TRINI: an adaptive load balancing strategy based on garbage collection for clustered Java systems

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

▶ **To cite this version:**

A Omar Portillo-Dominguez, Philip Perry, Damien Magoni, Miao Wang, John Murphy. TRINI: an adaptive load balancing strategy based on garbage collection for clustered Java systems. Software: Practice and Experience, 2016, 46, pp.1705-1733. 10.1002/spe.2391 . hal-01436430

HAL Id: hal-01436430
https://hal.science/hal-01436430v1

Submitted on 16 Jan 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

TRINI: An Adaptive Load Balancing Strategy Based on Garbage Collection for Clustered Java Systems

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

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

SUMMARY

Nowadays, clustered environments are commonly used in high-performance computing and enterprise-level applications to achieve faster response time and higher throughput than single machine environments. Nevertheless, how to effectively manage the workloads in these clusters has become a new challenge. As a load balancer is typically used to distribute the workload among the cluster's nodes, multiple research efforts have concentrated on enhancing the capabilities of load balancers. Our previous work presented a novel adaptive load balancing strategy (TRINI) which improves the performance of a clustered Java system by avoiding the performance impacts of Major Garbage Collection, which is an important cause of performance degradation in Java. The aim of this paper is to strengthen the validation of TRINI by extending its experimental evaluation in terms of generality, scalability and reliability. Our results have shown 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.

KEY WORDS: Load Balancing; Cluster Computing; Garbage Collection; Java; System Performance

1. INTRODUCTION

In recent years, cluster computing has gained popularity as a powerful and cost-effective solution for parallel and distributed processing [1]. 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.

One of the most important challenges in cluster computing is how to effectively distribute the workload among the available clustered instances (as load imbalance can lead to processing inefficiencies [2]). To address this challenge, multiple research efforts have aimed to develop more effective load balancing algorithms and strategies, based on different criteria and heuristics [3–5].

With an estimated business impact of a hundred billion dollars yearly, Java is a predominant player at enterprise level [6]. Therefore, this technology is commonly used to build clustered systems. A particular area of performance concern in Java is the Garbage Collection (GC) [7]. Even though it is a key feature of Java 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 [7].

Multiple research works have given evidence of the GC performance costs. For instance, the authors of [8] 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 [9] 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 [10, 11]. For example, the authors of [12] 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.

This discussion motivates the core research question here: "What techniques can be deployed so that the occurrence of MaGC events in the application nodes does not affect the performance of the cluster?". To address this challenge, our work has centred on enhancing a load balancer so that it selects the nodes which are not expected to have a MaGC event in the immediate future. This strategy can therefore help to avoid impacts in the cluster's performance due to MaGC events. In our previous work [13], we presented TRINI (shown in Figure 1), an adaptive GC-aware load balancing strategy which automatically self-configures based on the GC characteristics of a clustered application (typically located within a data centre). Internally, TRINI leverages on MaGC forecasts to decide on the best way to balance the workload among the application nodes.

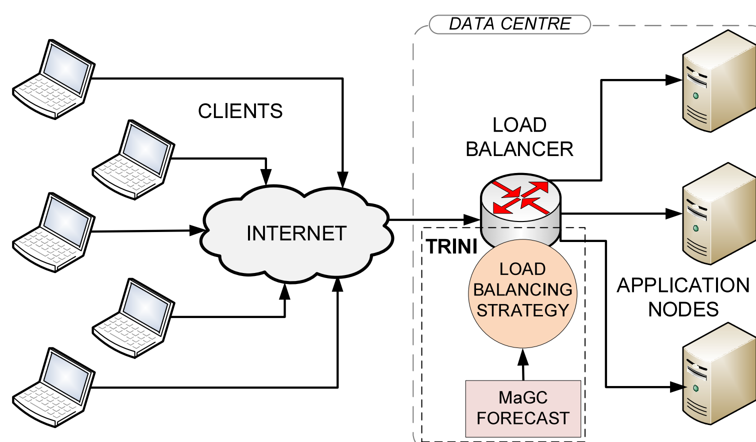

Figure 1. TRINI: A GC-Aware Load Balancing Strategy

The behaviour of a load balancing strategy is heavily influenced by the accuracy of its balancing decisions and the amount of resources it uses [14]. A deep understanding of these factors is key to comprehend the practicability of any load balancing strategy. Therefore, the aim of this

paper is to perform an exhaustive assessment of TRINI in terms of generality, scalability and reliability. For this purpose, three experiments were performed: Firstly, TRINI was applied to four load balancing algorithms to assess its generality (as it had previously been applied to only one). Secondly, TRINI was tested in ten cluster sizes to assess how scalable it is (as it had previously been tested in only one). Thirdly, TRINI was evaluated in 24-hour test runs to assess its reliability (as it had previously been tested in 1-hour runs). The obtained results showed that TRINI can offer significant performance gains under the above conditions, while only introducing a small overhead to the cluster.

The contributions of this paper are:

1. An extended description of our adaptive GC-aware load balancing strategy (TRINI), whose goal is to improve the performance of clustered Java systems.
2. A comprehensive practical evaluation of TRINI, consisting of a prototype and three 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.
3. Four GC-aware load balancing algorithms which were modified to use MaGC forecast information for improving their workload distribution.
4. 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.

The remainder of the paper is organized as follows: Sections 2 and 3 present the relevant background and related work, respectively. Section 4 explains the internal workings of TRINI; while Section 5 discusses the performed experiments and their results. Finally, Section 6 draws the conclusions of this work and provides pointers to our future work.

2. BACKGROUND

This section recalls the main features and characteristics of the GC process in Java, as well as a typical load balancing process, which are necessary to understand the rest of the paper.

GC. This form of automatic memory management offers significant software engineering benefits over 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 [15], as well as increasing the developers' productivity [16]. Despite these advantages, it is widely accepted that the GC comes with a performance cost (as discussed in Section 1).

Additionally, it is not possible to programmatically force the execution of the GC [17]. 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 [18] because the JVM usually does a better job on deciding when to do GC.

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* [19], where objects are segregated by age into memory regions known as generations. New objects are created in the youngest 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 are 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 [7]. Finally, running out of free memory in a generation triggers its respective type of GC event.

