentioning that this policy was inspired by the scenario (commonly experienced in the industry) where the number of available licenses for a particular tool is limited (e.g., due to budget constraints). In this scenario, the capability of efficiently sharing the available tool's instances across different teams and projects is highly desirable in order to maximise the return of investment [70] of the tool.

4.5. Multi-View Consolidation Policy

This policy was designed to minimise the effort and expertise required by a tester to analyse the results of using a diagnosis tool in a clustered application. This is done by providing four different views of the results: The main view is the *consolidated system-level* view. It allows a tester to easily track the progress of the performed diagnosis in the overall cluster and see if any relevant systemic-level issue has occurred. Three other complementary views are also available: The *individual system-level* allows to see the results obtained by an individual processing cycle (as controlled by the UI). Furthermore, the *consolidated node-level* and *individual node-level* views behave similar to their system-level counterparts, but focusing on a particular node. Together, this set of views offer a tester different levels of granularity, within the diagnosis results, so that system-level (or node-level) performance issues can be easily identified.

The policy process is depicted in Figure 10. It requires two user inputs: The severity type, which defines the set of applicable severity categories under which the identified performance issues can be classified; and the severity thresholds, which delimit the severity ranges of each category. An additional optional user input is the *Go No-Go Assessment*, which defines an alternative exit condition (other than the completion of the test run).

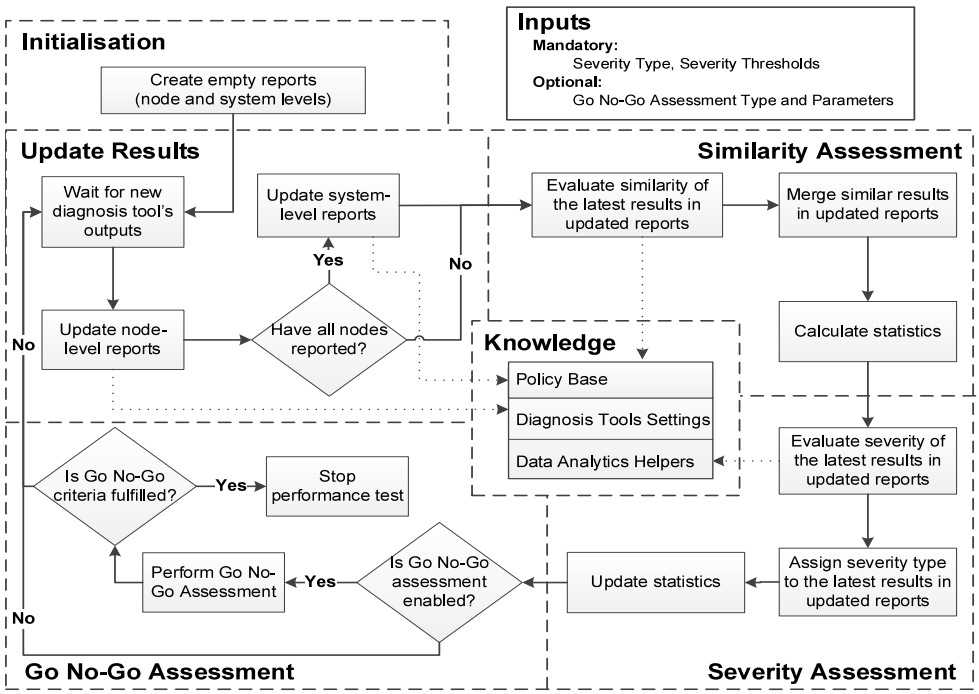

Figure 10. Multi-View Consolidation Policy.

The process starts by generating a set of empty reports. Next, the process waits until a new set of outputs is generated by the diagnosis tool. Once this occurs, the corresponding node-level report is updated. This step involves parsing all the received outputs (as a single upload process might generate multiple output reports) in order to extract the relevant qualitative information (i.e. the identified issues' information, including their categories and subcategories). Internally, the parsing relies on a set of rules defined in the knowledge base (as they vary among diagnosis tools). Once extracted, the new issues are incorporated into the node-level reports: A new individual-level report is generated by consolidating all the tools' outputs. Likewise, the consolidated-level report of the node is updated with the new issues. The newly added information is also tagged, so that the subsequent tasks (i.e. the similarity and severity assessments) can easily identify it. Furthermore, the updates per node are tracked. This is done so that, once results from all the nodes are obtained, the system-level reports are also updated (following a logic similar to the node-level reports previously discussed).

Once the reports have been updated, the similarity of the new results is evaluated. This is done to further consolidate the results (as the outputs of a diagnosis tool tend to overlap when monitoring multiple nodes during extended periods of time). The rules to evaluate the similarity are retrieved from the knowledge database (as they will depend on the diagnosis tool). A set of standard statistic metrics (i.e. average, standard deviation and coefficient of variation [71]) is also calculated to gradually build an historical trend per report type. Next, a severity assessment is performed. This is done with the aim of helping a tester to concentrate on the set of performance issues worth exploring (as a diagnosis tool might produce a considerable amount of outputs and normally only a subset is truly relevant). As an initial step, the severity of the new results is calculated, following the rules applicable for the diagnosis tool (e.g., based on the frequency of the identified issues). Next, the results are classified following the severity style and thresholds configured by the tester. Finally, standard statistics (similar to the ones previously discussed) are calculated per severity type. Once this is done, a new version of the consolidated reports is ready and accessible to testers.

An additional (and optional) step in the process is the evaluation of a *Go No-Go Assessment*. The objective of this step is to offer a performance test run an alternative exit criterion (other than the duration of the test run). In case this option is enabled by the tester, the corresponding Go No-Go assessment would be performed. In case its exit criteria is fulfilled, a stop action will be triggered. This action behaves similarly to the stop command described in Section 4.3.

Finally, it is worth mentioning that this policy was inspired by the scenario (commonly experienced in the industry) where a tester needs to monitor a clustered application (normally composed of multiple application nodes), during long periods of time (probably one or more days). Under these conditions, the amount of output data generated by a diagnosis tool can be vast, easily overloading a tester due to the considerable amount of effort and expertise required to consolidate and analyse the obtained results.

4.6. Data Analytics Helpers

The following sections describe the set of data analytics helpers developed to support the capabilities of PHOEBE. Similarly to the previously discussed policies, they are mainly based on the outcomes of the performed assessments of trade-offs (discussed in Sections 5.1, 5.2 and 5.3).

4.6.1. Similarity Assessments. As discussed in Section 4.5, a similarity assessment defines the logic that is used to identify those performance issues, reported by the diagnosis tool, which should be merged because their symptomatology suggests that they are instances of the same performance issue. This type of supporting logic is captured as a data analytics helper within PHOEBE.

Based on the results obtained from the assessment of trade-offs, as well as after analysing the outputs of the studied diagnosis tools, we have initially concentrated on implementing the following two similarity assessments:

- An *equality assessment*, applicable for those tools which exclusively generates qualitative issues' descriptions (e.g., IBM WAIT). In this case, the issues' descriptions can be directly compared.

- A *semantic similarity assessment*, applicable for those tools which generate qualitative issues' descriptions with some quantitative data embedded (e.g., IBM Health Center). In this case, it is preferable to compare the issues' descriptions in terms of their semantic similarity. For this work, the JaroWinkler distance [72] (originally developed in the field of record linkage to detect duplicate strings) was chosen. This is because this metric is widely-used in the literature and it also offers a normalised score (where 0 means no similarity and 1 means an exact match). As usually only the quantitative information of an issue's description changes (among instances of the same issue), a dissimilarity threshold of 0.1 was enough to identify similar issues.

4.6.2. Severity Styles. As discussed in Section 4.5, a severity style defines the set of severity categories in which a performance issue can be classified. This type of supporting logic is captured as a data analytics helper within PHOEBE. Furthermore, a severity style can be used within an assessment to customise its behaviour (e.g., the Go No-Go assessment discussed in Section 4.6.3).

Among the alternative strategies to develop severity styles for PHOEBE, we have initially concentrated on implementing the following two:

- A *five-level severity style* based on the well-known (and commonly used in the industry) severity categories suggested by the International Software Testing Qualifications Board (ISTQB) [73]: Critical, major, moderate, minor, and cosmetic [74].

- A *two-level severity style* which classifies the issues in two categories: Critical and non-critical. This simplified style, which can be easily mapped to the ISTQB one, allows testers to concentrate only on the most relevant issues (i.e. those within the critical category).

Finally, the classification of a performance issue within a severity category is based on the frequency of the issue (as defined per diagnosis tool) with respect to the thresholds configured for the applicable severity categories. This classification is done irrespectively of the chosen severity style.

4.6.3. Go No-Go Assessments. As discussed in Section 4.5, a Go No-Go assessment offers an alternative exit condition to a performance test run (other than waiting for the completion of its planned execution time). This type of supporting logic, captured as a data analytics helper within PHOEBE, involves the definition of the evaluation criteria responsible of assessing if the exit condition has been fulfilled.

Based on the results obtained from the assessment of trade-offs (where it was observed that the number of issues identified by a diagnosis tool tend to stabilise through time), a Go No-Go assessment based on the Coefficient of Variation (CV) [71] was developed. The CV was selected because this statistic metric allows to dimensionlessly measure the variability in the number of identified issues. For instance, if a CV of 0.1 is obtained across a set of different report versions, this value indicates a practically stable number of identified issues among the reports (regardless of the actual number of bugs or the number of compared reports). Therefore, this metric can capture when the process of identifying bugs has exhausted producing new results. Once that point has been reached, a test run can be stopped to prevent wasting valuable human and computational resources.

This Go No-Go assessment requires three user inputs:

- The severity categories to assess, which defines the subset of severity categories (within in the chosen severity style) that will be evaluated (as not all the severity categories might be of interest - e.g., the non-critical or cosmetic issues -).

- The number of consolidated system-level reports to assess, which will delimit the number of versions of this report (starting from the most recent one) that will be evaluated. Indirectly, this parameter also influences the minimum duration of the test (as there must be enough historical data available before the first assessment can be performed).

