Performance Optimisation of Clustered Java Systems

by

Andres Omar Portillo Dominguez

This thesis is submitted to University College Dublin in fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science.

School of Computer Science

Head of School: Prof. Padraig Cunningham
Supervisors: Prof. John Murphy and Dr. Philip Perry

April 2016
Abstract

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 optimisation of such clustered systems. Therefore, automatic techniques are needed to facilitate these performance-related activities, which otherwise would be highly error-prone and time-consuming. This thesis contributes to the area of performance optimisation of clustered systems in Java (a predominant technology at enterprise-level), especially aiming for large-scale environments. This thesis proposes two techniques to solve the problems of efficiently identifying workload-dependent performance issues and efficiently avoiding the performance impacts of major garbage collection, two problems that a typical clustered Java system would likely suffer in large-scale environments. In particular, this thesis introduces an adaptive framework to automate the usage of performance diagnosis tools in the performance testing of clustered systems. The aim is to ease the identification of performance issues by decreasing the effort and expertise needed to effectively use such tools. Additionally, an adaptive GC-aware load balancing strategy is introduced, which leverages on major garbage collection forecasts to decide on the best way to balance the workload across the available nodes. The aim is to improve the performance of a clustered system by avoiding the impacts in the cluster’s performance due to the major garbage collection occurring at the individual nodes. Experimental results of applying these techniques to a set of real-life applications are presented, showing the benefits that the techniques bring to a clustered Java system.
Statement of Original Authorship

“I hereby certify that the submitted work is my own work, was completed while registered as a candidate for the degree of Doctor of Philosophy, and I have not obtained a degree elsewhere on the basis of the research presented in this submitted work.”
I would like to take this opportunity to thank the people who supported me to make the work in this thesis possible.

Firstly, I would like to thank my family. They are an inspiration for me, showing me what it means to feel someone near even though they are thousands of kilometres away. In particular, I would like to deeply thank my wife, Vanessa Ayala Rivera. Without her constant and unconditional love and support, this work would not have been possible.

Secondly, I would like to sincerely thank my supervisors Prof. John Murphy and Dr. Philip Perry for their support and guidance throughout my PhD studies. Also very special thanks to Prof. Liam Murphy and Prof. Damien Magoni for their advice, guidance and encouragement during this journey. I would also like to thank Dr. Miao Wang, Dr. Nicola Stokes and Dr. Dawid Nowak for providing me guidance and suggestions to improve my research work. Finally, thanks to all my other colleagues in the Performance Engineering Lab for their friendship, assistance and helpful discussions throughout my PhD studies.

I would also like to thank Dr. Patrick O’Sullivan (IBM Dublin Software Lab) and Dr. Erik R. Altman (IBM Thomas J. Watson Research Centre) for giving me the opportunity to work with their System Verification (SVT) and Whole Analysis Idle Time (WAIT) teams. Furthermore, many thanks to Amarendra Darisam (IBM SVT), Haytham Assem (IBM SVT), Dr. Nick M. Mitchell (IBM WAIT) and Dr. Peter F. Sweeney (IBM WAIT) for giving me useful feedback and valuable advice on my research work, based on their industrial experience.

Finally, I would like to acknowledge that my research was supported by the Science Foundation Ireland through the grant 10/CE/I1855 to Lero, the Irish Software Engineering Research Centre (www.lero.ie).
To my dear family and to my lovely wife.
Contents

Abstract i
Statement of Original Authorship ii
Acknowledgements iii
Dedicatory iii
Contents iv
List of Figures vii
List of Tables x
List of Publications xii
List of Patents xiii

1 Introduction 1
  1.1 Motivation ........................................ 1
  1.2 Thesis Contributions ............................... 4
  1.3 Thesis Overview .................................... 6

2 State of the Art 7
  2.1 Background ......................................... 7
    2.1.1 Java Virtual Machine (JVM) ......................... 7
    2.1.2 Garbage Collection (GC) ............................ 8
    2.1.3 Generational Heap .................................. 8
    2.1.4 GC Strategies ........................................ 9
    2.1.5 Load Balancing ...................................... 11
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Generational Heap Example.</td>
<td>9</td>
</tr>
<tr>
<td>2.2</td>
<td>Load Balancing Example.</td>
<td>11</td>
</tr>
<tr>
<td>2.3</td>
<td>Classification of Forecast Methods.</td>
<td>12</td>
</tr>
<tr>
<td>2.4</td>
<td>MAPE-K model</td>
<td>15</td>
</tr>
<tr>
<td>2.5</td>
<td>Traditional Software Development Process.</td>
<td>16</td>
</tr>
<tr>
<td>3.1</td>
<td>PHOEBE - An Automation Framework for Perf. Testing</td>
<td>28</td>
</tr>
<tr>
<td>3.2</td>
<td>PHOEBE - Conceptual View</td>
<td>28</td>
</tr>
<tr>
<td>3.3</td>
<td>PHOEBE - Core Process</td>
<td>30</td>
</tr>
<tr>
<td>3.4</td>
<td>Control Agent - Component Diagram</td>
<td>31</td>
</tr>
<tr>
<td>3.5</td>
<td>Diagnosis Tool Component - Class Diagram</td>
<td>32</td>
</tr>
<tr>
<td>3.6</td>
<td>PHOEBE - Sequence diagram</td>
<td>33</td>
</tr>
<tr>
<td>3.7</td>
<td>Accuracy-Target Data Gathering Policy</td>
<td>34</td>
</tr>
<tr>
<td>3.8</td>
<td>Efficiency-Target Upload Policy</td>
<td>36</td>
</tr>
<tr>
<td>3.9</td>
<td>Multi-View Consolidation Policy</td>
<td>38</td>
</tr>
<tr>
<td>3.10</td>
<td>PHOEBE - Test Environment</td>
<td>44</td>
</tr>
<tr>
<td>3.11</td>
<td>Perf. Bugs vs. Throughput - WAIT.</td>
<td>46</td>
</tr>
<tr>
<td>3.12</td>
<td>Perf. Bugs vs. Throughput - HC.</td>
<td>46</td>
</tr>
<tr>
<td>3.13</td>
<td>Perf. Bugs vs. Throughput - EMAT.</td>
<td>46</td>
</tr>
<tr>
<td>3.14</td>
<td>Perf. Bugs vs. Throughput - GCLITE.</td>
<td>47</td>
</tr>
<tr>
<td>3.15</td>
<td>Perf. Critical Bugs vs. Throughput - WAIT.</td>
<td>48</td>
</tr>
<tr>
<td>3.16</td>
<td>Perf. Critical Bugs vs. Throughput - HC.</td>
<td>48</td>
</tr>
<tr>
<td>3.17</td>
<td>Perf. Critical Bugs vs. Throughput - EMAT.</td>
<td>48</td>
</tr>
<tr>
<td>3.18</td>
<td>Perf. Bugs Distribution - WAIT.</td>
<td>49</td>
</tr>
<tr>
<td>3.19</td>
<td>Perf. Bugs Distribution - HC.</td>
<td>49</td>
</tr>
<tr>
<td>3.20</td>
<td>Perf. Bugs Distribution - EMAT.</td>
<td>49</td>
</tr>
<tr>
<td>3.21</td>
<td>Performance Analysis Efforts per Tool</td>
<td>51</td>
</tr>
<tr>
<td>Figure</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>3.22</td>
<td>Reports per Tool/Node/Hour</td>
<td>53</td>
</tr>
<tr>
<td>3.23</td>
<td>Consolidated Reports per Tool/Node/Hour</td>
<td>53</td>
</tr>
<tr>
<td>3.24</td>
<td>Avg. CPU Utilisations per Tool</td>
<td>55</td>
</tr>
<tr>
<td>3.25</td>
<td>Avg. MEM Utilisations per Tool</td>
<td>55</td>
</tr>
<tr>
<td>3.26</td>
<td>Perf. Bugs vs. Throughput - WAIT</td>
<td>57</td>
</tr>
<tr>
<td>3.27</td>
<td>Perf. Bugs vs. Throughput - HC</td>
<td>57</td>
</tr>
<tr>
<td>3.28</td>
<td>Perf. Bugs vs. Throughput - EMAT</td>
<td>57</td>
</tr>
<tr>
<td>3.29</td>
<td>Critical Perf. Bugs vs. Throughput - WAIT</td>
<td>58</td>
</tr>
<tr>
<td>3.30</td>
<td>Resources vs. Configuration - All Tools</td>
<td>59</td>
</tr>
<tr>
<td>3.31</td>
<td>Performance Analysis Effort-savings</td>
<td>60</td>
</tr>
<tr>
<td>3.32</td>
<td>Performance Testing Time-savings</td>
<td>61</td>
</tr>
<tr>
<td>4.1</td>
<td>TRINI - A GC-Aware LB Strategy</td>
<td>70</td>
</tr>
<tr>
<td>4.2</td>
<td>TRINI - Conceptual View</td>
<td>70</td>
</tr>
<tr>
<td>4.3</td>
<td>TRINI - Core Process</td>
<td>72</td>
</tr>
<tr>
<td>4.4</td>
<td>MaGa - Forecast Process Overview</td>
<td>73</td>
</tr>
<tr>
<td>4.5</td>
<td>Forecast of Young MemAlloc</td>
<td>76</td>
</tr>
<tr>
<td>4.6</td>
<td>Forecast of MaGC Event</td>
<td>76</td>
</tr>
<tr>
<td>4.7</td>
<td>OldGen Memory Usage through time</td>
<td>77</td>
</tr>
<tr>
<td>4.8</td>
<td>Forecast Error per GC Strategy</td>
<td>86</td>
</tr>
<tr>
<td>4.9</td>
<td>pGC Sunflow - MiGC\textsubscript{CV} vs. MiGC\textsubscript{AVG} Trends</td>
<td>88</td>
</tr>
<tr>
<td>4.10</td>
<td>TRINI - Test Environment</td>
<td>90</td>
</tr>
<tr>
<td>4.11</td>
<td>(RT_{AVG}) - Perf. Improv. per LB</td>
<td>93</td>
</tr>
<tr>
<td>4.12</td>
<td>(RT_{MAX}) - Perf. Improv. per LB</td>
<td>93</td>
</tr>
<tr>
<td>4.13</td>
<td>(T_{AVG}) - Perf. Improv. per LB</td>
<td>93</td>
</tr>
<tr>
<td>4.14</td>
<td>(RT_{AVG}) - Perf. Improv. per GC</td>
<td>93</td>
</tr>
<tr>
<td>4.15</td>
<td>(RT_{MAX}) - Perf. Improv. per GC</td>
<td>93</td>
</tr>
<tr>
<td>4.16</td>
<td>(T_{AVG}) - Perf. Improv. per GC</td>
<td>93</td>
</tr>
<tr>
<td>4.17</td>
<td>Forecast Accuracy vs. (MiGC_{CV})</td>
<td>94</td>
</tr>
<tr>
<td>4.18</td>
<td>(RT_{AVG}) - Perf. Improv. per FA and (\text{A-MaGC}_{D})</td>
<td>95</td>
</tr>
<tr>
<td>4.19</td>
<td>(\text{A-MaGC}_{D}) per Forecast Accuracy Level</td>
<td>96</td>
</tr>
<tr>
<td>4.20</td>
<td>App.nodes - (\Delta CPU_{AVG})</td>
<td>97</td>
</tr>
<tr>
<td>4.21</td>
<td>App.nodes - (\Delta MEM_{AVG})</td>
<td>97</td>
</tr>
<tr>
<td>4.22</td>
<td>LB node - (\Delta CPU_{AVG})</td>
<td>97</td>
</tr>
<tr>
<td>4.23</td>
<td>LB node - (\Delta MEM_{AVG})</td>
<td>97</td>
</tr>
<tr>
<td>4.24</td>
<td>(RT_{AVG}) - Perf. Improv. per Cluster Size</td>
<td>100</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

4.25 App.nodes - $\Delta CPU_{AVG}$ per Size. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 101
4.26 App.nodes - $\Delta MEM_{AVG}$ per Size. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 101
4.27 LB node - $\Delta CPU_{AVG}$ per Size. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 101
4.28 LB node - $\Delta MEM_{AVG}$ per Size. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 101
4.29 $RT_{AVG}$ - Perf. Improv. on 24-hr runs. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 103
4.30 $RT_{MAX}$ - Perf. Improv. on 24-hr runs. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 103
4.31 $T_{AVG}$ - Perf. Improv. on 24-hr runs. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 103
List of Tables

2.1 Memory-related JVM Parameters. . . . . . . . . . . . . . . . 8
2.2 Algorithms per GC Strategy. . . . . . . . . . . . . . . . . . 10
2.3 DaCapo Programs. . . . . . . . . . . . . . . . . . . . . . . . 19
2.4 SPECJVM Programs. . . . . . . . . . . . . . . . . . . . . . . 20
3.1 M-WAIT and A-WAIT Comparison. . . . . . . . . . . . . . . 63
4.1 DaCapo Configurations. . . . . . . . . . . . . . . . . . . . . . 85
4.2 DaCapo/SPEC Program Classification per GC Behaviours. . . 98
List of Publications

- Portillo-Dominguez, A., Perry, P., Magoni, D., Wang, M. & Murphy, J. “TRINI: An Adaptive Load Balancing Strategy Based on Garbage Collection for Clustered Java Systems.” *Accepted for publication on the Journal of Software: Practice and Experience*


- Portillo-Dominguez, A., Murphy, J. & Sullivan, P. “Leverage of Extended Information to Enhance the Performance of JEE Systems.” *11th Information Technology and Telecommunications Conference*, 2012; 137-138
List of Patents

- Portillo-Dominguez, A., Mitchell, N., Sweeney, P., Altman, E., Sullivan, P., Assem, H. “Dynamically adapting the behaviour of a diagnosis tool to speed up the identification of performance issues.” *Patent under work by IBM legal team*


Chapter 1

Introduction

The chapter begins by discussing the main motivations behind the research work presented in this thesis. Next, it describes the different contributions of the thesis. Finally, the chapter concludes by presenting an overview of the rest of the thesis.

1.1 Motivation

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 [69] 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 [93]. 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.

A special challenge, documented by multiple authors [50, 107, 137], 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 [125]. 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 [50].

To simplify the performance analysis and diagnosis, many researchers have been developing tools with built-in expertise [46, 47, 50]. 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.

Even if the most critical performance bugs are fixed before releasing an application to a production environment, other factors can also degrade the performance of a clustered application. Among them, another key challenge in cluster computing is how to effectively distribute the workload among
the available clustered instances (as load imbalance can lead to processing inefficiencies [52]). To address this challenge, multiple research efforts have aimed to develop more effective load balancing algorithms and strategies, based on different criteria and heuristics [64] [117] [90].