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 [20] 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 throughput is more important than 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 response time is more important than throughput).

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 nodes, each one having an identical code image of the application. The application must be also partitionable into smaller grain-sized tasks (e.g., a web application, which is normally composed of atomic operations such as login, search, etc.).

The range of existing load balancing algorithms is broad [21]. 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 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.

3. RELATED WORK

In this section, we first expand on the recent work in GC optimisation. Then, we discuss the related work in the area of memory forecasting. Next, we review the state-of-the art work in distributed systems optimisation, with a special emphasis on load balancing.

GC Optimisation. Multiple research efforts have focused on improving the GC performance. For example, several works have proposed new concurrent [22] and parallel algorithms [23] that have smaller impacts on the performance of the applications. Other works have aimed to develop algorithms that might have predictable GC performance [24]. However this predictability comes in terms of soft-requirements, meaning that the GC might still take more time than expected. Another explored approach has been to develop algorithms for specific usage scenarios. For instance, [25] describes an algorithm suitable to Java Application Servers which exploits the different natures of the local and remote objects. 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 affect its performance (as discussed in Section 1).

Memory Forecasting. This is another active research area which focuses on the self-improvement of the JVM, looking for ways to invoke a GC when it is worthwhile. For example, the work presented in [26] exploits the observation that dead objects tend to group together to estimate how much space would be reclaimable for a MaGC to avoid low-yield GCs. Meanwhile, the authors of [27] 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 memory gains) to execute a GC. However, they do not provide enough information to know when the next MaGC would occur. In contrast, our work aims to forecast the MaGC events, also making this information available outside the JVM so that other actors (such as a load balancer) could leverage it and take more informed decisions.

Distributed Systems Optimisation. Research has also focused on the optimisation of distributed architectures, improving them from various viewpoints. For example, the authors of [28] 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 [29], 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 [30] 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 [31–35]. For example, the authors of [4] 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 [31] proposed an adaptive

load balancing strategy which aims to fulfil service level agreements based on a set of customer priorities. Likewise, the authors of [36] presented an agent-based solution to provide dynamic load balancing capabilities to cloud-based services and resources. Finally, other research efforts have focused on Java technologies: The authors of [3] developed a load balancing algorithm for 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 [5] 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, our research work has enhanced 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.

4. TRINI: AN ADAPTIVE GC-AWARE LOAD BALANCING STRATEGY

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

4.1. TRINI Overview

The objective of our research work was to define a GC-aware load balancing strategy (TRINI) which is able to dynamically adjust to the specific GC characteristics of the underlying application. This strategy would allow the load balancer to forecast the occurrence of the MaGC events with enough accuracy to exploit that information for improving the performance of a cluster.

In Figure 2, we depict the conceptual view of our solution. 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.

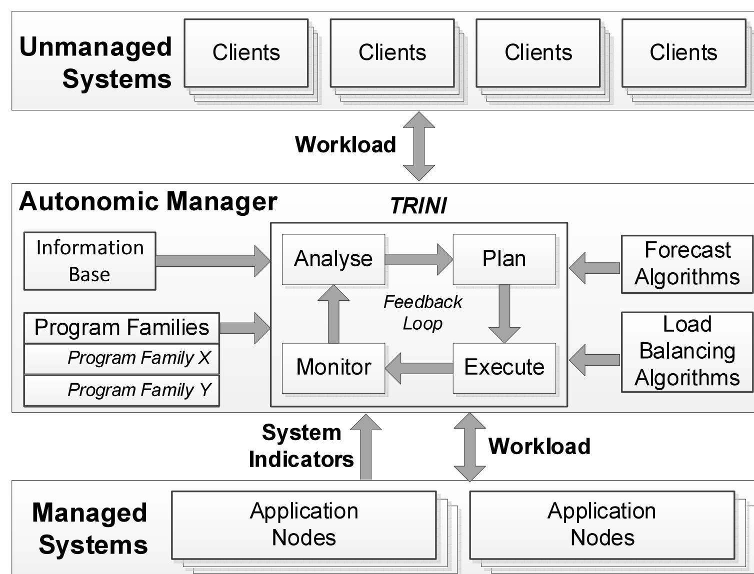

Figure 2. TRINI Conceptual View

As defined by multiple authors [37], 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 our context, it means minimising the performance impacts of the GC within the cluster. To incorporate self-adaptation to TRINI, we have followed the well-known MAPE-K adaptive model [38]. It is composed of the following elements (shown in Figure 2): A *Monitoring* element to obtain information from the managed systems; an *Analysis* element to evaluate if any adaptation is required; an element to *Plan* the adaptation, and an element to *Execute* it.

The fifth element of the MAPE-K model is the *Knowledge* element, which is responsible of supporting the other elements in their respective tasks. In TRINI, this role 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 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 hundreds 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 our 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 our 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.