- The set of CV thresholds (one for each severity category to assess) that will be used in the evaluation to determine if the calculated CV falls within an acceptable range for a particular severity category.

From a process perspective, the assessment starts by checking if there is enough historical information (in the form of consolidated system-level reports) to calculate the required CV values. If it is not the case, the assessment fails. Otherwise, the CV is calculated for each severity category. Then, the obtained values are compared against their corresponding thresholds. Only if the CV value is lower (or equal to) the corresponding threshold for all the severity categories, the assessment is considered fulfilled.

5. EXPERIMENTAL EVALUATION

This section presents the experiments performed to evaluate PHOEBE. To understand which policies would work best, we started by performing an assessment of the identified trade-offs. It involved three experiments: Firstly, we evaluated the accuracy of each diagnosis tool with respect to the overhead introduced by the data sampling processes that feed the tool. Secondly, we evaluated the effort required by a tester to analyse the outputs of each diagnosis tool with respect to the amount of outputs generated by the tool. Thirdly, we evaluated the resources required by each diagnosis tool with respect to the amount of samples processed by the tool. After developing the proposed policies (described in Section 4.4), two additional experiments were done to evaluate the benefits and costs of using PHOEBE: First, we assessed the accuracy of the implemented policies (i.e., how well they addressed the previously identified trade-offs). Then, we assessed the productivity gains that PHOEBE brings to the performance testing process. The section concludes with a discussion for practitioners where we summarise the key findings and observations.

5.1. Experiment #1: Accuracy Trade-off Assessment

Here, the objective was to evaluate the potential trade-off between the accuracy of the results generated by a diagnosis tool and the overhead introduced in the application nodes by the data sampling processes that feed the tool. The following sections describe this experiment and its results.

5.1.1. Experimental Set-up In the following paragraphs we present the developed prototype, the test environment and the parameters that defined the evaluated experimental configurations: The selected diagnosis tools, Java benchmarks, sample intervals (SI) and upload intervals (UI). We also describe the evaluation criteria used in this experiment.

Prototype. A prototype has been developed in conjunction with our industrial partner. The *Control Agent* was implemented on top of the Apache JMeter 2.9 [14], which is a leading open source tool used for application performance testing. Meanwhile, the *Application Node Agent* and the *Diagnosis Tool Agent* were implemented as stand-alone Java applications. Internally, each agent has an embedded Jetty Web Servlet Container [75] (a popular open source solution used for enabling machine to machine communications). This allows the different agents to communicate through HTTP requests.

Furthermore, two initial policies were implemented: A data gathering policy which uses a constant SI during the complete test execution; and an upload policy which uses a constant UI. As the SI controls the frequency of samples collection from the monitored application (which is the main potential cause of overhead introduced by a diagnosis tool), a broad range of values was tested (0.125, 0.25, 0.5, 1, 2, 4, 8 and 16 minutes). The smallest value in the range (0.125 minutes) was intentionally chosen to be smaller than the minimum recommended value for the chosen diagnosis tools (0.5 minutes). Similarly, the largest value in the range (16 minutes) was chosen to be larger than 8 minutes (a SI commonly used in the industry). Finally, as the UI is not involved in the data gathering process, a constant value of 30 minutes was used.

Environment. All the experiments were performed in an isolated test environment, so that the entire load was controlled. This environment was composed of eight VMs: A cluster of five

application nodes with one load balancer, one diagnosis tool server, and one load tester node (as shown in Figure 11). All the VMs had the following characteristics: 4 virtual CPUs at 2.20GHz, 3GB of RAM, and 50GB of HD; running Linux Ubuntu 12.04L 64-bit, and OpenJDK JVM 7u25-2.3.10 with a 1.6GB heap. The load tester node also used an Apache JMeter 2.9 [14] (a leading open source tool used for application performance testing), and the application nodes ran an Apache Tomcat 6.0.35 [76] (a popular open source Web Application Server for Java).

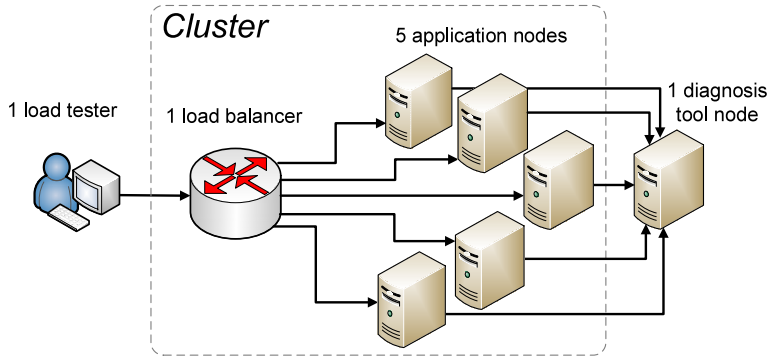

Figure 11. PHOEBE - Test Environment.

The VMs were located on a Dell PowerEdge T420 server [77] equipped with 2 Intel Xeon CPUs at 2.20Ghz (12 cores/24 threads), running Linux Ubuntu 12.04L 64-bit, 96 GB of RAM, 2TB of HD, and using KVM [78] for virtualisation.

Diagnosis Tools. The five performance diagnosis tools discussed in Section 2.2 were used: Eclipse Memory Analyser (EMAT), IBM Garbage Collection Lite (GCLITE), IBM Garbage Collection and Memory Visualiser (GCMV), IBM Health Center (HC), and IBM Whole Analysis Idle Time (WAIT). This decision was taken to diversify more the evaluated behaviours.

Benchmarks. One of the Java benchmarks most widely-used in the literature (DaCapo 9.12 [79]) was chosen because it offers a wide range of different program behaviours to test. Unlike other benchmarks (which are synthetically generated), these are real-life programs from different business domains and which are widely-used in the industry. Appendix A presents a summary of this benchmark and its respective set of programs.

In order to invoke the Dacapo programs from within a test script, a wrapper JSP was developed and installed in the Tomcat instance of each application node. It allowed the execution of any DaCapo program via an input parameter. Next, a JMeter test script was created to iteratively execute all DaCapo programs. For each program, its biggest workload size (among the available pre-defined workload sizes of DaCapo [80]) was used. Each individual program call was considered a transaction. Finally, a 24-hour test duration was chosen to reflect more realistic test conditions.

Participant Testers. All held bachelor degrees in computer science and had a professional experience (in software development and testing) above 10 years. Following the experience threshold used by other works [81], the participants were considered experienced testers. Furthermore, they had previous hands-on experience using the involved diagnosis tools (hence, there was no learning curve that could have impacted the results). Likewise, the usage of PHOEBE was explained before starting the experiment.

Evaluation Criteria. In terms of performance, the main metrics were throughput per second (tps) and response time (ms). Concerning response time, lower values are better; while for throughput, higher values are better. These metrics were collected with JMeter. In terms of testing productivity, the main metrics were the number of bugs found and the number of critical bugs found. In both cases, higher values are better. These metrics were obtained from the reports generated by the used diagnosis tools.

5.1.2. Experimental Results. In this section, we discuss the main results obtained from this experiment in terms of the relevant performance and testing metrics.

For all the sample-based tools (i.e. WAIT, HC, EMAT), the obtained results showed that there is a clear relationship between the selection of the SI and the performance cost of using the tools. This behaviour is depicted in Figures 12, 13 and 14. which summarise the results of the tested configurations per tool. There, it can be noticed how the throughput decreases when the SI decreases. These performance impacts are mainly caused by the involved sample generation processes. For instance, in the case of WAIT (depicted in Figure 12), the generation of a javacore (which is the main input used by WAIT, as explained in Section 2.2) involves temporarily pausing the execution of the application processes running within the JVM [82]. Even though the cost was minimum when using higher SIs, it gradually became visible (especially when using SIs below 0.5 minutes). On the contrary, the number of identified bugs increases when the SI decreases. This positive impact is a direct consequence of feeding more samples to the diagnosis tool, which is pushed to do a more detailed analysis of the monitored application. It is worth noticing that the biggest performance impacts were experienced by EMAT. This is because it requires the generation of heapdumps, process which is known to be time-consuming and which can have a severe performance impact on the monitored application [83].

For the trace-based tools (i.e. GCLITE and GCMV), the obtained results showed that there is no relationship between the selection of the SI and the performance cost of using these tools. This behaviour is depicted in Figure 15, which presents the results of GCLITE. There, it can be noticed how the differences in performance were minimal, and relatively constant and independent of the used SI. This is because the amount of generated GC verbose (which is the main input used by GCLITE, as explained in Section 2.2) only depends on the executed application functionality.

A second round of analysis was performed concentrating on the most critical issues identified by the sample-based tools. The objective was to assess if the previously described behaviours (with

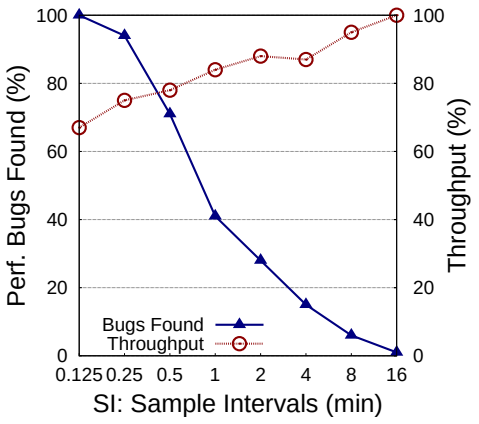

Figure 12. Perf. Bugs vs. Throughput - WAIT.

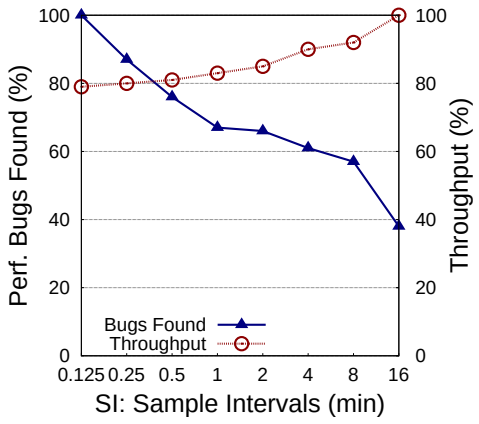

Figure 13. Perf. Bugs vs. Throughput - HC.