With an estimated business impact of a hundred billion dollars yearly, Java is a predominant player at enterprise level [81] [41]. Therefore, this technology is commonly used to build clustered systems. A particular area of performance concern in Java is the Garbage Collection (GC) [127]. Even though it is a key feature of the Java language which automates most of the tasks related to memory management, GC also comes with a cost: Whenever it is triggered, GC has an impact on the system performance by pausing the involved programs. Although pauses of milliseconds are normally not a problem, longer GC pauses can severely impact the system performance, affecting the involved business functions and the overall user experience. This is particularly true for applications requiring fast response time or high throughput. Furthermore, this issue is more likely to occur with the Major Garbage Collection (MaGC), which usually causes the longest type of GC pauses [127].

Multiple research works have given evidence of the GC performance costs. For instance, the authors of [141] identified the GC as a major factor degrading the behaviour of Java Application Servers (a classic Java business niche) due to the sensitivity of the GC to the workload. In their experiments, the GC took up to 50% of the total Java Virtual Machine (JVM) execution time (involving pauses as high as 300 seconds). The MaGC represented more than 95% of those pauses on the heaviest workload. Likewise, a survey conducted among Java practitioners and experts [123] identified the GC as a typical area of performance problems experienced in the industry.

Research studies have also shown that it is not possible to have a single “best-fit-for-all” GC strategy because the GC behaviour is dependent on the application inputs and the system configuration [84] [101] [94] [48] [124]. For example, the authors of [57] showed that the GC is particularly sensitive to the heap size and even small changes, which might appear trivial, could affect its behaviour. Due to the multiple factors (e.g., increases in workload, usage of huge heaps or non-ideal settings) which can provoke long MaGC pauses (probably of hundreds of milliseconds or longer), it is commonly agreed that the GC plays an important role in the performance of Java systems.
Finally, to ensure that this 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 performance optimisation of Java clusters, especially in large-scale environments.

1.2 Thesis Contributions

The above discussion motivates the core research question here:

\[
\text{What techniques can be developed to automatically conduct performance engineering in a cluster for improving its performance while avoiding errors and saving time?}
\]

To answer this question, this thesis contributes to the area of performance optimisation of clustered systems in Java (a predominant technology at enterprise-level), especially aiming for large-scale environments. The following paragraphs describe the highlights of the thesis contributions.

The first major contribution of this thesis is a novel policy-based adaptive automation framework (PHOEBE) that addresses the common usage limitations experienced by a diagnosis tool to be effectively used in the performance testing of clustered applications. The framework executes concurrently with a performance test, fully 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. The aim is to improve a tester’s productivity by decreasing the effort and expertise needed to use a diagnosis tool. Internally, the framework leverages on a set of policies to automatically control the different manual processes commonly involved in the configuration and usage of a performance diagnosis tool.

Complementary to the above contribution, the following supporting contributions are also discussed in this thesis:
CHAPTER 1. INTRODUCTION

• 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.

• A practical validation of PHOEBE and its policies, consisting of a prototype implementation around a set of well-known performance Java diagnosis tools and a set of experiments. They demonstrate the accuracy of the framework as well as its productivity benefits. Furthermore, based on the obtained results, a characterisation of the diagnosis tools used in the experimental evaluation is presented. It can be used as a framework of reference for practitioners to know the conditions under which each policy can be more suitable.

The second major contribution of this work is a novel GC-aware adaptive load balancing strategy (TRINI). This enhanced load balancer selects the application nodes which are not expected to have a MaGC event in the immediate future as optimal nodes for the incoming workload. This strategy can therefore help to avoid impacts in the cluster’s performance due to the occurrence of MaGC events in the individual nodes. Internally, TRINI uses a MaGC forecasting algorithm to decide on the best way to balance the workload among the application nodes.

Complementary to the above contribution, the following supporting contributions are also discussed in this thesis:

• A novel forecast algorithm that enables Java systems to predict when a MaGC event will occur in a Java Virtual Machine. Additionally, four GC-aware load balancing algorithms which have been modified to use MaGC forecast information for improving their workload distribution.

• A set of GC-oriented metrics to characterise the behaviour of Java applications in terms of Garbage Collections. Furthermore, based on their GC characteristics, a classification of the programs within the evaluated Java benchmarks is presented. Likewise, three GC-based program families, applicable to a broad range of application memory
behaviours, are presented. They are based on the different levels of variability in their Minor Garbage Collection behaviour.

- A comprehensive practical evaluation of TRINI, consisting of a prototype and a set of experiments to assess TRINI in terms of generality, reliability and scalability. These experiments demonstrate the performance benefits and overhead costs of using TRINI under those scenarios. Furthermore, based on the obtained results, key findings that could serve as guidelines for practitioners to integrate GC-awareness to a load balancing algorithm, as well as the conditions under which a GC-aware load balancing strategy can be useful, are presented.

1.3 Thesis Overview

The remainder of the thesis is structured as follows:

- Chapter 2 presents a summary of the pertinent state of the art, in terms of background information and related work.

- Chapter 3 presents the proposed automation framework, PHOEBE, as well as the developed policies and the implemented prototype. The chapter also discusses the performed experiments and their results, including the assessments performed to identify suitable policies.

- Chapter 4 explains the internal workings of the proposed load balancing strategy, TRINI, as well as the developed algorithms, policies and prototype. The chapter also discusses the performed experiments and their results.

- Chapter 5 presents the conclusions of this research work and provides pointers to future work in this area.
Chapter 2

State of the Art

This chapter presents the relevant state of the art in terms of the applicable background information and related work. The following two sections describe them.

2.1 Background

This section presents the main features and characteristics of Java, autonomic computing, as well as a typical load balancing and performance testing process, which are necessary to understand the rest of the thesis. The section also describes the forecast techniques, performance diagnosis tools and Java benchmarks used in this work.

2.1.1 Java Virtual Machine (JVM)

It is the run-time environment on which applications developed in Java can run. This is because a JVM can interpret the binary format (i.e. bytecode) in which a program developed in Java is compiled. As there are JVMs available for most contemporary operating systems, a compiled Java program is highly portable. Moreover, a JVM can be tuned at start-up time through multiple configuration parameters, including several ones related to memory management. Table 2.1 describes some of the most commonly used memory options.
Table 2.1: Memory-related JVM Parameters.

<table>
<thead>
<tr>
<th>JVM Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-XX:+DisableExplicitGC</td>
<td>Disable explicit requests to execute GCs</td>
</tr>
<tr>
<td>-XX:PermSize</td>
<td>Initialise the size of the PermGen</td>
</tr>
<tr>
<td>-XX:MaxPermSize</td>
<td>Define the max size of the PermGen</td>
</tr>
<tr>
<td>-Xms</td>
<td>Initialise the size of the Heap</td>
</tr>
<tr>
<td>-Xmx</td>
<td>Define the max size of the Heap</td>
</tr>
</tbody>
</table>

2.1.2 Garbage Collection (GC)

It is a core feature of Java. This form of automatic memory management offers significant software engineering benefits over the explicit memory management. For example, it frees developers from the burden of manual memory management, avoiding the most common sources of memory overwrites and leaks [136], as well as increasing the developers’ productivity [109]. Despite these advantages, it is widely accepted that the GC comes with a performance cost (as discussed in Section 1). Efforts such as the creation of the Real-Time Specification for Java (RTSJ) [39], which allows programs to fully skip the GC to offer real-time capabilities (at the cost of requiring significant changes in the source code and the loss of the benefits of the automatic memory management), can also be seen as qualitative evidence of this situation.

Additionally, it is not possible to programmatically force the execution of the GC [109]. The closest action a developer can do is to call the method Runtime.getRuntime().gc() (or its equivalent method System.gc()) to suggest the JVM to execute a MaGC. Nevertheless, the JVM is not forced to fulfil this request and may choose to ignore it. The usage of these methods is also discouraged by the JVM vendors [13] because the JVM usually does a better job on deciding when to do GC. Not surprisingly, this is one of the main motivations behind the usage of the automatic memory management.

2.1.3 Generational Heap

The memory area in Java is known as the heap. Nowadays, one of the most commonly used heap types is the generational heap [35], where the objects are segregated by age into memory regions (commonly limited to two or three) known as generations. New objects are created in the youngest
CHAPTER 2. STATE OF THE ART

generation because the survival rates of younger generations are usually lower than those of older generations. That is, younger generations are more likely to contain garbage and can be collected more frequently than older ones. The GC in the younger generations is known as Minor GC (MiGC). It is usually inexpensive and rarely causes a performance concern. MiGC is also in charge of moving the live objects, which have become old enough, to the older generations. This means that the MiGC plays a key role in the memory allocation of older generations. The GC in the older generations is known as MaGC and it is commonly accepted as the most expensive GC type due to its performance impact [127]. Finally, running out of free memory in a generation triggers its respective type of GC event.

An example of this type of heap is depicted in Figure 2.1, which shows the generational heap of the Oracle JVM HotSpot 7 [31]. In this heap, the young generation (YoungGen) is comprised of 3 sub-areas: One eden, where objects are created, and two survivors, where the MiGCs iteratively place the live objects until they become old enough to be moved to the old generation (OldGen). Finally, the permanent generation (PermGen) is a special region where the JVM stores its own data, such as the definitions of the available classes. It triggers a MaGC when its memory is exhausted.

![Figure 2.1: Generational Heap Example.](image)

2.1.4 GC Strategies

The heap is managed by the GC strategy selected at JVM start-up. Their availability is usually tied to the heap type. For instance, three of the most widely-used GC strategies in the industry [30] work exclusively on generational heaps: The Serial GC (which performs all its work using a single thread and it is preferable for client JVMs), the Parallel GC (which uses multiple threads and it is preferable for server JVMs when the throughput is more important than the response time), and the Concurrent GC (which
does most of its work concurrently with the application threads and it is preferable for server JVMs when the response time is more important than the throughput).

Each GC strategy involves two algorithms, one applicable to the Young Generation and another to the Old Generation. Table 2.2 summarises them. Furthermore, they are described in the following paragraphs.

Table 2.2: Algorithms per GC Strategy.

<table>
<thead>
<tr>
<th>GC Strategy</th>
<th>YoungGen Algorithm</th>
<th>OldGen Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial</td>
<td>Copy</td>
<td>Mark Sweep Compact</td>
</tr>
<tr>
<td>Parallel</td>
<td>PS Scavenge</td>
<td>PS Mark Sweep</td>
</tr>
<tr>
<td>Concurrent</td>
<td>PS Scavenge</td>
<td>Concurrent Mark Sweep</td>
</tr>
</tbody>
</table>

**Serial GC algorithms.** Serial GC uses the *Copy* algorithm for the YoungGen and the *Mark Sweep Compact* for the OldGen [127]. The *Copy* algorithm works by reviewing the age of the live objects (in terms of how many young generation collections they have survived). Then, those objects which are “old enough” are copied to the OldGen, while the remaining objects are copied to the unused survivor space. If this space becomes full, the live objects which exceeded its capacity are also copied to the OldGen, regardless of their age. Meanwhile, the *Mark Sweep Compact* algorithm is composed of three phases: In the mark phase, the live objects are identified. Then, in the sweep phase the garbage is identified. Next the algorithm slides the live objects towards the beginning of the OldGen space, leaving the free space in a single contiguous chunk at the opposite end (so that it is easily available for future allocations).

**Parallel GC algorithms.** Parallel GC uses the *PS Scavenge* algorithm for the YoungGen and the *PS Mark Sweep* for the OldGen [127]. These algorithms are the parallel versions of the algorithms used by the serial GC. They are still stop-the-world algorithms, but they perform most of their operations in parallel. This is done by leveraging the presence of multiple CPUs.

**Concurrent GC algorithms.** Concurrent GC uses the *PS Scavenge* algorithm in the YoungGen (the same algorithm used by the parallel GC) and the *Concurrent Mark Sweep* for the OldGen [127]. The *Concurrent Mark Sweep* algorithm does most of its work concurrently with the execution
of the application. It starts with a short pause (phase known as initial mark) to identify the set of live objects directly reachable from the application code. Then, all live objects that are transitively reachable from this set are also marked (phase known as the concurrent marking). Then, a second pause occurs (phase known as remark) to finalise the marking by revisiting any objects that were modified by the application during the concurrent marking phase. This step is needed to guarantee that all live objects have been correctly marked (as the application was running during the concurrent marking phase and some live objects might have been missed). The final phase of the algorithm is the concurrent sweep which reclaims all the garbage that has been identified.

2.1.5 Load Balancing

The objective of a load balancing strategy is to optimise the performance of an application running in a cluster composed of a set of application nodes (typically located within a data centre), each one having an identical code image of the application. This scenario is depicted in Figure 2.2. The clustered application must be also partitionable into smaller grain-sized tasks (e.g., a traditional web application, which is normally composed of atomic operations such as login, logout, search, etc.).

![Figure 2.2: Load Balancing Example.](image)

The range of existing load balancing algorithms is broad [56, 85]. Nowadays, four algorithms frequently used in the industry are: round robin,
random, weighted round-robin and weighted random. Round robin selects the nodes iteratively, eventually distributing the workload evenly across the available nodes. In the case of the random, each node is selected at random among the available ones. Finally, in the weighted versions of these algorithms, the number of times a node would be selected (as per their respective decision logic) is adjusted using a weight defined per node.

2.1.6 Forecasting Techniques

The objective of a forecasting technique is to make predictions of a future event based on the available data and the analysis of its trends. Furthermore, the factors that relate to what is being forecast should be known and well understood to achieve a reliable forecast. The range of existing forecasting techniques is broad \[99, 73\]. They are normally classified into two main categories: Qualitative and quantitative techniques (as shown in Figure 2.3).

![Figure 2.3: Classification of Forecast Methods.](image)

The qualitative techniques are subjective, based on the judgment of some type of subject matter experts. These techniques are appropriate when past data is not available. An example of a qualitative technique is the Delphi method \[80\], which uses a combination of expert judgment and historical information to produce individual estimations (per expert), which are later compared among the panel of participants. This process is iteratively repeated until a consensus is reached.

The quantitative forecasting techniques are used to predict future data as a function of past data. This is achieved by formally capturing the relationships between the involved variables in the form of mathematical equations. These techniques are appropriate when past numerical data is available and
when it is reasonable to assume that some of the patterns in the data are expected to occur in the future. The three quantitative techniques more frequently used are moving average, exponential smoothing and regression. They are described in the following paragraphs.