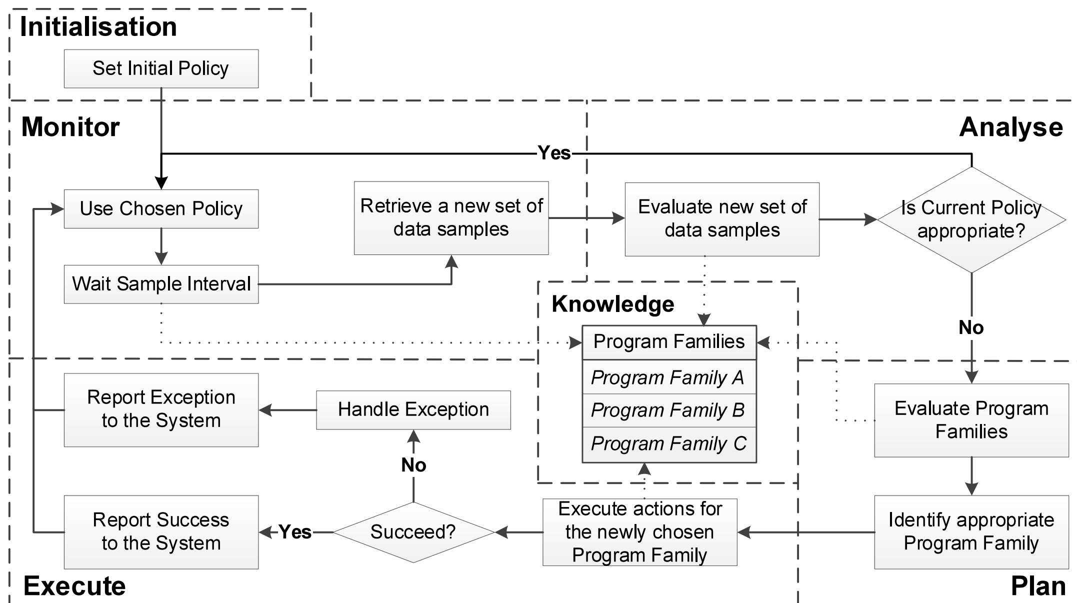

Figure 3. TRINI - Core Process

4.2. *TRINI Core Process*

TRINI has a core process which coordinates its MAPE-K elements. This process (depicted in Figure 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 their configurations from the database of program families (represented as dashed arrows in Figure 3). 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, in [39] we presented MaGA, which is an algorithm to forecast MaGC events in generational heaps. It works by periodically retrieving GC and memory samples from a monitored JVM (as per a configurable *sample interval*) to build the history of memory allocations (MemAlloc) that occur in the Young and Old Generations through time. Then, the algorithm uses the most recent historical data, as delimited by a configurable *Forecast Windows Size* (FWS), to forecast the next MaGC event. This is done in two steps. Firstly, the algorithm forecasts how much memory allocation needs to occur in the Young Generation (YoungGen) before the memory in the Old Generation (OldGen) gets exhausted (hence triggering a MaGC). An example of this process is shown in Figure 4. Firstly, the algorithm uses the OldGen historical data within the FWS (represented as a rectangle) to feed a linear regression model (LRM). This is done to predict the rate of increase in the YoungGen, as a function of the OldGen, and thus extrapolate the data to the point where the OldGen will exceed its maximum threshold (90 MB in our example) and trigger a MaGC. This yields a prediction that the next MaGC will occur when the YoungGen reaches 225 MB in our example. Secondly, the algorithm feeds this YoungGen threshold to another LRM which extrapolates the time series of the YoungGen memory and predicts that a new MaGC event will occur at a time of 600 ms (as shown in Figure 5). This forecast process continues iteratively until the monitored application finishes or the forecast is no longer needed.

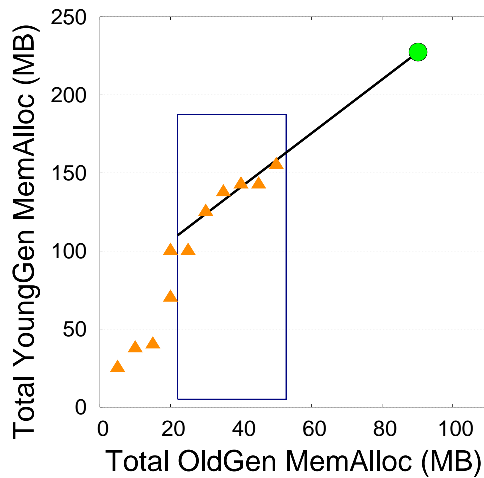

Figure 4. Forecast of Young MemAlloc

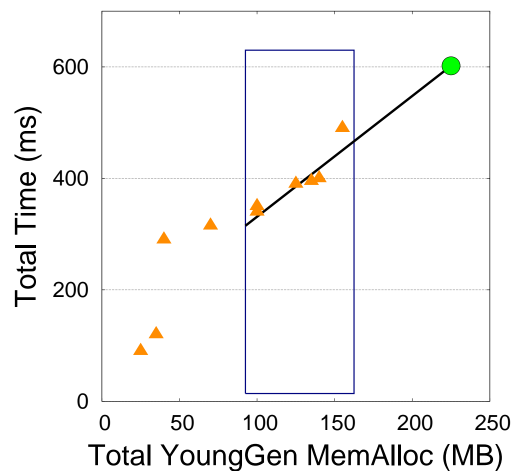

Figure 5. Forecast of MaGC Event

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, we have modified four well-known load balancing algorithms. Among the range of available algorithms, we selected the four described in Section 2: 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.

Algorithm 1: GC-WRR

```
Input:  D = {d_1, d_2, ..., d_n}, set of application nodes
        W = {w_1, w_2, ..., w_n}, set of weights per application node
        T ∈ {ℕ}, MaGC threshold
Output: d_i ∈ 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 ≥ 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) ≤ T then
18                     found := false
19                     w_i := w_i + 1
20                     i := i + 1
21                     fTries := fTries + 1
22         else
23             i := i + 1
24     use d_i for the next workload
```

The main difference of our 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 5:00:00PM, any nodes that have a MaGC predicted to occur between 5:00:00PM and 5:00:05PM 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.

An example of our proposed algorithms is presented in Algorithm 1, which shows the GC-aware weighted round robin (GC-WRR). When compared 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.

While integrating GC-awareness to the four chosen algorithms, we identified certain similarities across the performed changes. This allowed us to abstract the changes into a generic version of a GC-aware load balancing algorithm, as shown in Algorithm 2. 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. This new logic is encapsulated in the functions *IsMaGcClose*, *getForecast*, *markAsEval*, *AreNodesToEval* and *resetNodesToEval*. The function *IsMaGcClose* (shown in Algorithm 3) 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 *markAsEval* (which is responsible of marking, probably through a data structure like a hash table or a vector, the nodes after they have been evaluated for the current balancing decision). Meanwhile, the responsibility of *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 *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 *markAsEval*, *AreNodesToEval* and *resetNodesToEval*, we only describe their responsibilities, instead of specific implementations.

Algorithm 2: Abstract GC Load Balancing

```
   Input:  D = {d_1, d_2, ..., d_n}, set of application nodes
           T ∈ {ℕ}, MaGC threshold
   Output: d_i ∈ D
 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()
```

Algorithm 3: Evaluate Closeness of the MaGC

```
  Input:  d_i ∈ D, tentatively selected node
          T ∈ {ℕ}, MaGC threshold
  Output: bMaGcCloseness
1 fTime := getForecast(d_i)
2 cTime := getCurrentTime()
3 if (fTime − cTime) ≤ T then
4     bMaGcCloseness := true
5     markAsEval(d_i)
6 else
7     bMaGcCloseness := false
8 return bMaGcCloseness
```

4.5. MiGC-CV Program Families

Among the alternative strategies to develop policies for TRINI, we initially concentrated on automating the selection of the FWS. This is because our work at [39] 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.