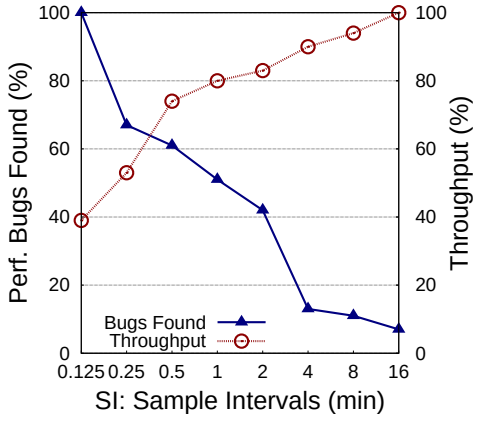

Figure 14. Perf. Bugs vs. Throughput - EMAT.

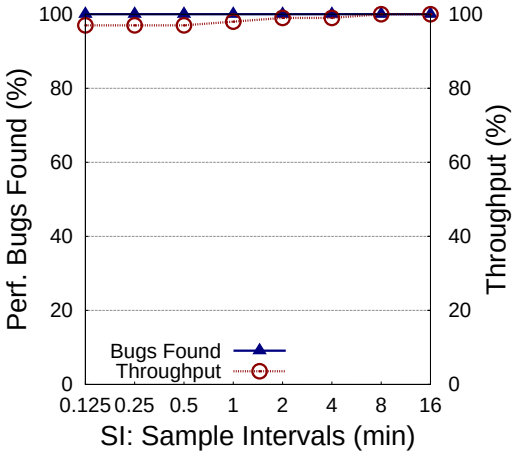

Figure 15. Perf. Bugs vs. Throughput - GCLITE.

respect to the existing trade-off between the selection of SI and the accuracy of the tools' results) were also observed there. As shown in Figures 16, 17, and 18, similar behaviours were observed, confirming the relevance of the trade-off.

An additional observation of this experiment was that the number of identified non-critical bugs was considerable higher than the number of critical bugs. This was because the diagnosis tools

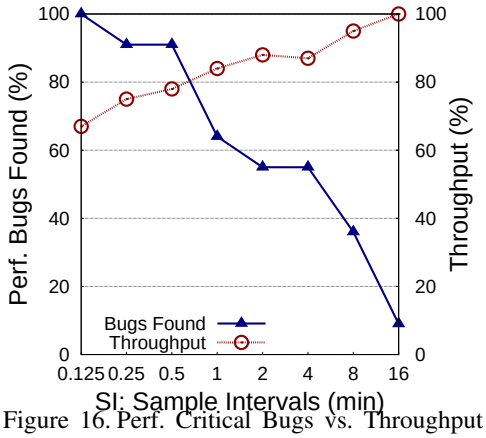

Figure 16. Perf. Critical Bugs vs. Throughput - WAIT.

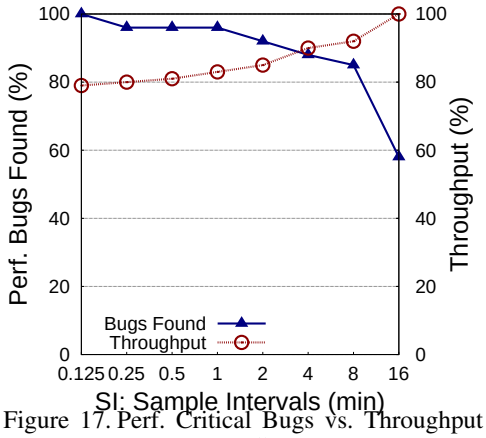

Figure 17. Perf. Critical Bugs vs. Throughput - HC.

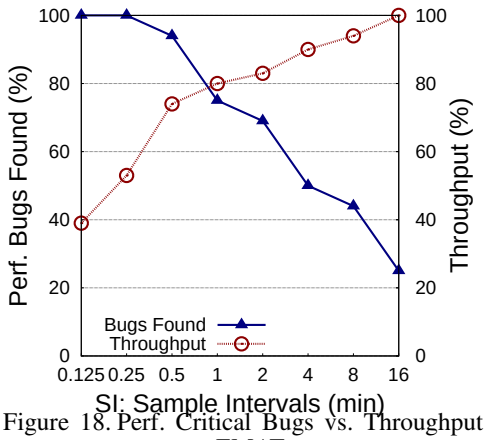

Figure 18. Perf. Critical Bugs vs. Throughput - EMAT.

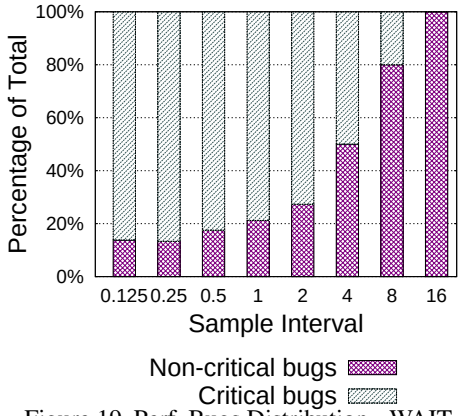

Figure 19. Perf. Bugs Distribution - WAIT.

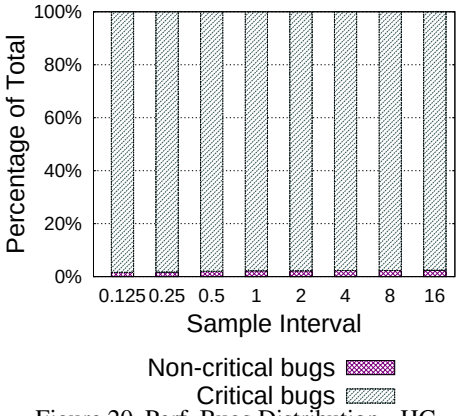

Figure 20. Perf. Bugs Distribution - HC.

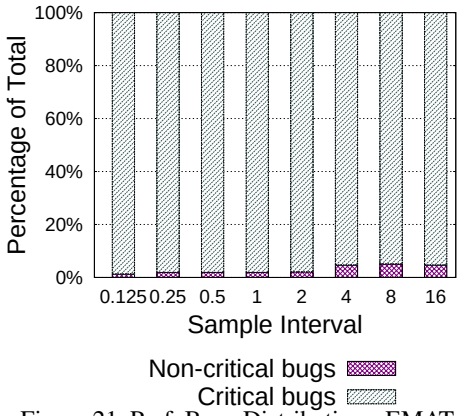

Figure 21. Perf. Bugs Distribution - EMAT.

tended to report potential performance issues even when their frequency was very low. This scenario is more likely to occur when using a small SI (e.g., 0.5 minutes or less) or a diagnosis tool that analyses the samples individually (e.g., HC and EMAT). For instance, most of the non-critical bugs reported by WAIT had a frequency below 1%, meaning that it is very likely that the suspected errors were only normal logic being processed. This behaviour is visually depicted in Figures 19, 20, and 21, which show the bug distributions for WAIT, HC and EMAT (respectively). Likewise, the number of new identified bugs decreased during the execution of a test run. This was because most of the bugs identified in late phases of a test run were merely instances of previously identified critical bugs. This behaviour suggested that the test run had already exhausted its benefits (in terms of found bugs) at some moment during its execution.

Summary. The results showed how the selection of the SI influences the performance overhead that sample-based tools introduce on the monitored application. This made that the automatic selection of the SI parameter was identified as an appropriate policy within PHOEBE. For trace-based diagnosis tools (which are insensitive to the selection of the SI), a constant SI can be more suitable. It was also observed that the number of non-critical bugs tends to be considerable larger than the critical ones, and the number of new identified bugs tends to decrease during the execution of a test run.

5.2. Experiment #2: Effort Trade-off Assessment

Here the objective was to evaluate the effort required to analyse the outputs of the chosen diagnosis tools, as well as understand the reasons behind it. The aim was to estimate the potential effort gains that PHOEBE can achieve. The following sections describe this experiment and its results.

5.2.1. Experimental Set-up. This assessment re-used some of the outputs produced by experiment #1 (discussed in Section 5.1). Primarily, the reports generated by the diagnosis tools and the effort invested by the testers in analysing the reports. Due to the huge number of reports (above 24000 per tool/SI combination) generated by using the diagnosis tools with a small SI (i.e., 0.125 or 0.25 minutes), it was not feasible to do the manual analysis for those experimental configurations (due to the limited availability of the testers). In those cases, the efforts were estimated. The average efforts (per report) obtained from the other experimental configurations (i.e. those using a SI above 0.5 minutes) were used to extrapolate those efforts. The exception was WAIT, as this tool did not produce a huge number of reports with any SI.

5.2.2. Experimental Results. In this experiment, the analysis focused on evaluating the effort invested in analysing the outputs of the diagnosis tools. The aim was to define a baseline to which PHOEBE can be compared against.

Analysis effort. From a qualitative perspective, this experiment allowed us to take a closer look to the process normally followed by a tester in order to analyse the outputs of a diagnosis tool: After collecting and processing the samples (tasks done by PHOEBE through the constant SI/UI policies discussed in Section 5.1.1), the testers iteratively reviewed the results of the new reports, then identified/merged any bugs which were instances of previously identified bugs, assessed their severity, and looked for any relevant bug trends. These learned lessons were reflected in the consolidation policy described in Section 4.5.

From a quantitative perspective, the overall results showed how the analysis of the reports generated by a diagnosis tool is normally a time-consuming process and there are significant potential gains (in terms of effort-savings) to address. This is shown in Figure 22, which depicts the obtained results for the evaluated diagnosis tools per SI. There is can be noticed how the analysis effort tends to considerably increase when using smaller SIs (e.g., 0.125). This is a reflection of the behaviour of the tools: All diagnosis tools (except WAIT) generates one report per processed sample. Therefore, the number of reports is directly related to the chosen SI. On the contrary, WAIT generates one report per processing cycle (i.e. UI), regardless of the sampled data. It is worth noticing that, even in this relatively controlled/steady scenario (where WAIT created 2 reports per node/hour), the effort required was not negligible (around 20 hours).