**Moving Average** [108]. This technique involves the calculation of the mean value of a time series (observations equally spaced in time) from several consecutive periods. It is called “moving” because it is continually recomputed as new data becomes available. It progresses by dropping the earliest value and adding the latest value.

Three of its most commonly-used variations are: (1) The *simple moving average*, in which the value for a given time period is replaced by the mean of that value based on the values for some number of preceding time periods; (2) the *naive moving average*, which is a special case of the simple moving average in which the number of periods used for smoothing is one; and (3) the *weighted moving average*, in which the value for a given time period is replaced by the weighted mean of that value based on the values for some number of preceding time periods.

**Exponential Smoothing** [108]. This technique can detect significant changes in data by ignoring the fluctuations irrelevant to the purpose at hand. In contrast to the moving average, the older data is given progressively-less relative weight (i.e. importance) whereas newer data is given progressively-greater weight.

Three of its most commonly-used variations are: (1) The *simple exponential smoothing* [108] weights past observations with exponentially decreasing weights to forecast future values; (2) the *double exponential smoothing* [108] (also known as the Holt method) is a refinement of the simple exponential smoothing technique which adds another component to take into account any trends in the data; and (3) the *triple exponential smoothing* [108] (also known as the Winters method) is a refinement of the double exponential smoothing technique which adds another component to take into account any seasonality (or periodicity) in the data.

**Regression** [108]. This technique helps to estimate the value of a dependent variable based on the available historical data of a set of independent variables. This is done by developing a regression model which is composed of the independent variables, the dependent variable and any unknown (constant) parameters.
Three of its most commonly-used variations are: (1) The *simple linear regression* involves a single independent variable. This property allows the model to fit a straight line through the set of available data in such a way that makes the sum of squared residuals of the model as small as possible; (2) the *multiple linear regression*, which is an extension of the single linear regression, involves multiple and/or vector-valued independent variables. This property allows the estimation to be influenced by multiple types of historical data; and (3) the *polynomial regression* is a form of regression in which the relationship between the independent variable and the dependent variable is modelled as an nth degree polynomial. This property allows the model to fit a non-linear relationship between the variables.

### 2.1.7 Autonomic Computing

Autonomic computing [77] is a self-managing computing model which defines the set of characteristics that a computer system requires in order to be able to adapt to unpredictable changes in its execution environment. Its goal is to create systems that can run by themselves. This means, not only capable of high-level functioning, but also keeping the system’s complexity invisible to operators and users. The aim is that this type of systems can help overcoming the rapidly growing complexity of computing systems management, as well as reducing the barrier that complexity poses to further systems’ growth [90]. The model defines four basic properties that an adaptive system should have: (1) *Self-configuring* in order to adapt to dynamically changing environments (hence increasing its responsiveness); (2) *self-healing* in order to discover, diagnose, and act to prevent disruptions (to achieve business resiliency); (3) *self-optimising* in order to tune resources and balance workloads to maximise the use of resources (for operational efficiency); and (4) *self-protecting* in order to anticipate, identify, and protect against attacks (hence securing the information and resources).

From an architecture perspective, control loops [122] have been extensively studied and have been identified as a key mechanism for achieving self-adaptation in software systems [61]. Based on them, many solutions to build self-* systems have been proposed. For instance, the *reactive adaptation* pattern [114] describes a mechanism to build reactive components which are capable of modifying their behavior in reaction to an external event, without the need of any control loop. Alternatively, the *internal feed-
back loop pattern [113] proposes a mechanism to enrich the standard logic of a system by implementing a control loop which monitors the context of its execution, determines the changes to be enforced, and enacts them. Finally, the most widely-used adaptive pattern is MAPE-K [82]. It is based on the feedback control system used in the control theory field. MAPE-K is composed of the five elements depicted in Figure 2.4: A Monitoring element to obtain information from the managed systems (through system indicators); an Analysis element to evaluate if any adaptation is required; an element to Plan the adaptation; an element to Execute it (through adaptations); and a Knowledge element to support the others in their respective tasks. The main advantage of MAPE-K (compared to the internal feedback loop) is that its external control loop is neatly separated from the application logic.

![MAPE-K model](image)

Figure 2.4: MAPE-K model.

### 2.1.8 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 [86]. 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 2.5). Meanwhile, in an agile software development process, performance testing is normally performed at the end of each iteration (or group of iterations).

![Figure 2.5: Traditional Software Development Process.](image)

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 [26], Apache JMeter [2] or HP LoadRunner [22]) 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 [43]: 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 thesis.

2.1.9 Performance Diagnosis Tools

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)** [16]. It is a diagnosis tool that helps to detect memory-related performance issues. EMAT is based on non-intrusive sampling mechanisms available at the JVM, in the form of heap-dumps [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)** [50]. It is a diagnosis tool that helps to detect GC performance issues. GCLITE is based on the tracing mechanisms available at the JVM, in the form of the GC verbose [18] (complete GC logs containing information such as the total size of the heap, the size of the generations 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) \[24\]. 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 \[18\]. 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) \[25\]. 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 \[46\]. WAIT is a diagnosis tool that implements this methodology and has proven to simplify the detection of performance issues in Java systems \[46, 138\]. 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 \[9\] (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.

2.1.10 Java Benchmarks

Nowadays, DaCapo and SPECJVM are two of the Java benchmarks most widely-used in the literature. The following paragraphs briefly describe the versions of these benchmarks which were used in the experimental evaluation of this research work.
DaCapo 9.12. 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 [40]. Table 2.3 lists these programs and briefly describes their functionality.

Table 2.3: DaCapo Programs.

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

SPECJVM 2008. This benchmark has been developed by the Standard Performance Evaluation Corp (SPEC), and companies such as HP, IBM and Sun have contributed to it. The benchmark is composed of 10 different programs. They are a mixture of real-life applications and specialised
benchmarks focused on the core java functionality [38]. Table 2.4 lists these programs and briefly describes their functionality.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>compiler</td>
<td>It compiles a set of java source files.</td>
</tr>
<tr>
<td>compress</td>
<td>It uses an universal loss-less data compression algorithm.</td>
</tr>
<tr>
<td>crypto</td>
<td>It encrypts and decrypts a set of files using a set of encryption protocols.</td>
</tr>
<tr>
<td>derby</td>
<td>An open-source database written in pure Java.</td>
</tr>
<tr>
<td>MPEGaudio</td>
<td>It uses a mp3 decoder in a set of audio files.</td>
</tr>
<tr>
<td>scimark</td>
<td>It executes a set of floating point operations.</td>
</tr>
<tr>
<td>serial</td>
<td>It serialises and deserialises a set of objects and primitives.</td>
</tr>
<tr>
<td>startup</td>
<td>It executes each other program one time.</td>
</tr>
<tr>
<td>sunflow</td>
<td>It executes a set of graphics visualization operations.</td>
</tr>
<tr>
<td>XML</td>
<td>It transforms and validates a set of XML documents by applying a set of style sheets.</td>
</tr>
</tbody>
</table>

2.2 Related Work

In this section, I first expand on the recent work in GC optimisation and memory forecasting. Then, I review the state-of-the art work in distributed systems optimisation, with a special emphasis on load balancing. Finally, I discuss the related work in the areas of performance monitoring and management, automation in testing and performance analysis.

2.2.1 GC Optimisation

Multiple research efforts have focused on improving the GC performance. For example, several works have proposed new concurrent [110, 132] and parallel algorithms [53, 120] that have smaller impacts on the performance of the applications. Other works have aimed to develop algorithms that might have predictable GC performance [88, 68]. However this predictability comes in terms of soft-requirements, meaning that the GC might still take
more time than originally expected. The applicability of these approaches is also limited because they require non-standard JVMs, which also usually involve higher license costs.

Another explored approach has been to develop algorithms for specific usage scenarios. For instance, [139] describes an algorithm suitable to Java Application Servers which exploits the different natures of the local and remote objects. Similarly, [70] presents the garbage first algorithm which splits the heap into multiple regions in order to only reclaim memory from those regions mostly populated by garbage. The work on [148] presents a GC suitable for multi-core processors, which balances data locality concerns with heap utilisation and fragmentation concerns to achieve good performance by maintaining the abstraction of a single large heap regardless of the number of cores. Even though all these works have helped to reduce the frequency and impact of the GC, it remains a major performance concern due to the diverse factors that can affect its performance (as discussed in Section 1).

Finally, other research has focused on addressing the usability limitations of the Real-Time Specification for Java (RTSJ), which allows programmers to side-step the GC altogether, at the cost of requiring significant changes in the source code and losing the benefits of the automatic memory management. For example, the work described in [126] focuses on automating the code changes required to use RTSJ, while the authors of [58] present an analysis of the design and programming challenges of working with RTSJ. As structural changes to the source code are still required to use RTSJ, this limitation remains as a major adoption roadblock for this technology.

2.2.2 Memory Forecasting

Memory forecast, which is closely related to the GC forecast, is another active research area in the GC community which focuses on the self-improvement of the JVM, looking for ways to invoke a GC only when it is worthwhile. For example, the work presented in [133] exploits the observation that dead objects tend to group together to estimate how much space would be reclaimable for a MaGC in order to avoid low-yield GCs. Meanwhile, the approach taken in [62] aims to maximise the GC benefits by profiling the code off-line to identify favorable collection points which are later used during the program execution to trigger a MaGC (or a MiGC) when the ratio of live objects is low. Finally the authors of [140] present an approach
to estimate the number of dead objects at any time, information that a JVM could use to decide when to trigger a MaGC.

In all these cases, the memory forecasts help to determine if it is a good time (in terms of potential memory gains) to execute a GC. However, these memory forecasts alone do not provide enough information to know when the next MaGC would occur. In contrast, my work aims to forecast the occurrence of the MaGC events, also making this information available outside the JVM so that other actors (e.g., a load balancer) could leverage it and take more informed decisions.

2.2.3 Distributed Systems Optimisation

Research has also focused on the optimisation of distributed architectures, improving them from various viewpoints. For example, the authors of [103] presented a method to facilitate the migration of a monolithic Java application to a distributed architecture through the automated dependency injection of source code. In the case of [44], this work described a mechanism to achieve high reliability in clustered web services, which was based on its capability of offering transparent fault-tolerance to different types of transactions. Furthermore, the work on [89] proposed a resource management solution for distributed systems, offering capabilities such as the automatic detection of overloaded resources.

Due to its importance, load balancing is a well-studied problem in the areas of parallel and distributed systems, where a significant body of literature exists [92, 71, 128, 51, 69, 130, 60, 79, 75, 119, 65]. For example, the authors of [117] proposed a technique to estimate the total workload of a load balancer to utilise this information in the balancing of new workload. Meanwhile, the work on [59] proposed an adaptive load balancing strategy which aims to fulfil service level agreements (SLAs) based on a set of customer priorities. Likewise, the authors of [104] presented an agent-based solution to provide dynamic load balancing capabilities to cloud-based services and resources. Meanwhile, the authors of [60] presented a load balancing scheme which considers the statistics of past executions to distribute the new workload. Finally, other research efforts have focused on Java technologies: The authors of [64] developed a load balancing algorithm for the static and dynamic layers of Java web applications which considers the utilisation of the
JVM heap, threads and CPU to decide how to distribute the load. Similarly, the work presented in [96] proposes a function to calculate the utilisation of an Enterprise JavaBean (EJB) and then uses this information to distribute the incoming load among the available EJB instances.

In contrast to all the previously discussed works, my research work has centred on enhancing a load balancer by considering the MaGC forecasts in its decision layer. In such a case, the load balancer can get additional knowledge about the JVM in order to control the workload of the system, in addition to other existing load balancing policies that might be applicable.

2.2.4 Performance Monitoring and Management

Research has also centred on enhancing performance monitoring and management of clustered systems, improving these processes from various perspectives. For example, the authors of [72] proposed the MonALISA framework, which provides capabilities for monitoring, managing, controlling and optimising a distributed system. MonALISA relies on an agent-based architecture in which a set of agents, using real-time monitoring information, collaborate to perform the supported tasks. Meanwhile, the work on [111] described NetHAM-nano, which is a platform for distributed monitoring based on a cloud architecture. By treating the monitoring needs as services, NetHAM-nano leverages on a publish-subscribe paradigm in order to allow high flexibility of resource management (as the monitoring jobs can be assigned to any nodes with available capacity). Likewise, the authors of [102] presented Ganglia, which is a distributed monitoring system especially tailored for high-performance systems. Ganglia relies on a multicast listen/announce protocol to monitor the states of each cluster as well as a hierarchical structure (in the form of federations of clusters) to allow different levels of abstraction in the analysis and presentation of its results.

Moreover, the work on [131] introduced Astrolabe, which is a distributed information management system which is able to collect large-scale volume of systems’ states and aggregate the information on-the-fly. Based on its capabilities, Astrolabe can offer a scalable way to track the system state through time and perform self-corrective activities based on this information (e.g., locate resources or perform a distributed synchronizations within a large system). Similarly, the authors of [106] proposed CoMon, which is
a scalable monitoring system for PlanetLab. Through a hybrid strategy of passively/actively gathering a set of useful metrics, CoMon is able to facilitate that administrators and users monitor and debug their experiments as well as identify problems (e.g., problematic machines) within the PlanetLab global testbed. Finally, the work on [49] introduced GridICE, which is a monitoring service especially tailored for grid systems. It offers the ability to deal with the different dimensions commonly required in a grid system (e.g., virtual organisations and sites) as well as an easy integration (through a modular architecture) with the middleware commonly used for monitoring a typical grid system.

Unlike the previously discussed works, which have been designed to assist on the management of a clustered system, my proposed framework has been designed to address the specific needs of a tester in the performance testing. It does not only isolate a tester from the complexities of configuring a performance diagnosis tool, but also enables the effective utilisation of such tools in the performance testing domain.

2.2.5 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 [45, 67, 83, 91, 97, 129, 143]. For example, the authors in [97] propose an approach to automate the generation of test cases based on specified levels of load and combinations of resources. Similarly, [67] presents an automation framework that separates the application logic from the performance testing scripts to increase the re-usability of the test scripts. Meanwhile, [143] 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.

Other research efforts have concentrated on automating specific analysis techniques. For example, [140] presents a combination of coverage analysis and debugging techniques to automatically isolate failure-inducing changes. Similarly, the authors of [95] developed a technique to reduce the number of false memory leak warnings generated by static analysis techniques by automatically validating and categorizing those warnings.

Finally, other researchers have proposed frameworks to support different software engineering processes. For example, the authors of [53] [76] present
frameworks to monitor software services. Both frameworks monitor the resource utilisation and the component interactions within a system. One focuses on Java [76] and the other focuses on Microsoft technologies [54]. Unlike these works, which have been designed to assist on operational support activities, my 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.