Meanwhile, the results of our work at [13] 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 [39] (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 [13]: *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 [40] 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 us to automate the selection of an appropriate FWS on a case by case basis. The results obtained in [13] demonstrated that these functions were able to accurately predict a good percentage of the MaGC events. Those results also showed that the number of outliers tend to decrease in larger (e.g. gigabytes) heap sizes. This behaviour supported our decision of ignoring the outliers from the derived functions.

5. EXPERIMENTAL EVALUATION

This section presents the three experiments performed to assess the benefits and costs of using TRINI. Firstly, we evaluated the generality of TRINI's behaviour across a set of different load balancing algorithms. Secondly, we evaluated the scalability of TRINI's behaviour across a range of different cluster sizes. Thirdly, we evaluated the reliability of TRINI's behaviour over extended time periods. The section concludes with a discussion for practitioners where we summarise our key findings and observations.

5.1. Experiment #1: Generality Assessment

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

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

Prototype. It was built on top of the Apache Camel [41], 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 our GC-aware load balancing algorithms.

Inspired by other works [42] that have aimed to minimize the potential impacts on the monitored environment, the forecast logic was implemented external (non-intrusive) to the JVM. For this purpose, we used the Java Management Extension (JMX) [43] 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).

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 virtual machines (VM): A cluster of fifty application nodes with one load balancer, and one load tester node (as shown in Figure 6). 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 [44] (a leading open source tool used for application performance testing), and the application nodes ran an Apache Tomcat 6.0.35 [45] (a popular open source Web Application Server for Java).

The VMs were located on two Dell PowerEdge M620 servers inside a Dell PowerEdge VRTX chassis. 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.

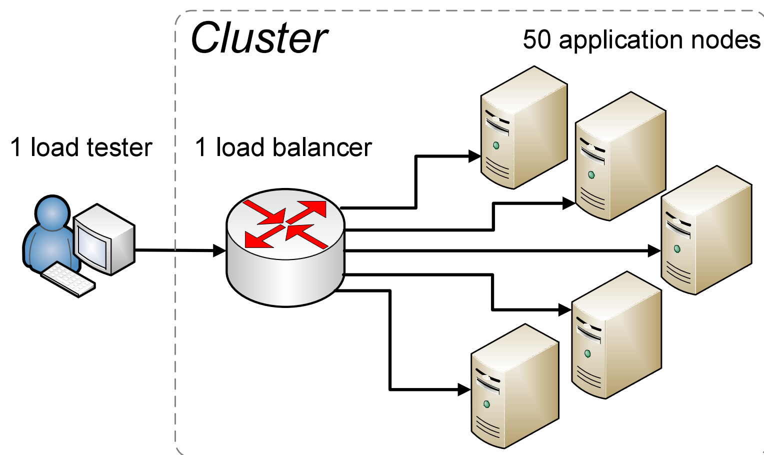

Figure 6. Test Environment

Load Balancing Algorithms. The four algorithms discussed in Section 2 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).

Benchmarks. Two of the Java benchmarks most widely-used in the literature (DaCapo 9.12 [46] and SPECJVM 2008 [47]) 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. Appendix A presents a summary of these benchmarks and their respective set of 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). 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 were used: Serial GC (sGC), Parallel GC (pGC), and Concurrent GC (cGC). This was done with the aim of diversifying more the evaluated GC behaviours (as the GC strategy is a major factor affecting the GC behaviour [12]).

Evaluation Criteria. In terms of performance, our 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, our main metrics were CPU (%) and memory (MB) utilisations. In both cases, lower values are better. These metrics were collected with the *top* command [48].

Regarding the forecast accuracy, the following three metrics were calculated:

1. The forecast error (FE) [39]. 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.
2. The average number of MiGCs that occurred between two MaGC events ($MiGC_{AVG}$) [39]. 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 [49] and [10], 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 contrary, a value far from zero (e.g., 1000 or higher) indicates that the old generation is infrequently exhausted.
3. The coefficient of variation ($MiGC_{CV}$) [39]. This metric is the standard deviation of the $MiGC_{AVG}$ expressed as a percentage of the average, and allows the comparison of different applications in terms of their variability in memory usage.

5.1.2. Experimental Results. In this section we present the results obtained from this experiment in terms of the relevant performance and overhead metrics.

Performance Improvements. As an initial step to understand the behaviour of TRINI across the evaluated experimental configurations, we focused our 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 7 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