As discussed in the results obtained in experiment #1, trace-based diagnosis tools (e.g., GCLITE and GCMA) do not benefit (from a bug finding efficiency) from using a small SI (e.g., 0.125 minute). Similarity, from an effort perspective, it is better to use a big SI (e.g., 8 minutes) because that decreases the number of reports to analyse without losing any precision (in terms of identified bugs). This behaviour can be noticed in Figure 22.

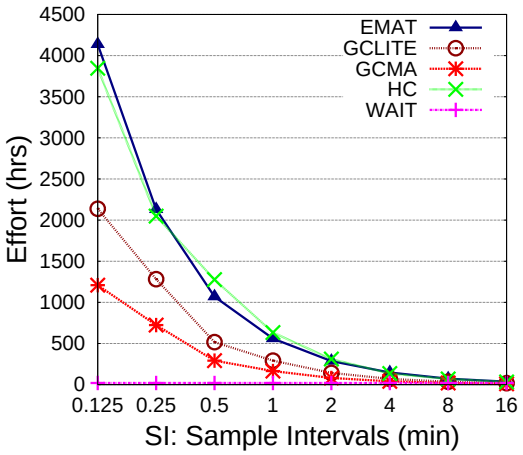

Figure 22. Performance Analysis Efforts per Tool.

To further contextualise the previously discussed results, it is worth remarking two aspects:

- The reported efforts did not consider the gathering and processing of the sampled data. This is because these tasks were automatically performed by PHOEBE. Therefore, bigger potential gains can be expected when comparing PHOEBE against a completely manual usage of a diagnosis tool. This scenario is explored in experiment #5 (described in Section 5.5).

- All the test runs lasted their planned duration (i.e. 24 hours). Therefore, bigger time-savings can be expected if the duration of the test can be decreased. As suggested by the results of experiment #1 (described in Section 5.1.2), a test run might have exhausted its benefits (in terms of found bugs) before completing its duration. However, a tester cannot know for sure until finishing the analysis of the obtained results. On the contrary, if an alternative stop condition can be triggered whenever this situation occurs (such as the one discussed in Section 4.6.3), the test might be able to finish before its planned duration, saving additional time and resources (e.g. shared test environments).

Effort-driven factors. The next round of analysis focused on understanding the main factors that driven the amount of effort required in the analysis of the outputs generated by the diagnosis tools. As an initial step, we analysed the differences in complexity among the reports generated by the tools. This qualitative analysis showed that the reports generated by EMAT, HC and WAIT were of similar complexity (due to their structure and the amount of presented information), taking a similar amount of time to review. On the contrary, the reports generated by GCLITE and GCMA took considerably less time due to the relatively narrow scope of the tools (i.e. GC issues).

Even though the observed differences in effort might be subjective to a tester' expertise, the main observation from this analysis was that the complexity of the tool was a minor factor with respect to the overall effort required by the analysis. This is because the main factor influencing the amount of effort required to do the analysis was the number of generated reports. Similarly, it was driven by multiple factors: Firstly, the behaviour of the tool (whether it creates a report per sample -i.e. SI- or per processing cycle -i.e. UI-). Additionally, the number of reports is directly related to the number of application nodes and the duration of the performance test. That is, the larger the number of application nodes (or the longer the test), the larger the number of reports that are generated.

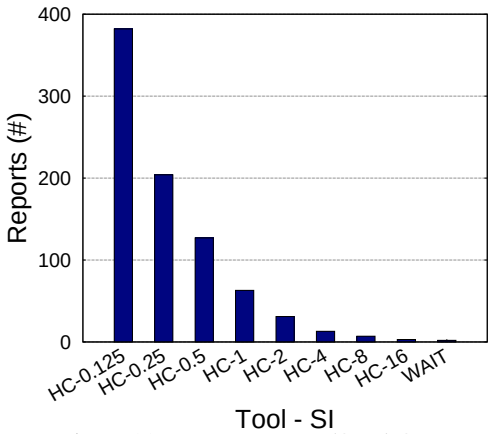

Figure 23. Reports per Tool/Node/Hour.

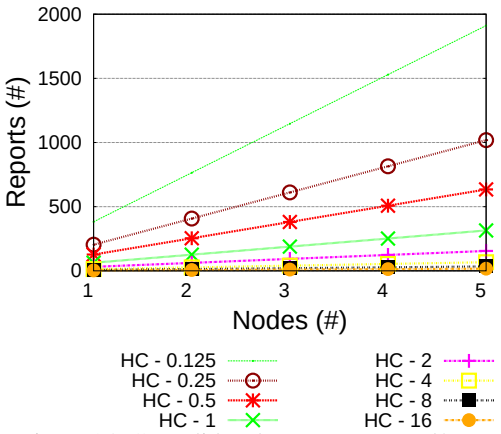

Figure 24. Consolidated Reports per Tool/Hour.

This scenario is exemplified in Figures 23 and 24. There, the tool which required the least effort (WAIT) is compared against one of the tools (HC) which required the most effort (as shown in Figure 22). Figure 23 shows the number of reports that are generated by each tool per node/hour (based on the used SI). It can be noticed how the number of reports generated by HC drastically increased when using smaller SIs (e.g., 0.125). On the contrary, WAIT only appears once in the figure because it is insensitive to the SI selection (hence it always generated 2 reports, per node/hour,

when using a 30-minute UI). Meanwhile, Figure 24 shows how the number of reports monotonically increased with respect to the number of monitored nodes. Similar trends were observed in terms of execution time, as the number of reports increased practically linear to the duration of the performance test run (i.e., the 24-hour duration used on this experiment).

Summary. In conclusion, the results of this experiment showed how the potential effort-savings that PHOEBE can address are significant (especially considering the experience of the involved testers and the relative modest size of the cluster - 5 nodes -). This is because the analysis of the reports generated by a diagnosis tool is a time-consuming process. Even though the complexities of the reports generated by each tool might differ, the effort involved in the analysis is mainly driven by the number of reports. As the number of reports is directly related to the duration of the test and the number of application nodes in the monitored environment, the potential gains will be considerable high 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 potential time savings will be higher under those conditions.

5.3. Experiment #3: Resource Trade-off Assessment

Here the objective was to evaluate the potential trade-off between the number of samples concurrently processed by a diagnosis tool and the amount of resources it requires to process the samples. The following sections describe this experiment and its results.

5.3.1. Experimental Set-up. The set-up was similar to that used in the experiment #1 (presented in Section 5.1.1), with two differences: First, as both SI and UI influence the number of samples that are sent to the diagnosis tool for processing, a range was chosen for each parameter. For the SI, the following 3 values were used: 0.5, 4 and 8 minutes. For the UI, the following 3 values were used: 5, 30 and 60 minutes. Second, the main metrics were the CPU (%) and memory (%) utilisations in the diagnosis tool node, during the processing of the samples. These metrics were collected using the "top" command.

5.3.2. Experimental Results. Overall, the results showed how the resource utilisation in the diagnosis tool node is related to the number of samples processed in parallel; which is a function of both the SI and UI. For example, the experimental configurations (per diagnosis tool) which used a SI of 4 minutes and an UI of 30 minutes, reported similar resource utilisations than the configuration which used a SI of 8 minutes and an UI of 60 minutes. This is because both combinations fed the same number of samples (per upload iteration) to the diagnosis tool.

Even though the CPU and memory utilisations showed similar trends (in the sense that they tended to grow with respect to the processed samples), each diagnosis tool experienced different CPU/memory behaviours: For instance, WAIT proved to be considerably more CPU-intensive (with its CPU_{AVG} exceeding the 90% utilisation in most of the tested configurations). On the contrary, WAIT was considerable less memory-intensive (with its highest MEM_{AVG} below 5% utilisation in all the tested configurations). The obtained results are presented in Figures 25 (CPU) and 26 (memory). There, it can be noticed how WAIT was the most CPU-intensive tool, followed by GCLITE and EMAT. Meanwhile, HC was (by far) the most memory-intensive. For instance, it was the only tool for which the MEM_{AVG} of one experimental configuration exceeded the 90% utilisation. Finally, it is worth remarking how several experimental configurations exceeded the 90% of CPU and/or memory utilisations (utilisation target frequently suggested in order to retain some unused capacity and provide a soft assurance for quality of service [69]), despite the relative modest size of the cluster (5 nodes).

Summary. The performed tests demonstrated how the selection of UI influences the resource utilisation in the diagnosis tool. Even though each diagnosis tool experienced different levels of CPU/memory-intensivities, the obtained results made that the automatic selection the UI parameter was identified as another appropriate policy within PHOEBE.

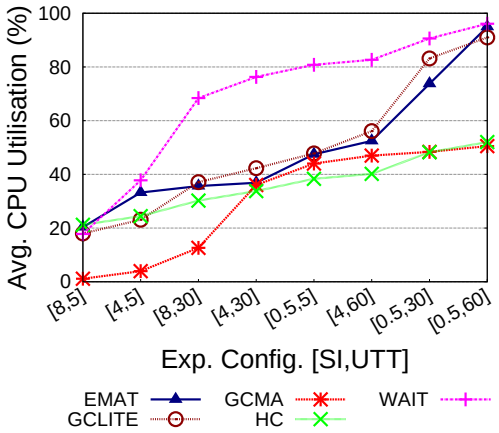

Figure 25. Avg. CPU Utilisations per Tool.

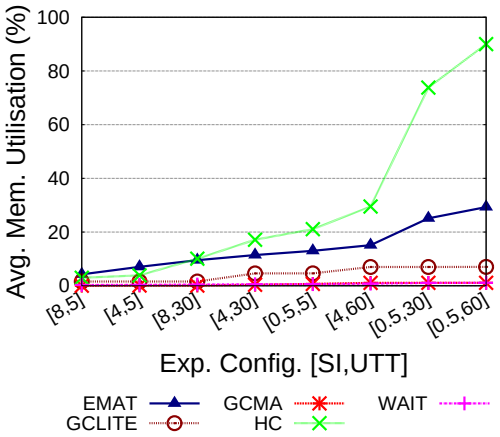

Figure 26. Avg. MEM Utilisations per Tool.