2.2.6 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 [142] 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 [147] 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 [55] analysed the memory heaps of several real-world object-oriented programs and provided insights to improve memory allocation and program analysis techniques.

A high percentage of the proposed performance analysis techniques require some type of instrumentation. For example, the authors in [145] 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 [63, 118] which rely on instrumentation to dynamically infer invariants within the applications and detect programming errors; or the approach proposed by [66] 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, my proposed framework does not require instrumenting the tested applications.

Furthermore, the authors of [57] 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 [74] 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, my 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 [87, 74] require information from previous runs to baseline their analysis, information which might not always be available.
Chapter 3

PHOEBE: An Automation Framework for Perf. Testing

This chapter presents PHOEBE. First, the context of the solution is provided. Next, the internal workings of PHOEBE are described, including the proposed policies. The chapter concludes with the performed experimental evaluation and the discussion of the obtained results.

3.1 Overview

The objective of this 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. PHOEBE is shown in Figure 3.1. 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.

A self-adaptive system is normally composed of a managed system and an autonomic manager [98]. 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 3.2 which shows the conceptual view of PHOEBE.
As defined by multiple authors [105, 134], 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, I have followed the well-known MAPE-K adaptive model [82] (discussed in Section 2.1.7). This model was chosen because it allows to neatly decouple the adaptive layer from the business logic, hence increasing the modularity of the solution.

Figure 3.1: PHOEBE - An Automation Framework for Perf. Testing.

Figure 3.2: PHOEBE - Conceptual View.
CHAPTER 3. PHOEBE

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 3.2), 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.
3.2 Core Process

From a process perspective, PHOEBE has a core process which coordinates the MAPE-K elements. The process is depicted in Figure 3.3. 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 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.

### 3.3 Architecture

PHOEBE is implemented with the multi-agent architecture depicted in Figure 3.1. 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).

Internally, each agent is comprised of different components. This is exemplified in Figure 3.4 which presents the component diagram 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 3.5, 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 [17].

![Diagram of Diagnosis Tool Component]

Figure 3.5: Diagnosis Tool Component - Class Diagram.

Finally, the agents communicate through commands, following the `Command` design pattern [7]: 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 3.6. 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. 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).

3.4 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 3.6.1, 3.6.2 and 3.6.3).
3.4.1 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 3.7, 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, 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 $\Delta SI$, 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 $\Delta SI$. Finally, the new SI is propagated to all the application nodes, which start using it since their next data gathering iteration.

Finally, it is worth mentioning that this policy was inspired 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.

### 3.4.2 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 3.8, where each step is mapped to the corresponding MAPE-K element. It requires two user inputs: The initial UI to be used; and the $\Delta UI$, 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 [115], 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 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 $\Delta UI$), 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 [37] of the tool.

3.4.3 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 3.9. 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).
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 \[^{[116]}\]) 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 \[^{[3.3]}\]

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.
3.5 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 3.6.1, 3.6.2 and 3.6.3).

3.5.1 Similarity Assessments

As discussed in Section 3.4.3, 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, I 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 (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.

3.5.2 Severity Styles

As discussed in Section 3.4.3, 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 3.5.3).

Among the alternative strategies to develop severity styles for PHOEBE, I 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) \[28\]: Critical, major, moderate, minor, and cosmetic \[10\].

- 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.

### 3.5.3 Go No-Go Assessments

As discussed in Section 3.4.3, a Go No-Go assessment offers an alternative exit condition to a 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) \[116\] 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 performance 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.

3.6 Experimental Evaluation

This section presents the experiments performed to evaluate PHOEBE. To understand which policies would work best, I started by performing an assessment of the identified trade-offs. It involved three experiments: Firstly, I evaluated the accuracy of each diagnosis tool with respect to the overhead introduced by the data sampling processes that feed the tool. Secondly, I 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, I 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 3.4), two additional experiments were
done to evaluate the benefits and costs of using PHOEBE: First, I assessed
the accuracy of the implemented policies (i.e., how well they addressed the
previously identified trade-offs). Then, I assessed the productivity gains that
PHOEBE brings to the performance testing process. The section concludes
with a discussion for practitioners where I summarise the key findings and
observations.

3.6.1 Experiment #1: Accuracy Trade-off Assessment

Here, the objective was to evaluate the potential trade-off between the accu-
racy 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.

3.6.1.1 Experimental Set-up

In the following paragraphs I present the developed prototype, the test en-
vironment and the parameters that defined the evaluated experimental con-
figurations: The selected diagnosis tools, Java benchmarks, sample intervals
(SI) and upload intervals (UI). I also describe the evaluation criteria used
in this experiment.

Prototype. A prototype has been developed in conjunction with our in-
dustrial partner. The Control Agent was implemented on top of the Apache
JMeter 2.9, which is a leading open source tool used for application per-
formance 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 (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 3.10). 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 [2] (a leading open source tool used for application performance testing), and the application nodes ran an Apache Tomcat 6.0.35 [3] (a popular open source Web Application Server for Java).

![Figure 3.10: PHOEBE - Test Environment.](image)

The VMs were located on a Dell PowerEdge T420 server [12] 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 [32] for virtualisation.

**Diagnosis Tools.** The five performance diagnosis tools discussed in Section 2.1.9 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.** The DaCapo benchmark 9.12 [40] was chosen as application set because it offers a wide range of different application behaviours to test. 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. 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 [112], 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.

### 3.6.1.2 Experimental Results

In this section I 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 3.11, 3.12 and 3.13 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 3.11), the generation of a Javacore (which is the main input used by WAIT, as explained in Section 2.1.9) involves tem-
porarily pausing the execution of the application processes running within
the JVM [20]. 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 moni-
tored 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 [19].

For the trace-based tools (i.e. GCLITE and GCMV), the obtained re-
sults 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 3.11: Perf. Bugs vs. Throughput - WAIT.
Figure 3.12: Perf. Bugs vs. Throughput - HC.
Figure 3.13: Perf. Bugs vs. Throughput - EMAT.
Figure 3.14 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 2.1.9) 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 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 3.15, 3.16, and 3.17 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 tended to report potential performance issues even when their frequency was very low. This is 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 3.18, 3.19, and 3.20 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 considerably larger than the critical ones, and the number of new identified bugs tends to decrease during the execution of a test run.

3.6.2 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.
3.6.2.1 Experimental Set-up

This assessment re-used some of the outputs produced by experiment #1 (discussed in Section 3.6.1). 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.

3.6.2.2 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 me 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 3.6.1.1), 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 3.4.3.

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 3.21, 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.
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). Similarly, 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 3.21.

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 3.6.5).

- 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 3.6.1.2), 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 3.5.3), 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, I 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.

This scenario is exemplified in Figures 3.22 and 3.23. 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 3.21). Figure 3.22 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 3.23 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).

Figure 3.22: Reports per Tool/Node/Hour. Figure 3.23: Consolidated Reports per Tool/Hour.

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.
3.6.3 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.

3.6.3.1 Experimental Set-up

The set-up was similar to that used in the experiment #1 (presented in Section 3.6.1.1), 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 (%) utilizations in the diagnosis tool node, during the processing of the samples. These metrics were collected using the “top” command.

3.6.3.2 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 considerably less memory-intensive (with its highest $MEM_{AVG}$ below 5% utilisation in all the tested configurations). The obtained results are presented in Figures 3.24 (CPU) and 3.25 (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\textsubscript{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 \cite{115}), 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.

### 3.6.4 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.

#### 3.6.4.1 Experimental Set-up

The set-up was similar to that used in the experiment #1 (presented in Section 3.6.1.1), 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 [115]). Regarding the consolidation policy, the two-level severity style (described in 3.5.2) 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 3.5.3) 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.

3.6.4.2 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 3.6.1, 3.6.2 and 3.6.3).

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
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 3.26, 3.28 and 3.27 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 3.29 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 during their execution. These behaviours are shown in Figure 3.30, 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.

Figure 3.29: Critical Perf. Bugs vs. Throughput - WAIT.
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 3.31, which shows the efforts required by the testers (per diagnosis tool) for four different experi-
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 3.5.3) 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 3.32 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.

3.6.5 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.

3.6.5.1 Experimental Set-up

The experimental set-up was similar to that used in the experiment #4 (presented in Section 3.6.4.1), with the following differences: First, the iBatis JPetStore 4.0 [23] 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 number of application nodes was increased (to 10 nodes) to test PHOEBE in a bigger test environment. This experiment also involved modifying the source code of JPetStore 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, I 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 me to define an improvement baseline (as the improvement in the other diagnosis tools would be higher). Finally, no GoNoGo assessment was configured in order to test PHOEBE in a reliability-type performance test run.

3.6.5.2 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 3.1.
Table 3.1: M-WAIT and A-WAIT Comparison.

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

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 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 3.6.2.2) 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.

3.6.6 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, I 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 2.1.8, 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 3.5.3) 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 3.4.1) 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 3.4.2) 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, I 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 3.5.2).

- 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) 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 3.4). This strategy can then be leveraged to make a more robust framework.

3.7 Summary

This chapter presented PHOEBE, a novel adaptive framework that automates the configuration and overall usage of a diagnosis tool in a clustered testing environment. The aim was to improve a tester’s productivity by decreasing the effort and expertise needed to use diagnosis tools. A detailed discussion of the different components and policies, which complement PHOEBE’s capabilities, was also presented. Additionally, a comprehensive experimental evaluation of PHOEBE was discussed. First, 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. The obtained results showed that the potential time/effort savings were significant. The obtained experimental results also demonstrated that PHOEBE can drastically reduce the time/effort required by a tester to do the performance testing and analysis processes. Finally, the results also proved that PHOEBE is capable of simplifying the configuration of a diagnosis tool. This goal was achieved by addressing the identified trade-offs.
without the need for manual intervention from the tester. Thus, PHOEBE was shown 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. It can be concluded that 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 applications.
Chapter 4

TRINI: A GC-Aware Load Balancing Strategy

This chapter presents TRINI. First, the context of the solution is provided. Next, the internal workings of TRINI are described, including the proposed algorithms and policies. The chapter concludes with the performed experimental evaluation and the discussion of the obtained results.

4.1 Overview

The objective of this research work was to define a GC-aware load balancing strategy, TRINI (shown in Figure 4.1), which is able to dynamically adjust to the specific GC characteristics of a clustered application (typically located within a data centre). This strategy would allow a load balancer to forecast the occurrence of the MaGC events with enough accuracy to exploit that information for improving the performance of the cluster.

The conceptual view of the solution is depicted in Figure 4.2. TRINI periodically retrieves information from the application nodes in order to characterise it. Then, it identifies the most suitable policy based on the GC characteristics of the application running on each node (termed as program family). Finally, the chosen policy is used to forecast the MaGC events and balance the incoming workload among the available application nodes.

As defined by multiple authors [134, 105], self-adaptation provides a system with the capability of adapting itself autonomously to changes in its environment to achieve particular quality goals in the face of uncertainty.
In the context of TRINI, it means minimising the performance impacts of the GC within the cluster. To incorporate self-adaptation to TRINI, I have followed the well-known MAPE-K adaptive model \[82\] (discussed in Section 2.1.7). This model was chosen because it allows to neatly decouple the adaptive layer from the business logic, hence increasing the modularity of the solution.

In TRINI, the Knowledge element is fulfilled by the set of identified program families. The encapsulation of the knowledge into families allows TRINI to be easily extensible and capable of incorporating multiple load-balancing policies, which might be suitable to different scenarios and application behaviours. In this context, a program family encompasses a set of

![Figure 4.1: TRINI - A GC-Aware LB Strategy.](image)

![Figure 4.2: TRINI - Conceptual View.](image)
programs which can be treated similarly because they share some common GC characteristics. For example, a set of program families might be defined according to the duration of the MaGCs. One family can be defined for those programs which tend to suffer MaGCs of small duration (e.g. a few hundred milliseconds). This is because these MaGCs do not normally represent a major performance issue. On the contrary, another family can be defined for those programs which tend to suffer MaGCs of longer duration.

Each program family has two properties: (1) An evaluation criteria to determine if the GC behaviour of an application qualifies for that family. In the previous example, a possible evaluation criterion might be the comparison of the MaGC duration of the monitored application against the duration ranges of each defined program family. (2) A policy which specifies the rules to perform the MaGC forecasting and load balancing. Following the previous example, a possible policy might be the selection of different ranges of historical data (per family) to be considered in the forecast of MaGCs. These policies also make use of the set of available forecast and load balancing algorithms. These algorithms are discussed in Sections 4.3 and 4.4 respectively.

4.2 Core Process

TRINI has a core process which coordinates its MAPE-K elements. This process (depicted in Figure 4.3) is triggered when the load balancer starts. As an initial step, it uses a default policy (e.g., all the available MiGC history might be used to forecast the MaGCs). This initial policy considers any additional configuration provided at start-up time (e.g., information base such as the load balancing algorithm to use) and it is utilised for all the application nodes. Next, the loop specified in the monitor and analyse phases starts for all the application nodes (in parallel), until the load balancing finishes: A new set of data samples is collected, based on the program GC characteristics used to define the set of available program families (e.g., GC and memory snapshots). After the collection occurs, the analyser process checks if the current program family suits the GC characteristics of the underlying program. If it is not the case, the evaluation criteria of the other program families are assessed to identify the new program family, which is then used until the next evaluation phase occurs. These actions retrieve
CHAPTER 4. TRINI

72

their configurations from the database of program families (represented as dashed arrows in Figure [4.3].) This core process continues iteratively until the load balancing finishes. Furthermore, any exceptions are internally handled and reported.

4.3 MaGA: A MaGC Forecast Algorithm

A fundamental capability required by TRINI is the ability of accurately predicting when the MaGCs will occur. To fulfil this need, I have developed MaGA, which is an algorithm to forecast MaGC events in generational heaps. The following sections describe the internals of the algorithm. Additionally, the below definitions will be used on the algorithm discussion:

- **Time** is always expressed as the number of milliseconds that have passed since the application started.

- **Young/Old Generation Samples** are composed of a timestamp and the usage of the corresponding memory generation.

- **MiGC sample** is composed of the start time, the end time and the memory usage before and after the latest MiGC event.

![Figure 4.3: TRINI - Core Process.](image-url)
• **Observations** are used in a statistical context and are composed of one independent and one dependent values. When the dependent value does not contain historical data, the observation is referred as a forecast observation.

• **Steady state** is the state an application reaches after the JVM finishes loading all its classes. It is assumed that this state has been reached if the number of loaded classes remain unchanged for a certain number of consecutive samples.