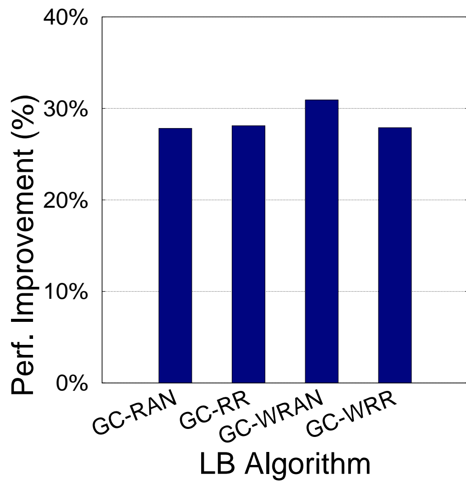

Figure 7. RT_{AVG} - Perf. Improv. per LB

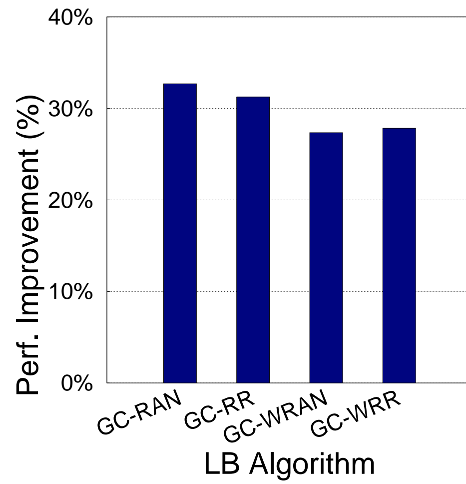

Figure 8. RT_{MAX} - Perf. Improv. per LB

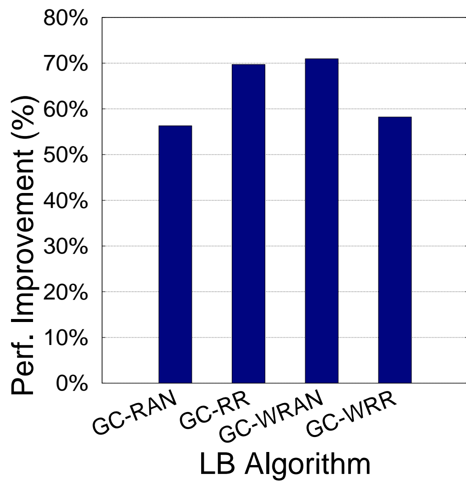

Figure 9. T_{AVG} - Perf. Improv. per LB

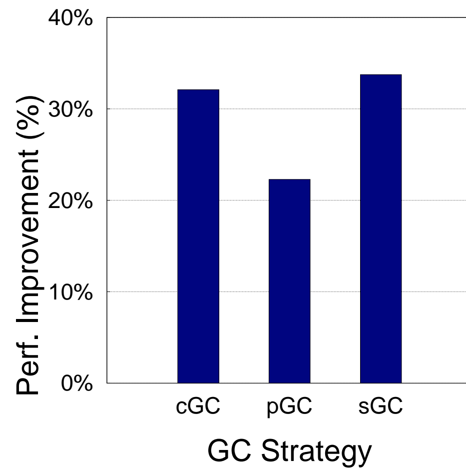

Figure 10. RT_{AVG} - Perf. Improv. per GC

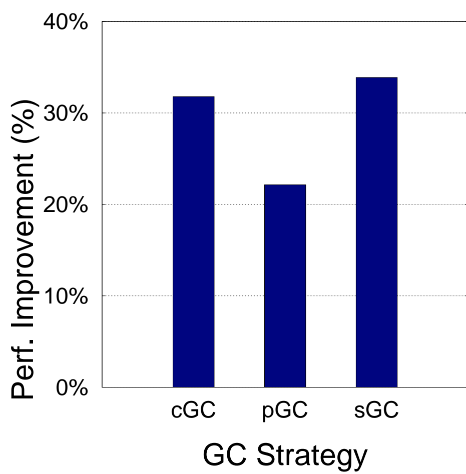

Figure 11. RT_{MAX} -Perf. Improv. per GC

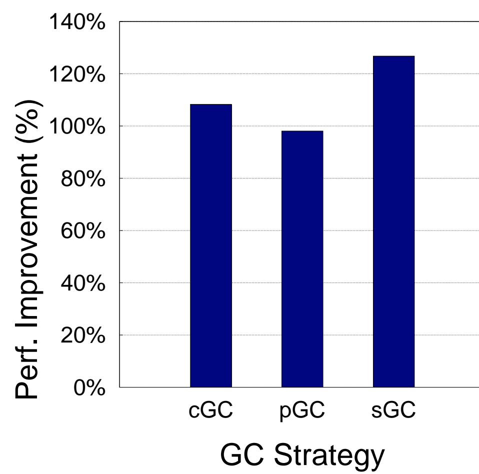

Figure 12. T_{AVG} - Perf. Improv. per GC

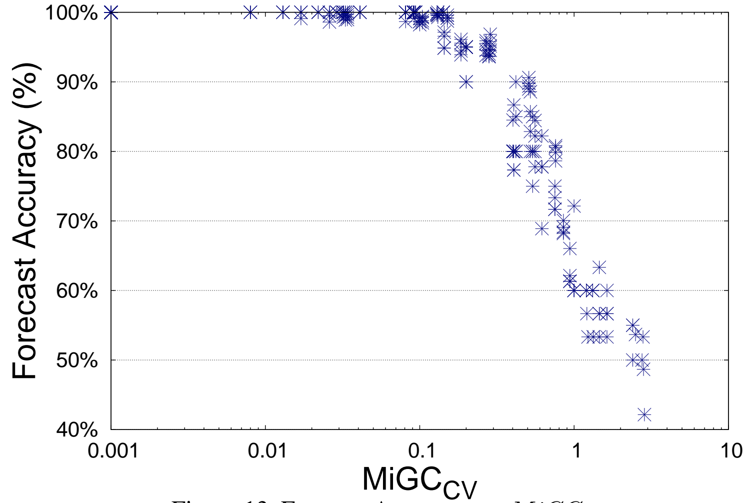

Figure 13. Forecast Accuracy vs. $MiGC_{CV}$

between 21% and 24%. Figures 8 and 9 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 our analysis focused on evaluating the sensitivity of TRINI with respect to the different GC strategies used. These results are presented in Figures 10 (RT_{AVG}), 11 (RT_{MAX}) and 12 (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 GC strategy. The biggest gains occurred when using 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 capture 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, we focused on understanding the reasons behind the achieved performance gains. For this purpose, we analysed the results in terms of $MiGC_{CV}$ behaviour, as illustrated in Figure 13. There, a clear relationship can be observed between the forecast accuracy achieved by TRINI and the $MiGC_{CV}$ 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 $MiGC_{CV}$ is an appropriate metric to characterise the program behaviours into families.