5.4. Experiment #4: Proposed Policies Assessment

The objective of this experiment was to evaluate the behaviour of PHOEBE, as well as the set of proposed policies, in order to assess how well they have fulfilled their purpose of addressing the identified trade-offs without the need for manual intervention from the tester. The following sections describe this experiment and its results.

5.4.1. Experimental Set-up. The set-up was similar to that used in the experiment #1 (presented in Section 5.1.1), with the following differences: First, the adaptive policies took the place of the manual configurations of the SI and UI parameters. Furthermore, the policies used the following configurations: For the accuracy policy, a 20% response time threshold was defined. This value was suggested by IBM 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 executed at least one. Finally, the minimum SI and the ΔSI were set to 30 seconds. Regarding the efficiency policy, the initial UI was set to 60 minutes (time range commonly used in the industry to monitor performance test runs); and the ΔUI was set to 15 minutes. Finally, the CPU and memory maximum utilisation thresholds were set to 90% to avoid saturation of these resources (scenario which should be avoided for optimal performance, as demonstrated by [69]). Regarding the consolidation policy, the two-level severity style (described in 4.6.2) was used. This parameter was suggested by IBM to simplify the classification of performance bugs (between critical and non-critical) as usually only the critical ones are of interest in performance testing. The severity threshold was set to a 50% frequency (so that the critical category would conceptually overlap the ISTQB critical and major categories). Finally, the CV-based Go No-Go assessment (described in Section 4.6.3) was enabled. It was configured to assess the results of the latest two hours for all the severity categories. The CV threshold was set to 0.1 for the critical category, while 0.3 for the non-critical one.

5.4.2. Experimental Results. In this experiment, the analysis focused on two main aspects: Evaluating the accuracy of the implemented policies, and assessing the productivity gains that PHOEBE brought to the performance testing process. Therefore, the obtained results were compared against the results from the previously performed assessments of trade-offs (discussed in Sections 5.1, 5.2 and 5.3).

Accuracy-Target Data Gathering Policy. The first part of the analysis focused on evaluating the accuracy policy. In terms of performance overhead, the results demonstrated that the accuracy policy worked well, as it was possible to finish the test with the overhead caused by the diagnosis tools within the desired threshold. This was the result of increasing the SI when the threshold was exceeded to reduce its performance impact. For instance, this adjustment for WAIT involved that the SI was increased twice, moving from its initial value of 30 seconds to 60 seconds, then to a final

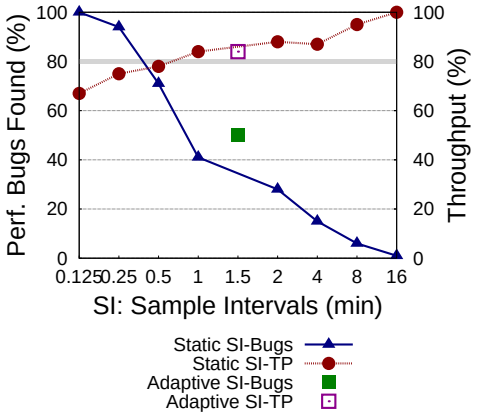

Figure 27. Perf. Bugs vs. Throughput - WAIT.

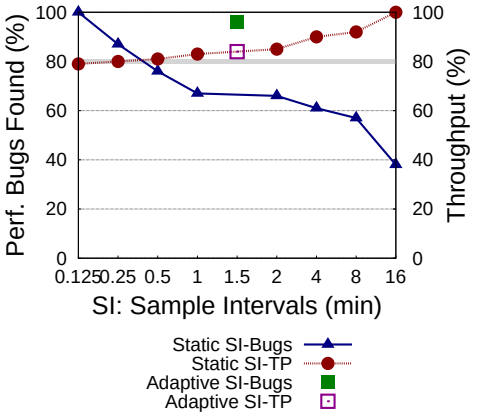

Figure 28. Perf. Bugs vs. Throughput - HC.

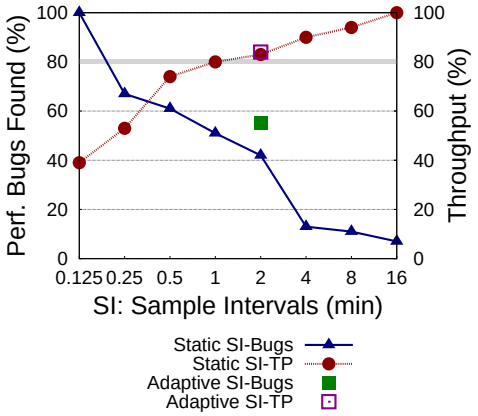

Figure 29. Perf. Bugs vs. Throughput - EMAT.

value of 90 seconds. Regarding bug coverage, the number of bugs found with the adaptive policy was always higher than the number of bugs found with the corresponding static SI (e.g., 90 seconds in the case of WAIT). This was the result of using other (smaller) SIs during the test, situation which provoked that the bug coverage was higher (compared to the corresponding static SI) during certain periods of the test. The results obtained for WAIT, EMAT and HC are presented in Figures 27, 29 and 28, respectively. In the figures, the response time threshold is shown as a grey horizontal line. Furthermore, the same analysis was done considering only the critical bugs, and similar behaviours were observed. An example is shown in Figure 30, which presents the results obtained with WAIT, comparing them against the results obtained when using a static SI. Finally, the results for GCMA and GCLITE are not presented because these tools are insensitive to the SI selection. Therefore, the SI did not change for them (remaining in their original value of 0.5 minutes).

Efficiency-Target Upload Policy. The second part of the 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 services (i.e. the used diagnosis tools in this scenario). This was the result of decreasing the UI when the threshold was exceeded in order to reduce the resource utilisation. For instance, in the case of WAIT, the CPU_{AVG} of the first round of uploads (which occurred before any adjustment) was 90.7%. As this value exceeded the target of maximum utilisation (U_{MAX}), the efficiency policy adjusted the UIs of the nodes after the first round of uploads. After the UI adjustment, the CPU_{AVG} decreased to 65.7%, remaining below the U_{MAX} during the rest of the test. Similarly, the UIs used by was GCLITE and HC during the performance test runs were adjusted. For GCLITE, the adjustment was triggered by CPU utilisation, while for HC it was triggered by memory utilisation. Meanwhile, GCMA and EMAT did not require an UI adjustment. This was because these tools never exceeded the U_{MAX} of any monitored resource

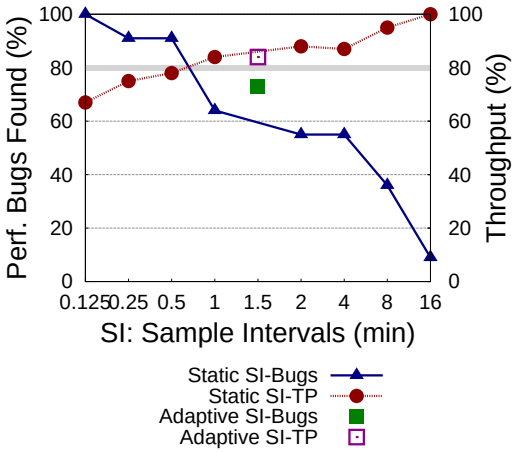

Figure 30. Critical Perf. Bugs vs. Throughput - WAIT.

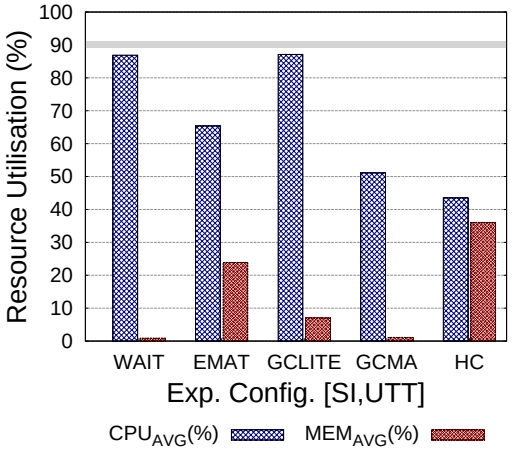

Figure 31. Resources vs. Configuration - All Tools.

during their execution. These behaviours are shown in Figure 31, which presents the average CPU and memory utilisations achieved by each diagnosis tool. In the figure, the U_{MAX} is shown as a grey horizontal line.

Multi-View Consolidation Policy. The third part of the analysis concentrated on evaluating the amount of effort-savings gained by using PHOEBE. This analysis identified two main types of savings: Those in the performance analysis tasks, and those in the performance testing tasks.

Regarding the performance analysis tasks, as PHOEBE is able to self-configure to keep a test run within the desired constraints (e.g., the overhead threshold), a tester does not longer risk to select an inappropriate configuration (among a set of candidate configurations). Therefore, the tester only needs to run one performance test run. For instance, assuming there are eight candidate configuration sets (such as the ones evaluated in experiment #1), a tester would be saving 87.5% of the time required to test the complete configuration spectre (as only one test run is needed, instead of eight).

To complement the analysis, by offering a more conservative perspective of the obtained gains, the adaptive runs were also compared against the best and worst performers (among the static experimental configurations tested in experiment #1). This comparison also showed that PHOEBE worked well, as the achieved improvements were also very significant. The considerable decrements in effort were the result of the automation of most of the analysis tasks previously done manually.