### 4.3.1 Overview

Figure 4.4 depicts an overview of the algorithm, which is composed of five phases. First the **Initialisation** which sets the parameters required by the algorithm. After it occurs, the other phases are iteratively done to produce MaGC forecasts continuously: New samples are retrieved from the monitored JVM in the **Data Gathering** phase. Then new observations are generated using the new samples in the **Observations Assembly** phase. Next the **Forecast Calculation** occurs. Finally, the logic awaits a **Sampling Interval** before a new iteration starts. This loop continues until the monitored application finishes or the forecast is no longer needed.

![Figure 4.4: MaGa - Forecast Process Overview.](image)

The algorithm is designed to work on generational heaps, as it is the most common type of Java heap. Furthermore, it only uses standard data that can be obtained from any JVM (such as memory or GC data) to make it easy to implement either within or outside the JVM. If the algorithm is implemented within the JVM, the interaction with potential consumers would be simplified. If it is implemented outside the JVM, the implementation would work with any JVM currently available, facilitating the adoption.
4.3.2 Detailed Algorithm

It is presented in Algorithm 1 and its different phases are explained in the following paragraphs.

Algorithm 1: MaGC Forecast.

<table>
<thead>
<tr>
<th>Input:</th>
<th>Sampling Interval, Forecast Window Size, Warm-up Window Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output:</td>
<td>Forecast time of the next MaGC event</td>
</tr>
</tbody>
</table>

1. steadyState := not reached

2. while forecast is needed do

3. Get new OldGen sample

4. if steadyState is not reached then

5. Get new loaded classes sample

6. if warm-up period is over then

7. steadyState := reached

8. Get new MiGC sample

9. Calculate new memory deltas

10. Update memory totals

11. Generate new observations

12. if steadyState is reached then

13. Forecast memory pending to be allocated

14. Forecast time of the next MaGC event

15. Wait the Sampling Interval

Initialisation. Here the configuration parameters are set:

- **Sampling Interval**: Time interval between each sample. This parameter was defined to allow flexibility in the frequency of the samples (which is related to the amount of overhead introduced in the monitored JVM).

- **Forecast Windows Size (FWS)**: How many observations are used as historical data in the forecast calculation. This parameter was defined to allow flexibility in the amount of MiGC data that is used (as the right amount might vary depending on the GC behaviour of the monitored JVM).
• **Warm-up Window Size**: How many samples are used to determine if the application has reached its steady state. This parameter was defined to allow flexibility in the amount of PermGen data that is used (as the right amount might vary depending on the GC behaviour of the monitored JVM).

**Data Gathering.** Its objective is to capture an updated snapshot of the monitored JVM. It starts by collecting a new Old Generation (OldGen) sample. Then, if the application has not reached the steady state yet, a new loaded classes sample is collected and its history is reviewed. If the warm-up period is over, a flag is set to indicate this. Later a new MiGC sample is collected and added to the MiGC history. After having samples from at least two MiGCs, the next metrics are calculated:

• **Time between MiGCs** ($\Delta T_{MiGC}$): How much time elapsed between the latest two MiGCs.

• **YoungGen Memory Allocation** ($\Delta YMA_{MiGC}$): How much memory was used to create new objects between the latest two MiGCs.

• **OldGen Memory Allocation** ($\Delta OMA_{MiGC}$): How much OldGen allocation occurred because of the latest MiGC (meaning that some objects are old enough to be moved to the OldGen by the latest MiGC).

The above metrics are added up into their respective totals (e.g., Total Time between MiGCs) to keep track of how the metrics grow through time. This data is the key input of the regression models used by the algorithm, as explained below.

**Observations Assembly.** Two types of observations are generated and added to their histories. Each is composed of one independent (y axis) and one dependent (x axis) values: The first type (YoungGen-OldGen) captures the relationship between the memory allocation rates (MAR) in the Young and Old Generations. This captures how the Old Generation grows (eventually leading to a MaGC) in relation to the object allocations requested by the application (which occur in the Young Generation). In this observation the dependent value is the Total YoungGen Memory Allocation and the independent value is the Total OldGen Memory Allocation. The second type of observation (Time-YoungGen) captures the relationship between the time and the Young MAR. Here the dependent value is the Total Time between MiGCs and the independent value is the Total YoungGen Memory Allocation.
CHAPTER 4. TRINI

Forecast Calculation. This phase first evaluates if the application has reached the steady state. If so, two projections are calculated using linear regression models (LRM). The first projection corresponds to how much memory allocation needs to occur in the Young Generation before the free memory in the Old Generation gets exhausted (hence triggering a MaGC). This is calculated by initializing a LRM with a subset of YoungGen-OldGen observations (defined by the FWS) and then feeding the LRM with a forecast observation whose independent value is the sum of the current Total OldGen Allocation and the free OldGen memory. This is shown in Figure 4.5. In this example, the free OldGen memory is 40MB. As the Total OldGen Allocation is 50MB, the independent value of the forecast observation is 90MB. Using the observations within the FWS (the rounded rectangle), the LRM predicts how much memory allocation needs to occur in the YoungGen before the next MaGC occurs (225MB).

The second projection is the core output of this algorithm: The MaGC forecast time. It is calculated by initialising a LRM with a subset of Time-YoungGen observations and feeding it with a forecast observation whose independent value is the result of the first projection. This is represented in Figure 4.6. Using the observations within the FWS, the LRM predicts when the necessary memory allocation in the YoungGen will occur (225MB in the example), consequently triggering the next MaGC (around the millisecond 600 in the example).

![Figure 4.5: Forecast of Young MemAlloc.](image1)

![Figure 4.6: Forecast of MaGC Event.](image2)
It is worth noticing that the LRM was chosen as the forecast technique used to predict the occurrence of the next MaGC after performing a comprehensive evaluation of the most widely-used forecasting techniques (discussed in Section 2.1.6). The aim was to identify which technique would model the GC better. In this evaluation, the GC behaviour of all the 23 applications belonging to the two most-widely used Java benchmarks (discussed in Section 2.1.10) was captured during 1-hour runs (using 5 different heap sizes). Next, each forecast technique was fed with the captured information. Then, the accuracies of the obtained forecasts were calculated and compared among the different techniques. For this purpose, two of the most popular metrics of forecast accuracy were used: The mean absolute error [108] (which measures how close the forecasts are to the eventual outcomes by calculating an average of the absolute forecast errors), and the root mean squared error [108] (which represents the standard deviation of the differences between the predicted values and the observed values). The results of this analysis showed that LRM outperformed the other techniques in both metrics. This was because LRM was able to capture better the behaviour which is commonly experienced by the OldGen memory usage of a Java application in a generational heap, which tends to grow relatively linear (regardless of the actual pace and slope) between MaGCs. An example of this behaviour is shown in Figure 4.7. There, it can be noticed how the OldGen memory usage gradually grows until exhausting. Whenever that occurs, a new MaGC is triggered (shown as a green circle in the figure), freeing some memory.

![Figure 4.7: OldGen Memory Usage through time.](image)
Sampling Wait Period. Finally, the process waits the number of milliseconds configured in the Sampling Interval before starting the next round of iterative steps of the algorithm.

4.4 GC-Aware Load Balancing Algorithms

To evaluate the performance gains that can be achieved by adapting the load balancing based on the MaGC forecast information, I have modified four well-known load balancing algorithms. Among the range of available algorithms, I selected the four described in Section 2.1: Round-robin (RR), random (RAN), weighted round-robin (WRR) and weighted random (WRAN). As the experimental results will show, the achieved performance improvements are evident for the four algorithms and so it is expected that TRINI can yield similar results when applied to other load balancing algorithms.

4.4.1 Overview

The main difference of my algorithms (compared against their original counterparts) is that they perform an additional check in the selection of the next node. That is, if the pre-selected node (as per their original selection criteria) is about to suffer a MaGC within a specified threshold (time when a node stops being considered a feasible candidate because the next MaGC is too close), that node is skipped and the next node is evaluated. Once the MaGC is over, the affected node is again available for selection. For example, if the time threshold is 5 seconds and the current time is 5:00:00 PM, any nodes that have a MaGC predicted to occur between 5:00:00 PM and 5:00:05 PM will be skipped in the load balancing iteration as their forecasts fall within the specified threshold. An additional change made to the GC-aware algorithms was the inclusion of an escape condition to prevent an infinite loop in the case that all nodes were about to suffer a MaGC within the defined threshold. If this occurs, the GC-aware algorithms would behave as their original counterparts.

4.4.2 Example GC-Aware Algorithm

An example of the proposed algorithms is presented in Algorithm 2, which shows the GC-aware weighted round robin (GC-WRR). When compared
Algorithm 2: GC-WRR.

Input: $D = \{d_1, d_2, \ldots, d_n\}$, set of application nodes  
       $W = \{w_1, w_2, \ldots, w_n\}$, set of weights per application node  
       $T \in \{\mathbb{N}\}$, MaGC threshold

Output: $d_i \in D$

1. $i := 0$

2. while load balancing is needed do

3.     $fTries := 0$

4.     $found := false$

5.     if isRuntimeWeightsZeroed() then

6.         resetRuntimeWeights($W$)

7.         $i := 0$

8.     while found = false do

9.         if $i \geq n$ then

10.             $i := 0$

11.         if $w_i > 0$ then

12.             $w_i := w_i - 1$

13.             $found := true$

14.         if $fTries < n$ then

15.             $fTime := getForecast(d_i)$

16.             $cTime := getCurrentTime()$

17.             if $(fTime - cTime) \leq T$ then

18.                 $found := false$

19.                 $w_i := w_i + 1$

20.                 $i := i + 1$

21.             else

22.                 $fTries := fTries + 1$

23.         else

24.             $i := i + 1$

end while

use $d_i$ for the next workload
against the original WRR, one can notice the two applied changes (lines 14 to 21): An additional check to consider the closeness of a MaGC in the node selection, and an escape condition (the $fTries$ variable) which keeps the count of the evaluated nodes to prevent the previously discussed infinite loop.

### 4.4.3 Abstract GC-Aware Algorithm

While integrating GC-awareness to the four chosen algorithms, I identified certain similarities across the performed changes. This allowed me to abstract the changes into a generic version of a GC-aware load balancing algorithm, as shown in Algorithm 3. There, it can be noticed how, after the original load balancing selection occurs (represented by the function `originalSelection`), the algorithm performs additional steps to select the next node to be used.

**Algorithm 3**: Abstract GC Load Balancing.

| Input: $D = \{d_1, d_2, \ldots, d_n\}$, set of application nodes $T \in \{\mathbb{N}\}$, MaGC threshold |
| Output: $d_i \in D$

```plaintext
1 i := 0
2 while load balance is needed do
3     found := false
4         while found = false do
5             i := originalSelection()
6             found := true
7             if IsMaGcClose($d_i, T$) & AreNodesToEval() then
8                 found := false
9             use $d_i$ for the next workload
10        resetNodesToEval()
```

This new logic is encapsulated in the functions `IsMaGcClose`, `getForecast`, `markAsEval`, `AreNodesToEval` and `resetNodesToEval`. The function `IsMaGcClose` (shown in Algorithm 4) is responsible of checking if the next MaGC is “too close” for the tentatively selected node. If it is the case, another node must be selected. Internally, this function uses `getForecast` (which
is a wrapper of the MaGA algorithm discussed in Section 4.3, and \textit{markAsEval} (which is responsible of marking, probably through a data structure like a hash table [5] or a vector [6], the nodes after they have been evaluated for the current balancing decision). Meanwhile, the responsibility of \textit{AreNodesToEval} is to check if all nodes have been evaluated for the current balancing decision (to avoid a potential never-ending loop). Finally, the function \textit{resetNodesToEval} is responsible of clearing the marks after the decision has been taken. Due to the relative low complexity (and broad spectrum of possible implementations) of the functions \textit{markAsEval}, \textit{AreNodesToEval} and \textit{resetNodesToEval}, I only describe their responsibilities, instead of specific implementations.

\begin{algorithm}
\caption{Evaluate Closeness of the MaGC.}
\begin{algorithmic}
\State \textbf{Input}: \(d_i \in D\), tentatively selected node \(T \in \{\mathbb{N}\}\), MaGC threshold
\State \textbf{Output}: \(b\text{MaGcCloseness}\)
\State \(f\text{Time} := \text{getForecast}(d_i)\)
\State \(c\text{Time} := \text{getCurrentTime}()\)
\If{\((f\text{Time} - c\text{Time}) \leq T\)}
\State \(b\text{MaGcCloseness} := \text{true}\)
\State \text{markAsEval}\((d_i)\)
\Else
\State \(b\text{MaGcCloseness} := \text{false}\)
\EndIf
\State \text{return} \(b\text{MaGcCloseness}\)
\end{algorithmic}
\end{algorithm}

\section{MiGC-CV Program Families}

Among the alternative strategies to develop policies for TRINI, I initially concentrated on automating the selection of the FWS. This is because the performed accuracy assessment (described in Section 4.6.1) showed that the accuracy of the MaGA algorithm is particularly sensitive to this configuration. This sensitivity occurs because the FWS delimits the degree of knowledge (in terms of historical memory data) which is used to forecast the MaGCs (as explained in Section 4.3). Also, in those experiments no
single FWS achieved the lowest forecast error in all the cases, showing that there is no “best-fit-for-all” FWS.

Additionally, those experimental results showed that the MaGA algorithm tends to benefit from having more historical data available. However, this growth is usually not monotonic. On the contrary, the optimal FWS might experience troughs. This behaviour is captured by the $MiGC_{CV}$ metric \cite{113} (which measures the coefficient of variation in terms of the number of MiGCs which occur between MaGCs). This approach makes the $MiGC_{CV}$ metric an appropriate criterion to classify the different program behaviours into families. For example, whenever there is a large variation in the number of MiGCs that occur between MaGCs (reflected in a high value of $MiGC_{CV}$), using more historical data is not useful because that history does not properly capture the dramatic (several orders of magnitude) changes in memory behaviour. On the contrary, if only the most recent history is used in this scenario (implicitly meaning the usage of a smaller FWS), the forecast accuracy is significantly improved.