In our experiments, we 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 ($MaGC_D$), 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 $MaGC_D$ is long (as there is more potential gain to exploit). However, the actual benefits depend on the amount of $MaGC_D$ that is actually addressed (A-$MaGC_D$). This behaviour is depicted in Figure 14, 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%≤FA≤50%), *medium* (50%<FA≤80%), and *high* (FA>80%). In

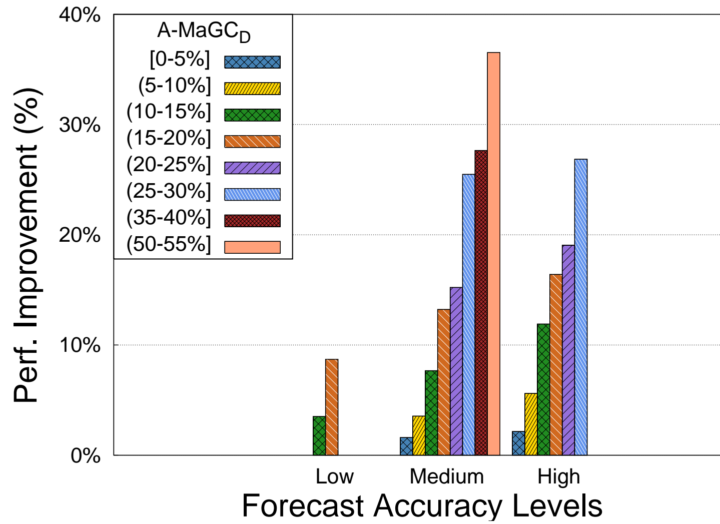

Figure 14. RT_{AVG} - Perf. Improvements per FA and A-$MaGC_D$

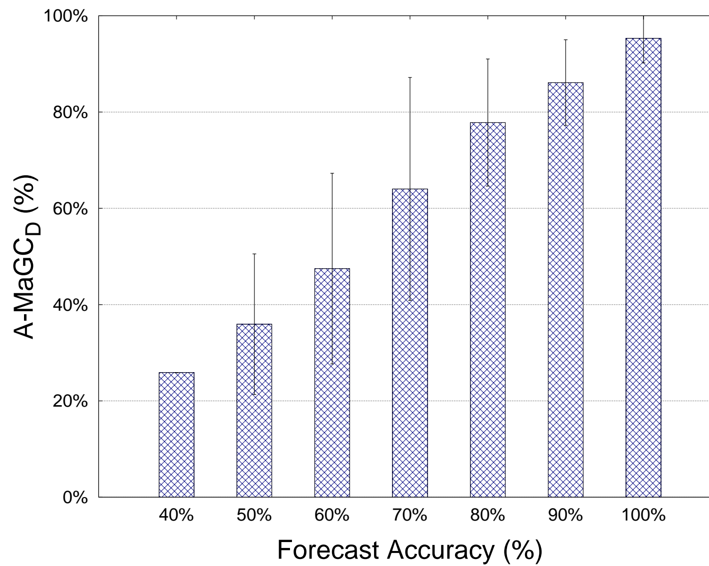

Figure 15. A-$MaGC_D$ per Forecast Accuracy Level

Figure 14, 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 15, 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

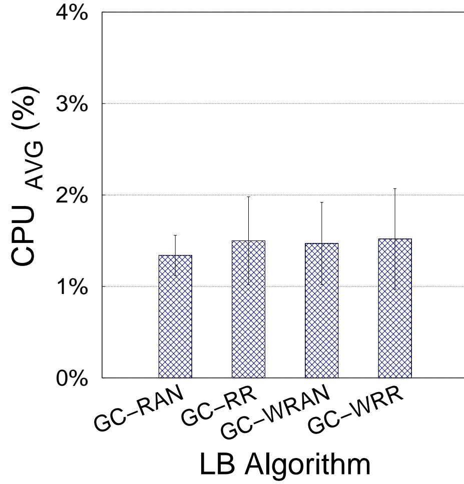

Figure 16. App.nodes - ΔCPU_{AVG}

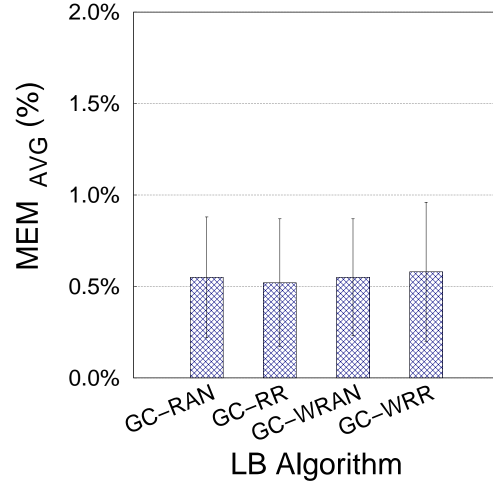

Figure 17. App.nodes - ΔMEM_{AVG}

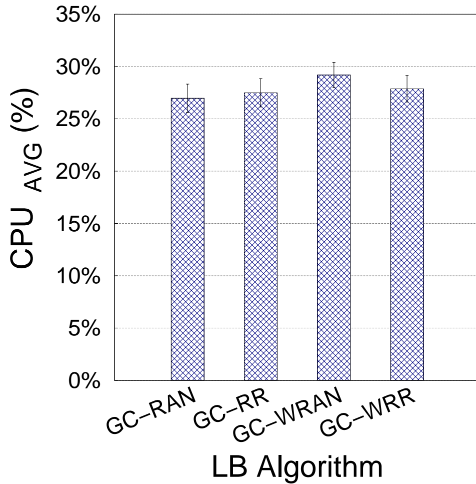

Figure 18. LB node - ΔCPU_{AVG}

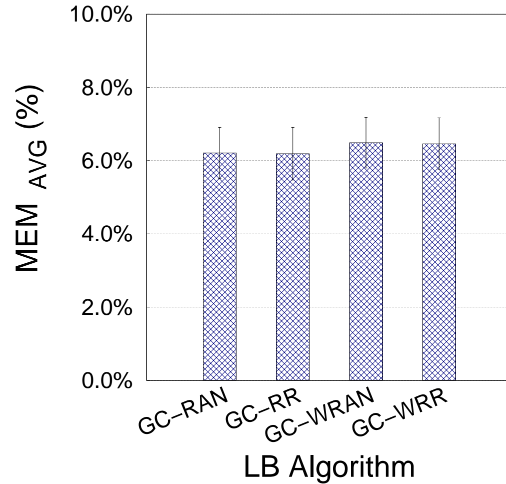

Figure 19. LB node - ΔMEM_{AVG}

that lasts two minutes, than a MaGC that lasts two seconds (even though both MaGC events are equally captured by the FA metric).