A summary of the results is presented in Figure 32, which shows the efforts required by the testers (per diagnosis tool) for four different experimental configurations: The adaptive run, the

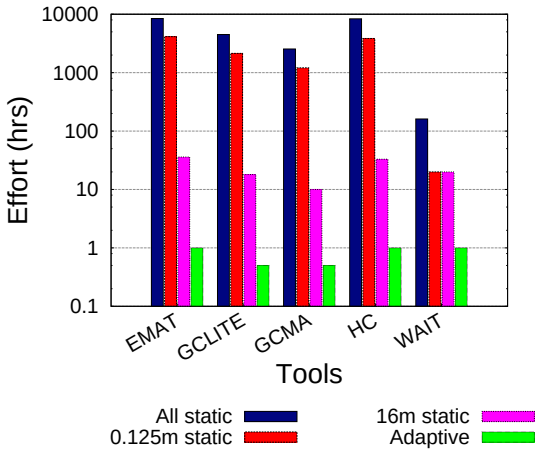

Figure 32. Performance Analysis Effort-savings.

best performer among the static runs (i.e. SI of 16 minutes), the worst performer among the static runs (i.e. SI of 0.125 minutes), and all the static runs together. There, it can be noticed how the adaptive runs drastically outperformed their static counterparts: Compared against all the static configurations, the decrements in effort ranged between 99.4% and 99.9% (with an average of 99.8% and a standard deviation of 0.3%). Compared against the worst performer (i.e. the static SI of 0.125 minutes), the decrements ranged between 95% and 99.9% (with an average of 98.9% and a standard deviation of 2.2%). Finally, compared against the best performer (i.e. the static SI of 16 minutes), the decrements ranged between 95% and 97.2% (with an average of 96.3% and a standard deviation of 1.2%).

Furthermore, PHOEBE was able to save additional time to the performance testing process. This was due to two main reasons:

- As previously discussed, a tester no longer needs to try different configuration sets. This behaviour directly translates into time-savings (in terms of test runs' execution time). For instance, in this experiment these reductions ranged between 95% and 98% (with an average of 97% and a standard deviation of 0.8%).

- The usage of the proposed CV-based Go No-Go assessment (discussed in Section 4.6.3) allowed PHOEBE to identify the moment (during the test run execution) in which the test run has exhausted its benefits (in terms of identified bugs). Whenever it occurred, the test run was stopped, bringing additional time-savings to the process. The exact time varied among the diagnosis tools (as it was influenced by the pace and amount of bugs identified by each tool) but in all cases considerable time was saved: Compared against any of their static counterparts, the time-savings ranged between 66% and 81% (with an average of 75% and a standard deviation of 6.6%). The results are summarised in Figure 33 which compares the results obtained by the adaptive test runs (per diagnosis tool) against their static counterparts.

Summary. The results of this experiment demonstrated how PHOEBE, through the set of proposed policies, achieved the intended goals: The accuracy policy kept the performance overhead introduced into the monitored application nodes within the desired threshold, while maximising the number of bugs found within the constrained conditions. Likewise, the efficiency policy decreased the resource utilisation of the shared service (i.e. the diagnosis tools), minimising the possibility of its saturation. Finally, the usage of PHOEBE proved how it can drastically decrease the amount of effort/time spent by testers during the processes of performance testing and analysis.

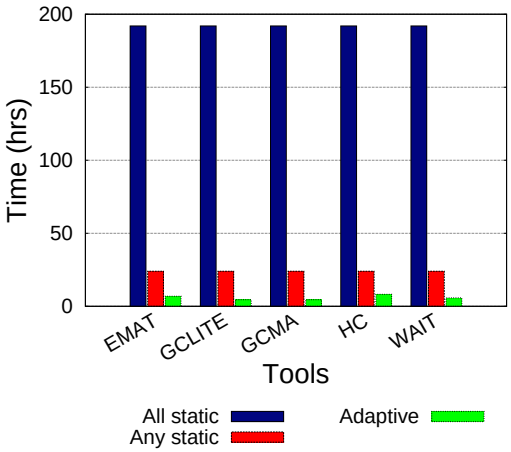

Figure 33. Performance Testing Time-savings.

5.5. *Experiment #5: Testing Productivity Assessment*

Here the objective was to further assess the benefits that PHOEBE brings to a performance tester (in terms of reduced effort and time) by comparing the automated usage of a diagnosis tool through PHOEBE against a purely manual usage of a diagnosis tool. The following sections describe this experiment and its results.

5.5.1. Experimental Set-up. The experimental set-up was similar to that used in the experiment #4 (presented in Section 5.4.1), with the following differences: First, the iBatis JPetStore 4.0 [84] application was used (with a workload of 2,000 concurrent users). JPetStore was selected because it is a well-documented open source application which is also easy to use. Second, the source code of JPetStore was modified to inject five performance issues (two lock contentions, two deadlocks and one I/O latency bug). As experiment #4 proved that PHOEBE works well irrespective of the diagnosis tool, we focused on WAIT. This tool was chosen following a suggestion made by our industrial partner (whom considered WAIT the most interesting tool, among the evaluated ones, due to its strong analytic capabilities). As WAIT had also shown to achieve the "lowest" (still highly significant) potential gains, its usage allowed us to define an improvement baseline (as the improvement in the other diagnosis tools would be higher). Finally, no Go No-Go assessment was configured in order to test PHOEBE in a reliability-type performance test run.

5.5.2. Experimental Results. 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 the automation framework (A-WAIT). In both cases, the tester did not know the number or characteristics of the injected bugs. The results of this experiment are summarised in Table I.

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 considerably 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/hour) 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 PHOEBE, it was possible to feed WAIT incrementally during the test run execution (in contrast to manual WAIT, where the tester needed to wait until the end of the performance test run). This behaviour allowed the tester using the automated WAIT to easily get intermediate results during the test run. In this experiment, all the bugs were identified by the tester using the automated WAIT after the first hour of test execution. Therefore, the tester was able to start the analysis of those bugs in parallel to the rest of the test run execution (which the tester

Table I. M-WAIT and A-WAIT Comparison.

Metric	M-WAIT (hr)	A-WAIT (hr)	M-WAIT vs. A-WAIT (%)
a. Duration of performance testing activity	32.8	24.1	-27%
b. Duration of performance testing	24.0	24.0	0%
c. Effort of performance analysis (d+e)	8.8	4.2	-52%
d. Effort of bug identification	6.8	2.2	-68%
e. Effort of root cause analysis	2.0	2.0	0%

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 run, 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.

An additional observation from this experiment is that the time savings gained by PHOEBE are directly related to the duration of the test and the number of application nodes in the environment. This behaviour (which reinforced the results obtained in experiment #2, discussed in Section 5.2.2) 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.

Summary. To summarise the experimental results, they allowed to further measure the productivity benefits that a tester can gain by using a diagnosis tool through PHOEBE. In particular, two time savings were documented: The effort required to identified bugs was significantly reduced (68% in this case), as well as the total duration of the testing activities (27% in this 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.

5.6. Discussion for Practitioners

The presented experimental results have demonstrated how automating the configuration and usage of a diagnosis tool can significantly improve the performance testing process. In the following paragraphs, we provide guidelines for practitioners to indicate the conditions under which PHOEBE can yield improvements and discuss the wider applicability of the technique.

- As discussed in Section 2.1, performance testing is usually performed multiple times during a software project (i.e., it is normally executed after a new version of the software is built). As there are usually budget or schedule constraints in such projects, using PHOEBE with a Go No-Go assessment (like the one discussed in Section 4.6.3) can be useful in the early phases of a project to make the most out of the performance testing, without the need of (necessarily) executing the test for its whole planned duration (e.g., 24-hours). On the contrary, in final phases of the project (or whenever the goal of the performance testing is to assess the reliability), PHOEBE should better be used without a Go No-Go assessment so that the test last the planned duration.

- An adaptive SI (like the policy described in Section 4.4.1) is useful when a sample-based diagnosis tool is used (e.g., WAIT, HC or EMAT). This is because the overhead introduced by a diagnosis tool into the application nodes is normally caused by the sampling process (e.g., it is widely-known that the generation of a heapdump is a very time-consuming process). Furthermore, the selection of an appropriate SI (i.e. one that will introduce a tolerable level of overhead) might vary depending on the particular usage scenario (e.g., the application-under-test or the workload used for testing). Under these conditions, an adaptive SI is preferable

as it frees the tester from the burden of manually configuring it. As the experimental results have shown, the costs of incorrectly selecting an appropriate SI can be high and considerable effort/time can be wasted. On the contrary, a static SI can be a better fit for those diagnosis tools that leverage on traces (e.g. GCMV or GCLITE, which relies on GC verbose). This is because that type of tools generate a constant overhead (regardless of the chosen SI).

- An adaptive UI (like the policy described in Section 4.4.2) is better suitable for resource-intensive tools and large-scale clustered applications. This is because, under those conditions, the possibility of saturating the diagnosis tool (due to the concurrent processing of multiple samples) is more likely. Likewise, an adaptive UI is suitable for enabling shared services (i.e. diagnosis tool servers which are shared across different testing teams or projects). This is because, as there are usually budget or schedule constraints in software projects, the amount of available licenses for a diagnosis tool is normally limited. Under these conditions, the capability of PHOEBE to efficiently enable a diagnosis tool as a shared service is highly desirable.

- In the experimental evaluation, we selected five of the most widely-used Java diagnosis tools in the industry. As the results have shown, the achieved time/effort savings are evident for all the tested diagnosis tools, and so it is expected that PHOEBE can yield similar results when using other Java diagnosis tools (especially those using the same type of inputs - i.e. heapdumps, javacores or GC verbose -). Likewise, it is expected that PHOEBE should be applicable to diagnosis tools used in other object-oriented languages (e.g., Python or C#) as long as the appropriate data analytics helpers are developed to interface with those tools.

- In the experimental evaluation, the participant testers were experienced ones. As the results have shown, the achieved time/effort savings are evident, and so it is expected that PHOEBE can yield better results when used by more inexperienced testers (i.e. undergraduates or postgraduates with little or no working experience in software development or testing). Likewise, a simplified two-level severity classification was used in the experimental evaluation. It is expected that PHOEBE can save more analysis effort whenever a more granular severity classification is used (e.g., the ISTQB-based one discussed in Section 4.6.2).