Based on the observed behaviours, three $MiGC_{CV}$ program families were experimentally identified: Low ($MiGC_{CV} \leq 0.1$), medium ($0.1 < MiGC_{CV} < 1.0$), and high ($MiGC_{CV} \geq 1.0$). For each family, a FWS trending function was derived, focusing on those MaGCs that benefit from using the increments in MiGC history (while leaving the outliers out of the trend). The validity of the derived models was reflected in their calculated coefficient of determination \cite{110} values, which were above 0.9 (a threshold commonly accepted in statistics as the minimum value to consider a trending function representative of the modelled data). These function-based policies then allowed to automate the selection of an appropriate FWS on a case by case basis. The obtained experimental results (described in Sections \ref{section:4.6.2}, \ref{section:4.6.3} and \ref{section:4.6.4}) demonstrated that these functions were able to accurately predict a good percentage of the MaGC events. Finally, the obtained accuracy results (described in Section \ref{section:4.6.1}) also showed that the number of outliers tend to decrease in larger (e.g. gigabytes) heap sizes. This behaviour supported the decision of ignoring the outliers from the derived functions.
4.6 Experimental Evaluation

The behaviour of a load balancing strategy is heavily influenced by the accuracy of its balancing decisions and the amount of resources it uses [135]. A deep understanding of these factors is key to comprehend the practicability of any load balancing strategy. Therefore, I have performed an exhaustive assessment of TRINI in terms of accuracy, generality, scalability and reliability. This section presents the four experiments performed to assess the benefits and costs of using TRINI. Firstly, I evaluated the accuracy of the MaGa algorithm across a set of Java programs and heap sizes. Secondly, I evaluated the generality of TRINI’s behaviour across a set of different load balancing algorithms. Thirdly, I evaluated the scalability of TRINI’s behaviour across a range of different cluster sizes. Finally, I evaluated the reliability of TRINI’s behaviour over extended time periods. The section concludes with a discussion for practitioners where I summarise the key findings and observations.

4.6.1 Experiment #1: Accuracy Assessment

Here, the objective was to identify which program GC characteristics could work better for defining an initial set of program families. To achieve this, I evaluated the accuracy of the MaGa algorithm (discussed in Section 4.3) against a broad set of Java programs and experimental configurations with the aim of analysing the conditions in which the algorithm performs better. The following sections describe this experiment and its results.

4.6.1.1 Experimental Set-up

In the following paragraphs I present the developed prototype, the test environment and the parameters that defined the evaluated experimental configurations: The selected range of FWS, Java benchmarks, and GC strategies. I also describe the evaluation criteria used in this experiment.

Prototype. Inspired by other works [46] that have aimed to minimise the potential impacts on the monitored environment, the forecast logic was implemented external (non-intrusive) to the JVM. For this purpose, I have used the Java Management Extension (JMX) [29] to interact with the monitored JVM. JMX was chosen because it is a standard Java technology which can retrieve all the information needed for predicting the MaGC events (e.g.,
memory usages or GC snapshots). The prototype also uses the OpenForecast library [36] for all statistical calculations.

**Java Benchmarks.** The DaCapo benchmark 9.12 [40] and the SPEC-JVM benchmark 2008 [38] were chosen because they offer a wide range of different program behaviours to test. Moreover, they are two of the most widely-used Java benchmarks in the literature (as discussed in Section 2.1.10). As the DaCapo benchmark offers different test workloads per program [8], the largest workload for each program was used to stress the GC as much as possible. Also their number of iterations were set in such a way that MaGCs were triggered for all the tested heap sizes. This information is summarised in Table 4.1. Similarly, the SPECJVM programs were configured to trigger MaGCs. It involved setting the iteration time of each program to 60 minutes. As the sunflow program is present in both benchmarks, it was run only once. Also, a warm-up timeframe of 5 seconds was found to be big enough to allow all programs to finish loading their classes before the first forecast was generated. Finally, 5 different heap sizes were tested per program (100, 200, 400, 800 and 1600MB) with the aim of diversifying more the evaluated GC behaviours (as the heap size is a major factor affecting the GC behaviour [57]).

**GC Strategies.** The three strategies discussed in Section 2.1 were used: Serial GC (sGC), Parallel GC (pGC), and Concurrent GC (cGC). This decision was taken with the aim of diversifying more the evaluated GC behaviours (as the selection of the GC strategy is also a major factor affecting the GC behaviour [57]).

**MaGC Forecast Algorithm.** An extensive range of FWS values was tested: [10..3000] in increments of 10. Moreover, a value of 100 ms was selected as sampling interval, assuming that no more than one MiGC would occur within that timeframe (hence not missing to sample any MiGC). Finally, a warm-up window size of 50 was used (the result of dividing the identified warm-up timeframe by the sampling interval).

**Environment.** All tests were done in a virtual machine (VM) with 4 virtual CPUs, 3GB of RAM, and 50GB of HD; running Linux Ubuntu 12.04L, and OpenJDK JVM 7u25-2.3.10. Additionally, the JVM was configured to initialise its heap to its maximum size to keep it constant during the experiments, and the calls to programmatically request a MaGC were disabled. The VM was located on a Dell PowerEdge T420 server [12] equipped

**Evaluation Criteria.** The following three metrics were calculated:

- The forecast error (FE) [113]. This metric is the ratio of the absolute forecasting error (the difference between the forecast time and the time of the real MaGC event) as a proportion of the time elapsed since the previous MaGC. It is usually expressed as a percentage to be comparable among different programs, where lower values are better. Alternatively, the FE can be expressed as forecast accuracy (FA), which is the difference between the maximum possible accuracy (100%) and the FE. In terms of FA, higher values are better.

- The average number of MiGCs that occurred between two MaGC events ($MiGC_{AVG}$) [113]. This metric captures the relationship between the heap size and the memory allocation required by an application (major factors influencing the GC, as proved in [121] and [101], respectively). The smaller the $MiGC_{AVG}$ is, the more MaGCs are triggered, in which case the application more frequently exhausts its
old generation memory. If the value is close to zero (e.g., 5 or lower),
the application is close to an out-of-memory exception. On the con-
trary, a value far from zero (e.g., 1000 or higher) indicates that the old
generation is infrequently exhausted.

- The coefficient of variation ($\text{MiGC}_{CV}$) [113]. This metric is the stan-
dard deviation of the $\text{MiGC}_{AVG}$ expressed as a percentage of the
average, and allows the comparison of different applications in terms
of their variability in memory usage.

4.6.1.2 Experimental Results

As a first step, the analysis focused on assessing the accuracy of the MaGa
forecast algorithm. Even though the results varied among the different GC
strategies, it was possible to achieve high accuracy (meaning a FE below
10%) for all the tested experimental configurations. Figure 4.8 presents the
results obtained for each of the GC strategies. There, it can be observed the
achieved average forecast error, which ranged between 3% and 5%; while
the standard deviation ranged between 2% and 4%.

![Forecast Error per GC Strategy](image)

Figure 4.8: Forecast Error per GC Strategy.

As no single FWS achieved the lowest FE for all programs, the analysis
next centred on identifying the optimal FWS (the FWS which achieved the
FE closest to zero) per combination of program and heap size. This analysis showed an interesting trending: In general, the forecast algorithm tends to benefit from having more historical data available. This causes that the optimal FWS tends to grow through time. However, this growth was not steady in most of the cases. On the contrary, the optimal FWS experienced troughs during the execution of most of the programs (meaning that in those cases, less history was better to achieve a low FE).

Based on this behaviour, the analysis centred on understanding the causes of these troughs. To assess if the troughs were caused by the variability of the program behaviours (in terms of memory usage), I analysed the \( \text{MiGC}_{CV} \) of the programs. This analysis showed that there is a relationship between the FWS troughs and the changes in the number of MiGCs that occur between the MaGCs (which is the key input used by the MaGC forecast algorithm). Whenever those changes are “too drastic” (reflected in a high value of \( \text{MiGC}_{CV} \)), using more historical data is not useful because that history does not properly capture the drastic (several orders of magnitude) changes in memory behaviour. On the contrary, if only the most recent history is used in this scenario (implicitly meaning the usage of a smaller FWS), the forecast accuracy is drastically improved.

The above finding led me to consider that functions could be derived from the observed behaviours (where the frequency of troughs in the optimal FWS trend is related to the level of \( \text{MiGC}_{CV} \)), and be used as a first set of policies within TRINI. These function-based policies would then allow the automatic selection of an appropriate FWS based on the optimal FWS trend. Three program families were identified, based on their \( \text{MiGC}_{CV} \) values: Low \( (\text{MiGC}_{CV} \leq 0.1) \), medium \( (0.1 < \text{MiGC}_{CV} < 1.0) \), and high \( (\text{MiGC}_{CV} \geq 1.0) \). Then, trending functions were derived from the optimal FWS results. This initial approach did not work well for the programs in the high family, and several programs in the medium family, due to their relatively frequent troughs. The obtained functions were not representative of the modelled data, as they produced coefficient of determination (R2) values below 0.9 (which is the threshold commonly accepted in statistics as the minimum R2 value to consider a trending function representative of the modelled data). These results led me to adjust the scope: Instead of concentrating on achieving high forecast accuracy for all the MaGC events, I focused only on those MaGCs which follow a similar growth trend as the
FWS (hence benefiting from using the increments in MiGC history), while leaving the outliers out of the policies. The hypothesis was that, even though these imperfect FWS trending functions might miss to select an appropriate FWS to accurately predict the MaGCs represented by the removed outliers, the functions could still be useful to accurately predict a fair percentage of the MaGCs; information which consequently would allow TRINI to improve a clustered system’s performance. After removing the outliers, representative FWS trending functions were successfully derived for all the program families.

A final observation of this experiment was that, as long as the total workload processed in an experimental configuration does not change (condition which was satisfied for all the configurations using the same program), the variability (in terms of $\text{MiGC}_{CV}$) decreases when the heap size increases. This is because the MaGCs are more homogeneous (in terms of the number of MiGCs) in such bigger heap sizes. This behaviour favours the chosen strategy of skipping the outliers, as their number decreases when the variability decreases. In contrast, the $\text{MiGC}_{AVG}$ increases when the heap size increases. This is because there is more memory to exhaust before triggering a MaGC. An example of these behaviours is shown in Figure 4.9, which shows the $\text{MiGC}_{CV}$ and $\text{MiGC}_{AVG}$ trends for the sunflow program with pGC when using the different heap sizes.

![Figure 4.9: pGC Sunflow - MiGC$\text{CV}$ vs. MiGC$\text{AVG}$ Trends.](image-url)


Summary. This experiment proved that the MaGa algorithm can accurately forecast the MaGC events (achieving a FE<10% for all the tested programs and heap sizes) when configured properly. The selection of the FWS is particularly important because the algorithm is sensitive to this configuration parameter. Furthermore, the experimental results showed that the $MiGC_{CV}$ metric is an appropriate criterion to classify the different program behaviours into families. These families then facilitated the derivation of function-based policies to automate the FWS selection without the need of manual tuning.

4.6.2 Experiment #2: Generality Assessment

The objective of this experiment was to evaluate the generality of the benefits and costs of using TRINI. To achieve this, I compared the behaviour of TRINI applied to four commonly used load balancing algorithms. The following sections describe this experiment and its results.

4.6.2.1 Experimental Set-up

In the following paragraphs I present the developed prototype, the test environment and the parameters that defined the evaluated experimental configurations: The selected load balancing algorithms, Java benchmarks, and GC strategies. I also describe the evaluation criteria used in this experiment.

Prototype. It was built on top of the Apache Camel [1], which is a popular light-weight load balancer. This solution was chosen because it is open source and developed in Java, characteristics which facilitated its integration with the MaGC forecast logic. Additionally, the architecture of this load balancer offers well-defined extension points. This characteristic facilitated the implementation of the GC-aware load balancing algorithms.

Environment. All the experiments were performed in an isolated test environment, so that the entire load was controlled. This environment was composed of fifty two VMs: A cluster of fifty application nodes with one load balancer, and one load tester node (as shown in Figure 4.10). 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, and OpenJDK JVM 7u25-2.3.10 with a 1.6GB heap. Each JVM was configured to initialise its heap to its maximum size, and the calls to programmatically request
a MaGC were disabled. The load tester node also used an Apache JMeter 2.9 [2] (a leading open source tool used for application performance testing), and the application nodes ran an Apache Tomcat 6.0.35 [3] (a popular open source Web Application Server for Java).

![Cluster diagram](image)

Figure 4.10: TRINI - Test Environment.

The VMs were located on two Dell PowerEdge M620 servers [11] inside a Dell PowerEdge VRTX chassis [13]. Each server was equipped with 2 Intel Xeon E5-2660 v2 CPUs at 2.20GHz (10 cores/20 threads), 192 GB of RAM, a 10 GbE network card, and VMware ESXi 5.5.0 as hypervisor. Additionally, the chassis provided 12.3 TB of SAS storage (composed of a hard-drive backplane with 14 hard drives) through a 6 Gbps PERC adapter.

**Load Balancing Algorithms.** The four algorithms discussed in Section 2.1 were tested: Round robin (RR), random (RAN), weighted round robin (WRR) and weighted random (WRAN); as well as their developed GC-aware counterparts (GC-RR, GC-RAN, GC-WRR, GC-WRAN). Two types of runs were performed: The first type used the original version of each algorithm and was considered the baseline in the analysis of the results. The second type of run used the GC-aware version of each algorithm. Regarding the MaGC forecast algorithm (which is internally used by the GC-aware algorithms, as explained in Section 4.4), the FWS was automatically selected by the function-based policies described in Section 4.5. Additionally, a value of 100 ms was selected as the *sampling interval*, assuming that no more than one MiGC would occur within that timeframe (hence not missing any MiGC).
CHAPTER 4. TRINI

Benchmarks. Two of the Java benchmarks most widely-used in the literature (DaCapo 9.12 [40] and SPECJVM 2008 [38]) were chosen because they offer a wide range of 23 different programs 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. Section 2.1.10 presents a summary of these benchmarks and their programs.

In order to be able to call a program from within a JMeter HTTP test script (so that multiple concurrent calls could be invoked per application node), a wrapper JSP was developed and installed in the Tomcat instance of each application node. For each program, a JMeter test script was created, adding some controlled diversity to the workload. For the DaCapo programs, it involved varying the workload size between program calls (using the available pre-defined workload sizes of DaCapo [8]). In the case of the SPECJVM programs, the controlled diversity involved varying the execution time (in the range of 30 to 90 seconds). Each JMeter test run lasted 60 minutes and used 500 concurrent users. Finally, each individual program call was considered a transaction.

GC Strategies. The three strategies discussed in Section 2.1 were used: Serial GC (sGC), Parallel GC (pGC), and Concurrent GC (cGC). This decision was taken with the aim of diversifying more the evaluated GC behaviours (as the selection of the GC strategy is a major factor affecting the GC behaviour [57]).

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 overhead, the main metrics were CPU (%) and memory (MB) utilisations. In both cases, lower values are better. These metrics were collected with the top command [33].

Regarding forecast accuracy, the three metrics discussed in Section 4.6.1.1 were also calculated: The forecast error (FE), the average number of MiGCs that occurred between two MaGC events ($MiGC_{AVG}$), and the coefficient of variation ($MiGC_{CV}$).