Overhead. We also studied the costs of using TRINI. For this analysis, we 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 (ΔCPU_{AVG}) across all tested applications was 1.46%, with a standard deviation of 0.43%; while the increment in average memory usage (ΔMEM_{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 5.1.1). These results are presented in Figures 16 and 17, which show the ΔCPU_{AVG} and ΔMEM_{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 ΔCPU_{AVG} was 27.88%, with a standard deviation of 1.30%; while the ΔMEM_{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 18 (ΔCPU_{AVG}) and 19 (ΔMEM_{AVG}). The ΔCPU_{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 ΔMEM_{AVG} was caused by the initialisation of TRINI. The remaining increment was due to the historical data that was kept for forecasting purposes.

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 5.1.1).

5.2. *Experiment #2: 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.

5.2.1. Experimental Set-up. The set-up was similar to that used in experiment #1 (presented in Section 5.1.1), with the following differences: The cluster size was variable, covering the range of [5..50] application nodes in increments of 5. The minimum value was the size used in our previous works [13,39], while the maximum value was constrained by our available computational resources. 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 #1 proved that TRINI works well irrespective of the load balancing algorithm, we centred on the WRR because it is currently the most widely-used load balancing algorithm [31]. Likewise, we concentrated on the Serial GC strategy because it tends to suffer the longest pauses [7], hence benefiting more from our work. Finally, we focused on 5 programs which were representative of the program classification that we presented in our previous work [13]. This configuration allowed us to test a diverse set of GC behaviours with a smaller set of experimental configurations. That classification (shown in Table I) grouped the programs of the DaCapo and SPECJVM benchmarks according to their GC characteristics (the $MaGC_D$ and the $MiGC_{CV}$). In Table I, the programs underlined are the ones used in this experiment.

Table I. DaCapo/SPEC Program Classification per GC behaviours

$MiGC_{CV}$	$MaGC_D$		
	Short	**Medium**	**Long**
Low	compiler, jython	avrora, compress, fop, luindex, lusearch, mpegaudio, tomcat, startup, sunflow, xalan	
Medium		batik, crypto, eclipse, pmd, tradebeans	h2, scimark, tradesoap, xml
High			derby, serial

5.2.2. Experimental Results. In this experiment, our 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. Our hypothesis was that the performance improvements should not degrade when the cluster size increases, as each forecast process is independent of each other

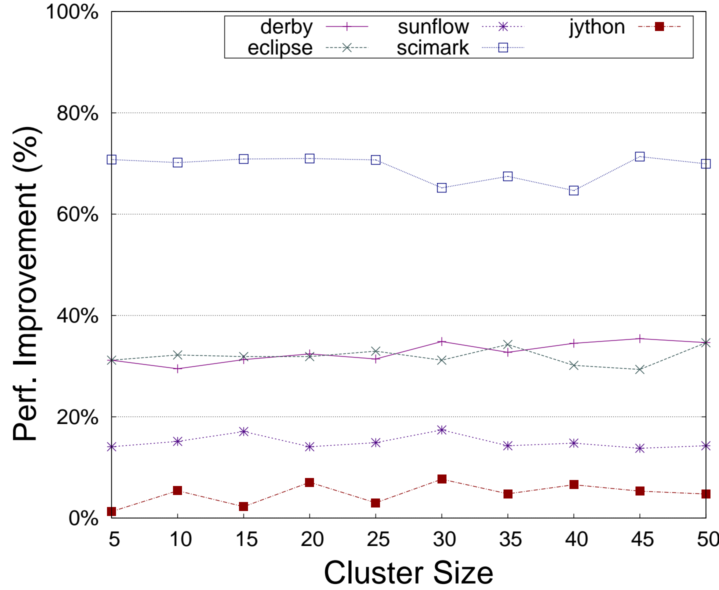

Figure 20. RT_{AVG} - Performance Improvements per cluster size

(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 variances in the percentage of performance improvements that TRINI achieved, the improvements were closely similar, across the different cluster sizes, per tested program. Figure 20 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 $MaGC_D$ and also achieved a high forecast accuracy (above 90%). On the contrary, the *jython* program obtained the smallest improvements because it suffered the shortest $MaGC_D$ (meaning it had the smallest potential gains). For the sake of brevity, we only present the improvements in RT_{AVG}. However, 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 ΔCPU_{AVG} and ΔMEM_{AVG} in the application nodes per cluster size (shown in Figures 21 and 22, 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 ΔCPU_{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 23. In the case of the ΔMEM_{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 24, which presents the ΔMEM_{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