- In terms of the potential time savings that PHOEBE can achieve, they are directly related to the duration of the test and the number of application nodes in the environment. Therefore, the biggest time savings are obtained when the performance testing uses long-term runs (e.g., one or more days) and the testing environment is highly distributed (i.e. it is composed of multiple application nodes). Under these conditions, PHOEBE is able to mitigate most of the effort required to use a diagnosis tool in performance testing. As the effort is also considerable (hence offering a lot of potential savings), PHOEBE can convert them into actual time savings. It is also worth mentioning that time savings can usually be expected. This is because PHOEBE significantly reduces the effort and expertise required to use a diagnosis tool, independently of the duration of the test or the number of nodes (e.g., a tester only needs to monitor a single report which is automatically updated during the test run execution).

- As of now, PHOEBE has centred on interfacing with a human user (i.e. testers). However, PHOEBE might interact with other non-human actors. For instance, a policy might be in charge of reporting issues to a bug tracking system (e.g., bugzilla [85]) or interface with an e-mail server to communicate relevant events (e.g., the occurrence of a critical issue) to any interested parties (e.g., testers or developers). Likewise, PHOEBE has been tested within the performance testing domain. However, PHOEBE might also be suitable to monitor non-testing environments (e.g., production servers). In that scenario, PHOEBE might interface with a capacity management system to report relevant events (e.g., whenever a critical issue arises, serious performance degradations are observed or a resource utilisation exceeds a defined threshold).

- Based on the previously discussed points, it is concluded that a framework that automates the usage of a diagnosis tool in a clustered testing environment can offer significant benefits to the performance testing process. Given the broad spectrum of functional behaviours and workloads that an application might experience, such framework should not rely on a static configuration. On the contrary, it should be able to adapt to the non-functional characteristics of the underlying application (as PHOEBE does). As there are similarities in the tasks usually performed by a tester on the performance testing and analysis of an application (using a diagnosis tool), such tasks can be abstracted into policies (such as the ones proposed in Section 4.4). This strategy can then be leveraged to make a more robust framework.

6. CONCLUSIONS AND FUTURE WORK

The identification of performance problems in clustered environments is complex and time-consuming. Even though researchers have been developing diagnosis tools to simplify this task, various limitations exist in those tools that prevent their effective usage in performance testing. To address this challenge, in our previous work we presented PHOEBE, a novel adaptive framework that automates the usage of a performance diagnosis tool in a clustered environment. The aim was to improve a tester's productivity by decreasing the effort and expertise needed to use diagnosis tools. Internally, PHOEBE utilises a set of policies to control the different set of processes commonly involved in the configuration and usage of a diagnosis tool. The aim of this paper was to extend our previous work by broadening the set of policies available in PHOEBE in order to cover the whole spectrum of processes (i.e. sample gathering, sample processing and results' consolidation) normally involved on the usage of a diagnosis tool in the performance testing of a clustered application, as well as by performing a comprehensive assessment of PHOEBE in terms of its benefits, costs and generality (with respect to the used diagnosis tool). For this purpose, a prototype was developed around a set of well-known diagnosis tools to experimentally evaluate the framework. First, the different trade-offs that are commonly experienced when using a diagnosis tool (i.e. bug finding accuracy, testers' effort and resource utilisations) were evaluated. Then, the time/effort savings obtained by a tester through the usage of PHOEBE (and its set of policies) were assessed.

The obtained experimental results have demonstrated that relevant time savings can be gained by applying the proposed framework: Not only the effort and expertise required to uses the different diagnosis tools were significantly reduced (between 95% and 99.9%), but also the total duration of the performance testing was considerably reduced (between 66% and 98%). These time savings were achieved independent of the used diagnosis tool, proving the generality of PHOEBE. The results have also demonstrated that such an adaptive policy-enabled framework is capable of simplifying the configuration of a diagnosis tool. This was achieved by automatically addressing the trade-offs identified in each tool without the need for manual intervention from the tester. Thus, the framework has demonstrated to simplify the usage of a diagnosis tool and to reduce the time required to analyse performance issues, thereby reducing the costs associated with performance testing. From the above results, we conclude that an automation framework, focused on effectively addressing the common usage limitations experienced by a diagnosis tool, can bring significant benefits to the performance testing of clustered applications.

In our future work, we will continue investigating how best to extend the capabilities of the framework. For instance, PHOEBE has exclusively leveraged on the qualitative data (i.e. performance bugs or tuning recommendations) that each diagnosis tool provides. However, those tools also provide a considerable amount of quantitative data that can be exploited. Hence, PHOEBE can be extended to use that data to identify additional types of issues, in the form of performance anti-patterns. Likewise, PHOEBE has also mainly focused on controlling the behaviour of the performance diagnosis tools. However, it can be extended to control other actors in the performance testing process. For example, policies can be developed to dynamically enable (or disable) diagnosis actions once it is suspected that a performance issue is occurring (e.g., varying the level of logging).

ACKNOWLEDGEMENTS

This work was supported, in part, by Science Foundation Ireland grant 10/CE/I1855 to Lero - the Irish Software Research Centre (www.lero.ie). Our thanks to Patrick O'Sullivan (IBM Dublin Software Lab), as his experience in performance testing helped us through the scope definition and validation, as well as for giving us the opportunity to work with his IBM System Verification Teams.

REFERENCES

1. Compuware. *Applied Performance Management Survey*. 2007.
2. Lee WY, Hong SJ, Kim J, Lee S. Dynamic load balancing for switch-based networks. *Journal of Parallel and Distributed Computing* 2003; **63**(3):286–298.
3. Portillo-Domínguez AO, Murphy J, O'Sullivan P. Leverage of extended information to enhance the performance of jee systems. *Information Technology and Telecommunications Conference*, 2012.
4. Angelopoulos V, Parsons T, Murphy J, O'Sullivan P. GcLite: An Expert Tool for Analyzing Garbage Collection Behavior. *Annual Computer Software and Applications Conference Workshops* 2012; .
5. Parsons T, Murphy J. Detecting Performance Antipatterns in Component Based Enterprise Systems. *International Middleware Doctoral Symposium*, 2008.
6. Portillo-Dominguez AO, Wang M, Murphy J, Magoni D. Automated wait for cloud-based application testing. *Software Testing, Verification and Validation Workshops (ICSTW), 2014 IEEE Seventh International Conference on*, IEEE, 2014; 370–375.
7. Woodside M, Franks G, Petriu DC. The Future of Software Performance Engineering. *Future of Software Engineering* 2007; .
8. Spear W, Shende S, Malony A, Portillo R, Teller PJ, Cronk D, Moore S, Terpstra D. Making Performance Analysis and Tuning Part of the Software Development Cycle. *DoD High Performance Computing Modernization Program Users Group Conference* 2009; .
9. Altman E, Arnold M, Fink S, Mitchell N. Performance analysis of idle programs. *ACM SIGPLAN Notices* Oct 2010; **45**(10).
10. Ammons G, Choi Jd, Gupta M, Swamy N. Finding and Removing Performance Bottlenecks in Large Systems. *European Conference on Object-Oriented Programming*, 2004.
11. Portillo-Dominguez AO, Wang M, Murphy J, Magoni D, Mitchell N, Sweeney PF, Altman E. Towards an automated approach to use expert systems in the performance testing of distributed systems. *Proceedings of the 2014 Workshop on Joining AcadeMiA and Industry Contributions to Test Automation and Model-Based Testing*, ACM, 2014; 22–27.
12. Jiang ZM. Automated analysis of load testing results. *International Symposium on Software Testing and Analysis* 2010; :143.
13. IBM Rational Performance Tester. URL `http://www-03.ibm.com/software/products/en/performance`, last accessed: 2016-06-06.
14. Apache JMeter. URL `http://jmeter.apache.org/`, last accessed: 2016-06-06.
15. HP LoadRunner. URL `http://www8.hp.com/ie/en/software-solutions/loadrunner-load-testing/`, last accessed: 2016-06-06.
16. Adrion WR, Branstad MA, Cherniavsky JC. Validation, verification, and testing of computer software. *ACM Computing Surveys* 1982; **14**(2):159–192.
17. International Data Corporation (IDC). Java : Two and a Half Years After the Acquisition. *Technical Report August* 2012.
18. TIOBE Programming Index. URL `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`, last accessed: 2016-06-06.
19. Eclipse Memory Analyzer. URL `https://eclipse.org/mat/`, last accessed: 2016-06-06.
20. The Java Virtual Machine. URL `https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-1.html#jvms-1.2`, last accessed: 2016-06-06.
21. Heap Dump. URL `http://help.eclipse.org/mars/topic/org.eclipse.mat.ui.help/concepts/heapdump.html?cp=46_2_0`, last accessed: 2016-06-06.
22. Sun Microsystems. *Memory Management in the Java HotSpot Virtual Machine*. 2006.
23. Portillo-Dominguez AO, Perry P, Magoni D, Wang M, Murphy J. Trini: an adaptive load balancing strategy based on garbage collection for clustered java systems. *Software: Practice and Experience* 2016; .
24. GC Verbose. URL `http://www.oracle.com/technetwork/java/javase/clopts-139448.html#gbmtm`, last accessed: 2016-06-06.
25. IBM Garbage Collection and Memory Visualizer. URL `https://www.ibm.com/developerworks/java/jdk/tools/gcmv/`, last accessed: 2016-06-06.
26. IBM Health Center. URL `http://www.ibm.com/developerworks/java/jdk/tools/healthcenter/`, last accessed: 2016-06-06.
27. IBM WAIT Tool. URL `https://wait.ibm.com/`, last accessed: 2016-06-06.
28. Wu H, Tantawi AN, Yu T. A self-optimizing workload management solution for cloud applications. *International Conference on Web Services*, 2013; 483–490.
29. Deep Dive on Java Virtual Machine (JVM) Javacores and Javadumps. URL `http://www-01.ibm.com/support/docview.wss?uid=swg27017906&aid=1`, last accessed: 2016-06-06.
30. Albert, Elvira J Miguel Gomez-Zamalloa. Resource-Driven CLP-Based test case generation. *Logic-Based Program Synthesis and Transformation* 2012; .