4.6.2.2 Experimental Results

In this section I present the results obtained from this experiment in terms of the relevant performance and overhead metrics.
CHAPTER 4. TRINI

Performance Improvements. To understand the behaviour of TRINI across the evaluated experimental configurations, I initially focused my analysis on assessing the performance improvements that TRINI achieved. In this context, a performance improvement for a particular metric (e.g., response time) is the difference between an experimental configuration using a GC-aware load balancing algorithm (e.g., GC-WRR) and its counterpart using the corresponding original algorithm (e.g., WRR). In terms of throughput, a performance improvement implies a positive difference (as higher throughput is better) and has a value greater than 0%. In terms of response time, a performance improvement implies a negative difference (as lower response time is better) and has a value in the range between 0% and 100%.

The overall results showed that TRINI worked well, as all the GC-aware experimental configurations achieved performance improvements. More importantly, the behaviours of the four tested GC-aware load balancing algorithms were similar, as they achieved comparable performance improvements. Figure 4.11 shows the results in terms of average response time ($RT_{AVG}$). There, it can be observed the achieved average performance improvements, which ranged between 28% and 31%. It should be noted that these results are aggregated across the full set of benchmark applications, which have a wide range of memory behaviours, so that the observed standard deviations ranged between 21% and 24%. Figures 4.12 and 4.13 depict the obtained results in terms of maximum response time ($RT_{MAX}$) and average throughput ($T_{AVG}$). There, it can be noticed that both metrics also experienced behaviours which were similar across all the tested load balancing algorithms.

The next round of my analysis focused on evaluating the sensitivity of TRINI with respect to the different GC strategies used. These results are presented in Figures 4.14 ($RT_{AVG}$), 4.15 ($RT_{MAX}$) and 4.16 ($T_{AVG}$). There, it can be seen that the average performance improvements achieved by TRINI were relatively close across the three GC strategies, meaning that TRINI worked well irrespectively of the strategy. The biggest gains occurred with the sGC as this GC strategy experienced the most time-consuming MaGC events (hence having the largest potential gain to exploit).

The previous two analyses were useful to obtain a high-level view of the achieved performance improvements. However, these analyses did not cap-
CHAPTER 4. TRINI

Figure 4.11: $RT_{AVG}$ - Perf. Improv. per LB.

Figure 4.12: $RT_{MAX}$ - Perf. Improv. per LB.

Figure 4.13: $T_{AVG}$ - Perf. Improv. per LB.

Figure 4.14: $RT_{AVG}$ - Perf. Improv. per GC.

Figure 4.15: $RT_{MAX}$ - Perf. Improv. per GC.

Figure 4.16: $T_{AVG}$ - Perf. Improv. per GC.
ture the differences in memory behaviours across the tested applications (reflected in the relatively high standard deviations obtained after consolidating the results). Therefore, additional investigation, from a more memory/GC-oriented perspective, was required.

As a next step, I focused on understanding the reasons behind the achieved performance gains. For this purpose, I analysed the results in terms of MiGC\textsubscript{CV} behaviour, as illustrated in Figure 4.17. There, a clear relationship can be observed between the forecast accuracy achieved by TRINI and the MiGC\textsubscript{CV} of the different application behaviours. In general, the lower the variability, the more accurate TRINI is. More importantly, the forecast accuracy reaches practically 100\% when the variability is below 0.1. This behaviour persisted irrespective of the chosen load balancing algorithm or GC strategy. These results show how MiGC\textsubscript{CV} is an appropriate metric to characterise the program behaviours into families.

In my experiments, I also identified that the performance improvements yielded by TRINI are mainly driven by two factors:

1. The total time spent on MaGC in all the application nodes (MaGC\textsubscript{D}), as it captures the amount of potential gain that can be obtained.
2. The forecast accuracy (FA) of TRINI, which is the actual enabler that allows the potential gains to be converted into actual gains (by diverting the workload from any node which is suffering a MaGC).

In general terms, the performance improvements tend to be bigger when the MaGC\textsubscript{D} is long (as there is more potential gain to exploit). How-
ever, the actual benefits depend on the amount of $MaGC_D$ that is actually addressed ($A-MaGC_D$). This behaviour is depicted in Figure 4.18 which shows the achieved performance improvements (in terms of $RT_{AVG}$) with respect to the $A-MaGC_D$ and the FA. The $A-MaGC_D$ is expressed as a percentage of the total execution time. The FA is grouped in three levels: low ($30\% \leq FA \leq 50\%$), medium ($50\% < FA \leq 80\%$), and high ($FA > 80\%$). In Figure 4.18 it can be seen how the improvements for a particular level of FA (e.g., high), tend to be bigger when the $A-MaGC_D$ is longer. It can also be noticed how the $A-MaGC_D$ highly influences the achieved performance improvements. For example, the biggest improvements were achieved by those configurations which experienced the longest $A-MaGC_D$, even though they only achieved a medium level of FA.

In this context, a MaGC was considered addressed if it was forecasted accurately enough that it was possible to prevent sending transactions to the affected node during the MaGC occurrence. Under these conditions, the only transactions affected by the MaGC event were those in the pipeline to be processed by a node which suffered the MaGC.

This behaviour is further explained by Figure 4.19 which shows how the FA translates into $A-MaGC_D$. In general terms, the higher the FA, the bigger the amount of $A-MaGC_D$. However, the relationship is not entirely linear. This is because the amount of $A-MaGC_D$ depends not only on the number of MaGCs which were not addressed (as measured by the FA), but also on the durations of those MaGCs. For instance, it is not the same
performance impact to inaccurately forecast a MaGC that lasts two minutes, than a MaGC that lasts two seconds (even though both MaGC events are equally captured by the FA metric).

**Overhead.** I also studied the costs of using TRINI. For this analysis, I categorised the possible overhead in two types: The overhead introduced in the application nodes, and the overhead in the load balancer node.

In the application nodes, TRINI proved to be light-weight in terms of CPU and memory across all the load balancing algorithms. The increment in average CPU usage ($\Delta CPU_{AVG}$) across all tested applications was 1.46%, with a standard deviation of 0.43%; while the increment in average memory usage ($\Delta MEM_{AVG}$) was 0.55%, with a standard deviation of 0.35%. These increments were caused by the data gathering process, which collects information from the different application nodes (performed through JMX, as explained in Section 4.6.1.1). These results are presented in Figures 4.20 and 4.21 which show the $\Delta CPU_{AVG}$ and $\Delta MEM_{AVG}$, respectively.

In the load balancer node, the introduced overhead was higher (compared to the application nodes), but still within a reasonable level for a 50-node cluster. The $\Delta CPU_{AVG}$ was 27.88%, with a standard deviation of 1.30%; while the $\Delta MEM_{AVG}$ was 6.34%, with a standard deviation of 0.71%. Additionally, the four load balancing algorithms performed similarly, suggesting that the level of introduced overhead was independent of the algorithm. These results are presented in Figures 4.22 ($\Delta CPU_{AVG}$)
and \[4.23\] (\(\Delta MEM_{AVG}\)). The \(\Delta CPU_{AVG}\) was mainly caused by the forecast algorithm, as it continuously generates an updated MaGC forecast for each application node. Regarding the memory consumption, approximately 4\% of the \(\Delta MEM_{AVG}\) was caused by the initialisation of TRINI. The remaining increment was due to the historical data that was kept for forecasting purposes.

**DaCapo/SPEC Program Classification.** The results of this experiment also allowed to classify the 23 tested programs according to their GC characteristics (the \(MaGC_D\) and the \(MiGC_{CV}\)). This program classification is shown in Table \[4.2\]
### Table 4.2: DaCapo/SPEC Program Classification per GC Behaviours.

<table>
<thead>
<tr>
<th>MiGC&lt;sub&gt;CV&lt;/sub&gt;</th>
<th>MaGC&lt;sub&gt;D&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Low</strong></td>
<td>Short</td>
</tr>
<tr>
<td></td>
<td>compiler, jython</td>
</tr>
<tr>
<td></td>
<td>Medium</td>
</tr>
<tr>
<td></td>
<td>avrora, compress, fop, luindex, lusearch, mpegaudio, tomcat, startup, sunflow, xalan</td>
</tr>
<tr>
<td></td>
<td>High</td>
</tr>
<tr>
<td></td>
<td>batik, crypto, eclipse, pmd, tradebeans</td>
</tr>
<tr>
<td></td>
<td>Medium</td>
</tr>
<tr>
<td></td>
<td>h2, scimark, tradesoap, xml</td>
</tr>
<tr>
<td></td>
<td>High</td>
</tr>
<tr>
<td></td>
<td>derby, serial</td>
</tr>
</tbody>
</table>

**Summary.** This experiment demonstrated the performance gains that TRINI can bring to a cluster. By avoiding the impact of most of the MaGC events in the individual nodes, the performance of the clustered applications was significantly improved. More importantly, the improvements were achieved irrespectively of the used load balancing algorithm or GC strategy, proving the generality of TRINI. Regarding the overhead, the increments in CPU and memory usage in the application nodes were minimal, hence not affecting their normal operation. Even though the level of tolerable overhead in the load balancer node would depend on the particular usage scenario, the obtained increments were considered acceptable because the load balancer node was far from exhausting its resources (especially considering the relative modest characteristics of the load balancer node, described in Section 4.6.2.1).

### 4.6.3 Experiment #3: Scalability Assessment

Here the objective was to evaluate the scalability of TRINI by assessing its behaviour in different sizes of clusters. The following sections describe this experiment and its results.

#### 4.6.3.1 Experimental Set-up

The set-up was similar to that used in experiment #2 (presented in Section 4.6.2.1), with the following differences: The cluster size was variable, covering the range of [5, 50] application nodes in increments of 5. The number of concurrent users was increased proportionally to the cluster size (e.g.,
the 5-node cluster used 50 users, the 10-node cluster used 100 users, and so on) so that the workload was increased accordingly. As experiment #2 proved that TRINI works well irrespective of the load balancing algorithm, I centred on the WRR because it is currently the most widely-used load balancing algorithm [59]. Likewise, I concentrated on the Serial GC strategy because it tends to suffer the longest pauses [127], hence benefiting more from my work. Finally, I focused on 5 programs which were representative of the program classification presented in Section 4.6.2.2. This configuration allowed to test a diverse set of GC behaviours with a smaller set of experimental configurations. That classification (shown in Table 4.2) grouped the programs of the DaCapo and SPECJVM benchmarks according to their GC characteristics (the MaGC\textsubscript{D} and the MiGC\textsubscript{CV}). In Table 4.2 the programs underlined are the ones used in this experiment: jython, sunflow, eclipse, scimark, and derby.

4.6.3.2 Experimental Results

In this experiment, the analysis focused on two main aspects: Evaluating the performance improvements yielded by TRINI in different sizes of clusters; and assessing the behaviour of the overhead in such clusters.

**Performance Improvements.** The hypothesis was that the performance improvements should not degrade when the cluster size increases, as each forecast process is independent of each other (hence not affected by the number of monitored application nodes). This was confirmed by the results of the experiment. Even though there were some minor variations in the percentage of performance improvements that TRINI achieved, the improvements were closely similar, across the different cluster sizes, per tested program. Figure 4.24 shows the obtained performance improvements in $RT_{AVG}$. There, it can be noticed how the improvements were relatively constant, per program, through the different cluster sizes. Furthermore, the differences in improvements among the tested programs were due to their diversities in memory/GC behaviour. For example, the scimark program obtained the biggest improvements because it experienced the longest MaGC\textsubscript{D} and also achieved a high forecast accuracy (above 90%). On the contrary, the jython program obtained the smallest improvements because it suffered the shortest MaGC\textsubscript{D} (meaning it had the smallest potential gains). Similar trends were observed in terms of $RT_{MAX}$ and $T_{AVG}$. 
Overhead. Two main findings were identified in terms of overhead. First, the cost in the application nodes of using TRINI was minimal and relatively constant and independent of the cluster size. As previously explained, the forecast process for each application node is independent of each other. Thus, the same principle applies to the data gathering that occurs in the nodes. This can be noticed in the results of the analysis of the $\Delta CPU_{AVG}$ and $\Delta MEM_{AVG}$ in the application nodes per cluster size (shown in Figures 4.25 and 4.26 respectively). There, it can be observed how the increments in utilisation of both resources were very low. Additionally, they presented a relatively uniform distribution across all the tested cluster sizes.

Second, the overhead in the load balancer node was dependent of the cluster size, following a relatively smooth growth trend. In the case of the $\Delta CPU_{AVG}$, these increments were mainly caused by the increase in the number of concurrent forecast processes (as there was one forecast process per monitored application node). This explains the relatively linear nature of the growth. These trends are shown in Figure 4.27. In the case of the $\Delta MEM_{AVG}$, the observed increments were mainly caused by the amount of data that was gathered from the application nodes for forecasting purposes. Under these circumstances, if the application triggers a considerably high number of MaGCs and/or MiGCs, the amount of memory required to keep this historical data might become significant. This behaviour can
be observed in Figure 4.28, which presents the $\Delta \text{MEM}_{AVG}$ trending per application. There, it can be noticed how the *derby* program presented a relatively higher slope (compared to the other programs). This is because *derby* generated not only the largest amount of GC historical data, but it was also considerably bigger (several orders of magnitude) than the other programs. It is worth mentioning that, despite the relatively high slope, the amount of memory required by TRINI to support *derby* was still below 10% of the total available memory (on the load balancer node), even with 50 application nodes. This level of utilisation leaves a considerable amount of idle resources to support many more application nodes.
Summary. In conclusion, the results of this experiment showed how TRINI can scale gracefully for larger clusters. The achieved performance improvements did not degrade when increasing the size of the cluster, while also the computational resources used by TRINI did not significantly increase.

4.6.4 Experiment #4: Reliability Assessment

Here the objective was to evaluate the reliability of TRINI by assessing its behaviour in longer (24-hour) experimental test runs. The following sections describe this experiment and its results.

4.6.4.1 Experimental Set-up

The set-up was similar to that used in the experiment #3 (presented in Section [4.6.3.1]), with two differences: First, the evaluated cluster was composed of 50 application nodes (same size as experiment #2, described in Section [4.6.2.1]). Second, the duration of the test runs was increased from 1 to 24 hours to evaluate TRINI on a longer, more realistic duration.

4.6.4.2 Experimental Results

In this section I present the results obtained from this experiment in terms of the relevant evaluated metrics.