31. Chen S, Moreland D, Nepal S, Zic J. Yet Another Performance Testing Framework. *Australian Conference on Software Engineering* 2008; .
32. L C Briand M Y Labiche. Using genetic algorithms for early schedulability analysis and stress testing in real-time systems. *Genetic Programming and Evolvable Machines* 2006; .
33. M S Bayan J. Automatic stress and load testing for embedded systems. *Computer Software and Applications Conference* 2006; .
34. V Garousi Y L C Briand. Traffic-aware stress testing of distributed systems based on uml models. *International Conference on Software Engineering* 2006; .
35. Xingen Wang W Bo Zhou. Model-based load testing of web applications. *Journal of the Chinese Institute of Engineers* 2013; .
36. J Zhang S. Automated test case generation for the stress testing of multimedia systems. *Software: Practice and Experience* 2002; .
37. Chan EYK, Chan WK, Poon PL, Yu YT. An empirical evaluation of several test-a-few strategies for testing particular conditions. *Software: Practice and Experience* 2012; **42**(8):967–994.
38. Manolache L, Kourie DG. Software testing using model programs. *Software: Practice and Experience* 2001; **31**(13):1211–1236.
39. Kumar M, Sharma A, Kumar R. An empirical evaluation of a three-tier conduit framework for multifaceted test case classification and selection using fuzzy-ant colony optimisation approach. *Software: Practice and Experience* 2015; **45**(7):949–971.
40. Liu H, Xie X, Yang J, Lu Y, Chen TY. Adaptive random testing through test profiles. *Software: Practice and Experience* 2011; **41**(10):1131–1154.
41. Shihab E, Jiang ZM, Adams B, Hassan AE, Bowerman R. Prioritizing the creation of unit tests in legacy software systems. *Software: Practice and Experience* 2011; **41**(10):1027–1048.
42. Gonzalez-Sanchez A, Piel É, Abreu R, Gross HG, van Gemund AJ. Prioritizing tests for software fault diagnosis. *Software: Practice and Experience* 2011; **41**(10):1105–1129.
43. Guo HF, Qiu Z. A dynamic stochastic model for automatic grammar-based test generation. *Software: Practice and Experience* 2015; **45**(11):1519–1547.
44. Yu, Kai E. Practical isolation of failure-inducing changes for debugging regression faults. *IEEE/ACM International Conference on Automated Software Engineering* 2012; .
45. Li, Mengchen E. Dynamically Validating Static Memory Leak Warnings. *International Symposium on Software Testing and Analysis* 2013; .
46. Csallner C, Smaragdakis Y. Jcrasher: an automatic robustness tester for java. *Software: Practice and Experience* 2004; **34**(11):1025–1050.
47. Barham P, Donnelly A, Isaacs R, Mortier R. Using magpie for request extraction and workload modelling. *Symposium on Opearting Systems Design and Implementation* 2004; .
48. Hoorn V, Rohr M, Hasselbring W, Waller J, Ehlers J, Kieselhorst D. Continuous Monitoring of Software Services: Design and Application of the Kieker Framework 2009; .
49. Dobre C, Voicu R, Legrand IC. Monalisa: A monitoring framework for large scale computing systems. *International Journal of Computing* 2014; **11**(4):351–366.
50. Xiao, Xusheng E. Context-Sensitive Delta Inference for Identifying Workload-Dependent Performance Bottlenecks. *International Symposium on Software Testing and Analysis* 2013; .
51. Yu, Tingting G Witawas Srisa-an. SimRacer: an automated framework to support testing for process-level races. *International Symposium on Software Testing and Analysis* 2013; .
52. Barr, E M Christian Bird. Collecting a Heap of Shapes. *International Symposium on Software Testing and Analysis* 2013; .
53. Delahaye M, Bousquet L. Selecting a software engineering tool: lessons learnt from mutation analysis. *Software: Practice and Experience* 2015; **45**(7):875–891.
54. Introduction to Instrumentation and Tracing. URL `https://msdn.microsoft.com/en-us/library/aa983649%28v=vs.71%29.aspx`, last accessed: 2016-06-06.
55. Yang J, Evans D, Bhardwaj D, Bhat T, Das M. Perracotta: mining temporal api rules from imperfect traces. *International Conference on Software Engineering* 2008; .
56. C Csallner Y. Dsd-crasher: a hybrid analysis tool for bug finding. *International Symposium on Software Testing and Analysis* 2006; .
57. S Hangal M. Tracking down software bugs using automatic anomaly detection. *International Conference on Software Engineering* 2002; .
58. Chen MY, Accardi A, Kiciman E, Lloyd J, Patterson D, Fox A, Brewer E. Path-based failure and evolution management. *USENIX Symposium on Networked Systems Design and Implementation* 2004; .
59. Jiang ZM, Hassan AE, Hamann G, Flora P. Automated performance analysis of load tests. *International Conference on Software Maintenance*, 2009; 125–134, doi:10.1109/ICSM.2009.5306331. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5306331`.
60. Haroon Malik A Bram Adams. Pinpointing the subsys responsible for the performance deviations in a load test. *Software Reliability Engineering* 2010; .
61. M Salehie L. Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems* 2009; .
62. P Robertson H R Laddaga. Introduction: the First International Workshop on Self-Adaptive Software. *International Workshop on Self-Adaptive Software* 2000; .
63. Weyns, Danny J M Usman Iftikhar. Do external feedback loops improve the design of self-adaptive systems? a controlled experiment. *International Symposium on Software Engineering for Adaptive and Self-Managing Systems* 2013; .
64. J Kephart D. The vision of autonomic computing. *Computer* Jan 2003; .

65. UML basics: The component diagram. URL https://www.ibm.com/developerworks/rational/library/dec04/bell/, last accessed: 2016-06-06.
66. Factory Pattern. URL http://www.oodesign.com/factory-pattern.html, last accessed: 2016-06-06.
67. Command Pattern. URL http://www.oodesign.com/command-pattern.html, last accessed: 2016-06-06.
68. Load Test Run Settings Properties. URL https://msdn.microsoft.com/en-us/library/ff406976.aspx, last accessed: 2016-06-06.
69. Rolia, Jerry M Artur Andrzejak. Automating enterprise application placement in resource utilities. *Self-Managing Distributed Systems* 2003; .
70. Return on investment. URL http://www.businessdictionary.com/definition/return-on-investment-ROI.html, last accessed: 2016-06-06.
71. Ruggeri F, Kenett R, Faltin FW. *Encyclopedia of statistics in quality and reliability*. John Wiley, 2007.
72. Yancey WE. Evaluating string comparator performance for record linkage. *Statistical Research Division Research Report* 2005; .
73. International Software Testing Qualifications Board. URL http://www.istqb.org/, last accessed: 2016-06-06.
74. Defect Severity. URL http://softwaretestingfundamentals.com/defect-severity/, last accessed: 2016-06-06.
75. Eclipse Jetty. URL http://www.eclipse.org/jetty, last accessed: 2016-06-06.
76. Apache Tomcat. URL http://tomcat.apache.org/, last accessed: 2016-06-06.
77. Dell PowerEdge T420 Tower Server. URL http://www.dell.com/ie/business/p/poweredge-t420/pd, last accessed: 2016-06-06.
78. KVM Hypervisor. URL http://www.linux-kvm.org/, last accessed: 2016-06-06.
79. The DaCapo Benchmark Suite. URL http://dacapobench.org/, last accessed: 2016-06-06.
80. DaCapo Benchmark Suite - Programs and Sample Sizes. URL http://www.dacapobench.org/benchmarks.html, last accessed: 2016-06-06.
81. Poon PL, Tse T, Tang SF, Kuo FC. Contributions of tester experience and a checklist guideline to the identification of categories and choices for software testing. *Software Quality Journal* 2011; **19**(1):141–163.
82. Generating Javacores and Userdumps Manually For Performance, Hang or High CPU Issues on Windows. URL http://www-01.ibm.com/support/docview.wss?uid=swg21138203, last accessed: 2016-06-06.
83. Generating heap dumps manually. URL https://www-01.ibm.com/support/knowledgecenter/SSAW57_8.5.5/com.ibm.websphere.nd.doc/ae/tprf_generatingheapdumps.html, last accessed: 2016-06-06.
84. iBATIS JPetStore. URL http://sourceforge.net/projects/ibatisjpetstore/, last accessed: 2016-06-06.
85. Bugzilla. URL https://www.bugzilla.org/, last accessed: 2016-06-06.

APPENDIX A: DACAPO BENCHMARK

Nowadays, DaCapo is one of the Java benchmarks most widely-used in the literature. The following paragraphs briefly describe the version 9.12 of this benchmark, which is the latest release of the benchmark as well as the version used in this paper.

This benchmark has been developed by the DaCapo research project, which has been sponsored by companies such as IBM, Intel, and Microsoft; and institutions such as the Australian Research Council. The benchmark is composed of 14 different programs. They are all open source, real-world applications, and with non-trivial memory loads [79]. Table II lists these programs and briefly describes their functionality.

Table II. DaCapo Programs

Name	**Description**
avrora	A program that simulates a set of programs running on a grid of microcontrollers.
batik	A program that processes a set of vector-based images.
eclipse	A program that executes a set of performance tests in an eclipse development environment.
fop	A program that generates PDF files based on a set of XSL-FO files that are parsed and formatted.
h2	A program that executes a set of banking transactions against a database-centric application.
jython	A program that executes a set of python scripts in Java.
luindex	A program that indexes a set of documents.
lusearch	A program that performs a set of keyword searchs over a corpus of data.
pmd	A program that reviews a set of Java classes, looking for bugs in their source code.
sunflow	A program that renders a set of images.
tomcat	A program that executes a set of queries against a Tomcat server.
tradebeans	A program that executes a set of stock transactions, via Java Beans calls, using an Apache Geronimo/h2 backend.
tradesoap	A program that executes a set of stock transactions, via SOAP calls, using an Apache Geronimo/h2 backend.
xalan	A program that transforms a set of XML files into HTML files.