Performance Improvements. To understand the performance improvements achieved by TRINI through the experiment, I carried out a breakdown of the behaviour of each experimental configuration on an hourly basis. The results of this analysis showed no serious degradation in the obtained improvements during the 24-hr test runs, proving that the behaviour of TRINI (in terms of performance improvements) remains stable through time (reflected in a low standard deviation). Among the tested programs, the largest standard deviation occurred in the scimark program. This behaviour was compensated by the performance improvements achieved (e.g., an average of 69\% in terms of $RT_{AVG}$), which were the highest among the tested programs. Figure [4.29] shows the results in terms of $RT_{AVG}$. Similar results were obtained in terms of $RT_{MAX}$ and $T_{AVG}$ (as shown in Figures [4.30] and [4.31] respectively).
Overhead. The results of the analysis showed that TRINI does not degrade the behaviour of the application nodes through time. This is because TRINI only causes a minimal (and relatively constant) overhead to them. The $\Delta CPU_{AVG}$ across all tested applications was 1.02%, with a standard deviation of 0.38%; while the $\Delta MEM_{AVG}$ across all tested applications was 0.42%, with a standard deviation of 0.15%.

In the load balancer node, the results of my analysis showed that the $\Delta CPU_{AVG}$ caused by TRINI remained quite steady during the whole experimental test runs (25.57% with a standard deviation of 2.15%). This is because the main contribution to this increase is the number of forecast processes, which is not influenced by time but by the size of the cluster. In
In terms of memory, the $\Delta MEM_{AVG}$ across all tested applications was 6.08\%, with a standard deviation of 0.86\%. This increment remained within a well-defined band during the 24-hour test runs. This stability in the memory footprint of TRINI is the result of an efficient management of the historical data (e.g., MiGC events) which is temporarily stored by TRINI. This data is closely monitored and controlled, so that whenever it becomes older than the required FWS (which delimits the history that is used for forecasting), the data is automatically purged.

Summary. The results of this experiment demonstrated the reliability of TRINI through time, as TRINI was capable of improving the performance of a clustered system without suffering from a degradation in its behaviour. In terms of overhead, TRINI experienced a relatively uniform $\Delta CPU_{AVG}$ during the whole test runs. Similar behaviour was observed in terms of $\Delta MEM_{AVG}$ in the application nodes. Finally, TRINI experienced a minimum increase in terms of $\Delta MEM_{AVG}$ in the load balancer node. This was due to the historical data that TRINI temporarily preserved for forecasting purposes.

4.6.5 Discussion for Practitioners

The presented experimental results have demonstrated how adding GC-awareness to a load balancing strategy can significantly improve the performance of a cluster. In the following paragraphs I provide guidelines for practitioners to indicate the conditions under which TRINI can yield improvements and discuss the wider applicability of the technique.

- To estimate the forecast accuracy that TRINI can achieve in a particular usage scenario, the $MiGC_{CV}$ has proven to be a useful metric. In general terms, the lower the GC variability, the more accurate TRINI can be. Specifically, the highest forecast accuracy is obtained when the GC variability is very low ($MiGC_{CV} \leq 0.1$). Under these conditions, the forecast accuracy reaches practically 100\%. This means that basically all the MaGC events are forecasted accurately enough that it is possible to prevent sending transactions to the affected nodes during the occurrence of the MaGC events. Thus, minimising the impact that the GC has on the overall cluster performance. In cases of higher GC variability, the accuracy tends to decrease. However, it re-
mains within reasonable levels. For instance, in the experiments, the programs which experienced the highest variability ($MiGCCV \geq 1.0$) obtained an average forecast accuracy around 55%. This means that even in such volatile conditions, more than half of the MaGCs were accurately forecasted.

- In terms of potential performance improvements, more GC intensive applications (in terms of the amount of time the application spends doing MaGC - $MaGC_D$ -), can benefit most from TRINI. Even though the level of forecast accuracy is important to estimate the amount of MaGC which is actually addressed, the results have shown that even a medium level of forecast accuracy ($50\% < FA \leq 80\%$) can offer significant performance improvements in cases where the $MaGC_D$ is long ($MaGC_D \geq 25\%$). This scenario is more likely to occur when using huge (e.g., gigabytes) heaps because they tend to experience longer MaGC pauses, in comparison to smaller heaps (e.g., megabytes or lower). Additionally, the biggest performance improvements are obtained when an application experiences a long $MaGC_D$ as well as a low GC variability. Under these conditions, TRINI is able to mitigate most of the performance costs caused by the GC. As these costs are also considerable (hence offering a lot of potential gains), TRINI can convert them into actual performance gains. It is also worth mentioning that performance improvements can usually be expected, regardless of the exact amount of $MaGC_D$. This is because the GC is a fundamental feature of Java and, given enough time, any Java application will eventually experience one or more MaGC events (as part of its automatic memory cleaning process).

- In the experimental evaluation, I selected three of the most widely-used GC strategies in the industry. As the results have shown, the achieved performance improvements are evident for all three GC strategies, and so it is expected that TRINI can yield similar results when using other GC strategies. Likewise, it is expected that TRINI should be applicable to other object-oriented languages which rely on GC principles and strategies similar to those used by Java (e.g., Python or C#).

- In the experimental evaluation, I selected four of the most frequently used load balancing algorithms in the industry. As the results have
CHAPTER 4. TRINI

shown, the achieved performance improvements are similar across all the tested load balancing algorithms. Therefore, it is expected that TRINI can yield similar results when using other GC-aware load balancing algorithms. To support practitioners in the task of adding GC-awareness to other algorithms, I have discussed (in Section 4.4) the changes required to make a load balancing algorithm GC-aware, as well as presented an abstract version of a GC-aware load balancing algorithm.

• In terms of the overhead introduced by TRINI to the application nodes, the results have shown that the increments in CPU and memory utilisations are minimal, hence not affecting the normal operation of the application nodes. Nonetheless, if this level of overhead is not tolerable for a particular usage scenario, the overhead can be decreased. This can be done by adjusting the sampling interval to a higher value (e.g., in my experiments I used 100ms). This change would have the effect of decreasing the frequency of the sampling in the application nodes (hence decreasing the amount of resources used), at the expense of increasing the probability of missing to sample a MiGC event. For this reason, it is recommended that the sampling interval should not be higher than the average time elapsed between MiGC events.

• In terms of the overhead introduced by TRINI to the load balancer node, the results have shown that the overhead usually follows a relatively linear growth with respect to the cluster size. For this reason, the results can be used as a valuable input information for a capacity planning process. This would allow practitioners to estimate the CPU and memory characteristics required by a load balancer node to support a particular cluster size.

• Based on the previously discussed points, it is concluded that a GC-aware load balancing strategy can offer significant benefits to a clustered system. Given the broad spectrum of GC behaviours that an application might experience, such GC-aware load balancing strategy should not rely on a static configuration. On the contrary, it should use an adaptive configuration which can self-adjust based on the GC characteristics of the underlying application (as TRINI does). Moreover, there are similarities in the GC behaviours that certain applications
share (e.g., the identified MiGCV program families) and which can be leveraged to make a more robust GC-aware load balancing solution.

4.7 Summary

This chapter presented TRINI, a novel adaptive GC-aware load balancing strategy which enhances the performance of clustered Java systems by avoiding the performance impacts of the MaGC (which is a common cause of performance degradation in Java systems). A detailed discussion of the different algorithms and policies, which complement TRINI’s capabilities, was also presented. Additionally, a comprehensive experimental evaluation of TRINI was discussed. Its results showed that TRINI can achieve significant performance improvements, as well as a consistent behaviour, when it is applied to a set of commonly used load balancing algorithms, demonstrating its generality. TRINI also proved to be scalable across different cluster sizes, as its performance improvements did not noticeably degrade when increasing the cluster size. Finally, TRINI exhibited reliable behaviour over extended time periods, introducing only a small overhead to the cluster in such conditions. These results offer practitioners a valuable reference regarding the benefits that a load balancing strategy, based on garbage collection, can bring to a clustered Java system.
Chapter 5

Conclusions and Future Work

The chapter starts presenting the main conclusions drawn from this thesis work. Next, some ideas on possible directions for future work in the area are discussed.

5.1 Impact of the Contributions

Cluster computing has gained popularity as a powerful and cost-effective solution for parallel and distributed processing. For example, enterprise applications are commonly deployed in clustered instances to achieve higher system performance, compared to single machine solutions. However, it is also a well-known challenge that the usage of clusters has considerable increased the complexity of the systems, further complicating all activities related to the performance optimisation of the clustered systems.

To address the above challenge, this thesis contributes to the area of performance optimisation of clustered systems in Java (which is one of the most predominant technologies at enterprise-level), especially aiming for the large-scale environments (as these environments are more likely to suffer performance issues). In particular, this thesis proposes two techniques to solve the problems of efficiently identifying workload-dependent performance issues and efficiently avoiding the performance impacts of major garbage collection, two important problems that a typical clustered Java system would likely suffer in large-scale environments. The following paragraphs summarise the main contributions of the work presented in this thesis.
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 these limitations, this thesis presented a novel adaptive framework, PHOEBE, to automate the usage of a performance diagnosis tool in a clustered environment. Internally, PHOEBE utilises a set of adaptive policies to control the different set of processes commonly involved in the configuration and usage of a diagnosis tool. A prototype has been developed around a set of well-known diagnosis tools to experimentally evaluate the framework. 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 use 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%). 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.

Another of the most important challenges in cluster computing is how to efficiently distribute the workload among the cluster’s nodes. To address this challenge, this thesis presented TRINI, a novel adaptive GC-aware load balancing strategy. It enhances the performance of clustered Java systems by avoiding the performance impacts of the Major Garbage Collection (which is a common cause of performance degradation in Java systems). Internally, TRINI utilises JVM data, and a set of defined program families, to adapt to the GC characteristics of the underlying application. A prototype has been implemented, and TRINI has been comprehensively evaluated in terms of generality, scalability and reliability in order to offer a valuable reference regarding the behaviour of TRINI in such circumstances. The experimental results have demonstrated that TRINI can significantly improve the response time and throughput of a cluster (e.g., the average performance improvements in $RT_{AVG}$ ranged between 28% and 31%). These performance
improvements were achieved independent of the used load balancing algorithm, proving the generality of TRINI. The results also showed that TRINI is scalable across different cluster sizes, and reliable through time, as the obtained performance improvements did not noticeably degrade when either the cluster size or the length of the test run increased. In terms of overhead, TRINI introduced a minimal overhead to the application nodes of the cluster (e.g., the increment in average memory usage ranged between 0.52% and 0.58%). Additionally, the overhead in the load balancer was low (e.g., the increment in average memory usage ranged between 6.19% and 6.49%), especially considering the modest characteristics of the used load balancer node. From the above results, it can be concluded that a GC-aware load balance strategy, such as TRINI, can bring significant benefits to a clustered Java system.

5.2 Future Work

We believe that this research makes a positive contribution towards the goal of facilitating the performance optimisation of clustered Java systems by automatic means. Nevertheless, there are multiple directions to extend the presented techniques. In the following paragraphs, some plans for possible future work are presented.

Regarding the automation framework for performance testing (PHOEBE), only a subset of possible policies has been explored so far. Therefore, an attractive track of future work is to investigate how best to extend the adaptive and data analytic 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. Moreover, these anti-patterns would not be limited to the outputs of an individual tool. On the contrary, their symptomatology might cover the full spectrum of available diagnosis tools in order to have a better understanding of the performance issues, their potential root causes and solutions.

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. A particularly interesting idea to explore is to investigate how best to dynamically adapt the workload that is used by a test loader (e.g., Apache JMeter) to further improve the process of identifying performance issues. This is because current test loader tools rely on static workloads. This characteristic limits their effectiveness because normally the workload required to identify performance issues would vary depending on the specific application logic and usage scenario. Therefore, it would be preferable that the workload is dynamically adjusted (probably based on the outputs of the used performance diagnosis tools) during the test execution to maximise the number of performance bugs identified in a test run. A similar policy 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).

Furthermore, PHOEBE’s capabilities can be extended to support other types of distributed architectures. For instance, to be better suitable for the performance testing of SaaS [78] applications, PHOEBE needs to properly handle the presence of heterogeneous hardware (as this is a commonly experienced scenario in cloud-based applications, in contrast to clusters, where application nodes are typically homogeneous). To support this usage scenario, PHOEBE would need to leverage the hardware characteristics of the application nodes and consider this additional information in its consolidation processes (e.g., the severity assessment of the identified issues, which would be influenced by the hardware characteristics of the involved nodes). Similarly, PHOEBE can be extended to support distributed systems which, unlike a cluster, are not composed of interchangeable application nodes (i.e. each node has an identical code image of the application). To support this usage scenario, PHOEBE would need to know the topology of the system, so that it can identify the different subsystems within the distributed system and offer new subsystem-level perspectives of PHOEBE’s results.

Regarding the GC-aware load balancing strategy (TRINI), a natural extension to the work presented in this thesis is to continue investigating which other GC characteristics might be suitable to broaden the classification of the different memory behaviours into program families. Then, use that additional knowledge to develop more portable load balancing policies which can exploit the behaviour similarities of the new families. Across the broad range of possibilities, the option that I plan to explore next is to assess the
feasibility of applying pattern recognition and pattern matching techniques to enhance the MaGC forecast accuracy (consequently increasing the obtained performance gains) for those applications which tend to experience high variability in their GC behaviour (currently characterised by the High MiGC\textsubscript{CV} family, and achieving the lowest levels of forecast accuracy among the existing set of program families). The aim is that this complementary strategy could make TRINI more adaptable to the changing GC and memory patterns that those applications might experience during their execution.

As of now, TRINI has exclusively centred on exploring and using an unexploited aspect of the available system resource information: The GC. However, TRINI can be extended to take into consideration other types of inputs (e.g., other system resources or workload-dependent performance issues) in its decision layer in order to build a more sophisticated load balancing solution. As a first step in that direction, I plan to explore the feasibility of using the outputs of a performance diagnosis tool (i.e. through its automatic usage with PHOEBE) to monitor the health of the different application nodes. Then, leverage that diagnosis information (in addition to the available MaGC forecasts), to decide how best to balance the workload distribution within a cluster.

Finally, the GC prediction capabilities of TRINI (i.e. the MaGa forecast algorithm) have been only exploited within the load balancing domain. However, they might be applicable to other domains and usage scenarios. A particularly interesting idea to explore is to investigate how the GC predictions can be useful to improve the actual GC triggering within a JVM. This is because, by leveraging on the predictions made by the MaGa algorithm, the triggering decision could take into consideration the dynamic GC/memory behaviour of the underlying application, instead of relying on a static threshold of the available free memory (strategy which is currently used by all commercial JVMs).
Bibliography