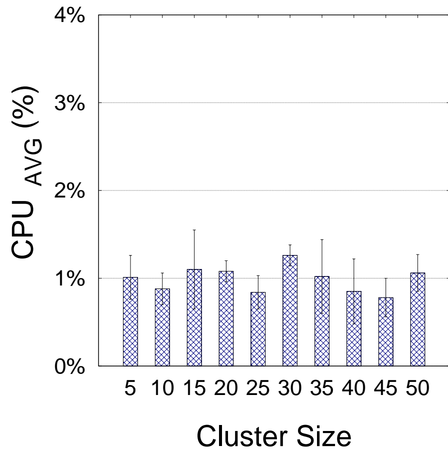

Figure 21. App.nodes - ΔCPU_{AVG} per size

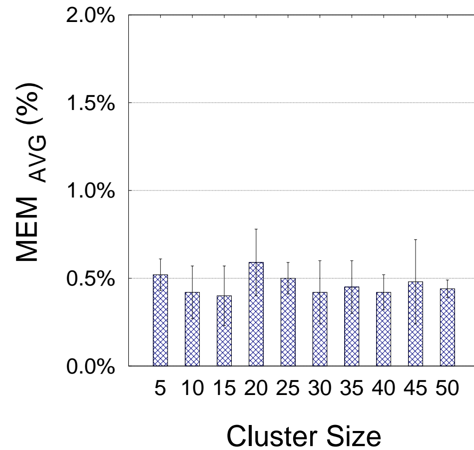

Figure 22. App.nodes - ΔMEM_{AVG} per size

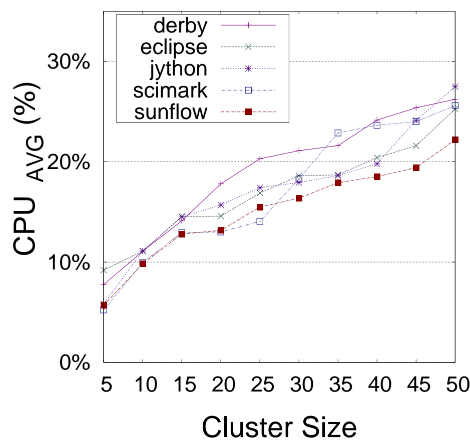

Figure 23. LB node - ΔCPU_{AVG} per size

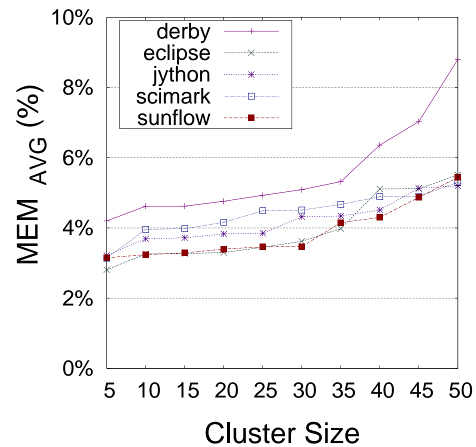

Figure 24. LB node - ΔMEM_{AVG} per size

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.

5.3. Experiment #3: 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.

5.3.1. Experimental Set-up. The set-up was similar to that used in the experiment #2 (presented in Section 5.2.1), with two differences: First, the evaluated cluster was composed of 50 application nodes (same size as experiment #1, described in Section 5.1.1). Second, the duration of the test runs was increased from 1 to 24 hours to evaluate TRINI on a longer, more realistic duration.

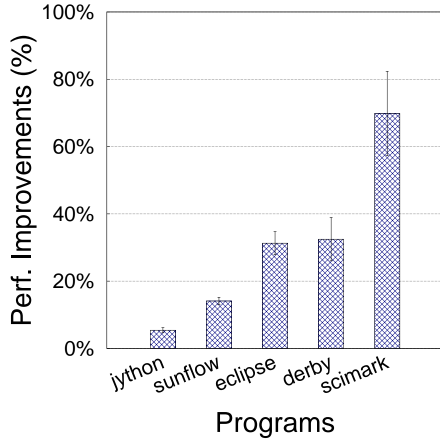

Figure 25. RT_{AVG} - Perf. Improv. on 24-hr runs

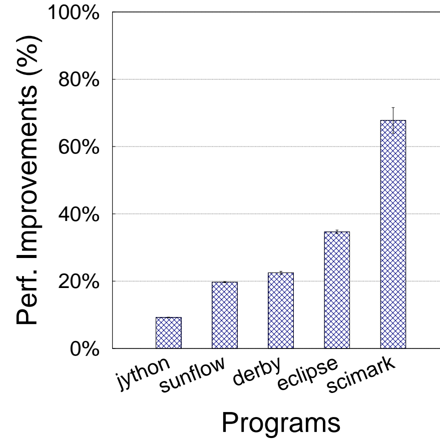

Figure 26. RT_{MAX} - Perf. Improv. on 24-hr runs

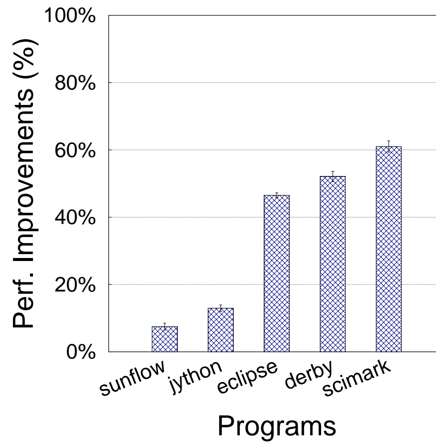

Figure 27. T_{AVG} - Perf. Improv. on 24-hr runs

5.3.2. Experimental Results. In this section we 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, we 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 25 shows the results in terms of RT_{AVG}. Similar results were obtained in terms of RT_{MAX} and T_{AVG} (as shown in Figures 26 and 27, respectively).

Overhead. The results of our 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 ΔCPU_{AVG} across all tested applications was 1.02%, with a standard deviation of 0.38%; while the ΔMEM_{AVG} across all tested applications was 0.42%, with a standard deviation of 0.15%.

In the load balancer node, the results of our analysis showed that the ΔCPU_{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 terms of memory, the ΔMEM_{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 ΔCPU_{AVG} during the whole test runs. Similar behaviour was observed in terms of ΔMEM_{AVG} in the application nodes. Finally, TRINI experienced a minimum increase in terms of ΔMEM_{AVG} in the load balancer node. This was due to the historical data that TRINI temporarily preserved for forecasting purposes.

5.4. Final 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 we 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 remains within reasonable levels. For instance, in our experiments, the programs which experienced the highest variability ($MiGC_{CV} \geq 1.0$) 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, our results have shown that even a medium level of forecast accuracy (50%<FA≤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 our experimental evaluation, we selected three of the most widely-used GC strategies in the industry. As our 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 our experimental evaluation, we selected four of the most frequently used load balancing algorithms in the industry. As our results have 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, we 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, our 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 our experiments we 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, we recommend 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, our results have shown that the overhead usually follows a relatively linear growth with respect to the cluster size. For this reason, our 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, we conclude 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 $MiGC_{CV}$ program families) and which can be leveraged to make a more robust GC-aware load balancing solution.

6. CONCLUSIONS AND FUTURE WORK

One of the most important challenges in cluster computing is how to efficiently distribute the workload among the cluster's nodes. To address this challenge, in our previous work we 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). The aim of this paper was to comprehensively evaluate TRINI in terms of generality, scalability and reliability in order to offer practitioners a valuable reference regarding the behaviour of TRINI in such circumstances. For this purpose, three experiments were performed. Firstly, TRINI was applied to four commonly used load balancing algorithms to assess the generality of its performance improvements and overheads. Secondly, TRINI was evaluated in different sizes of clusters to assess how scalable our solution was. Thirdly, TRINI was evaluated in 24-hour test runs to assess its reliability over extended time periods.

Our experimental results have demonstrated that TRINI can significantly improve the response time and throughput of a cluster. 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. Additionally, the overhead in the load balancer was low, especially considering

the modest characteristics of the used load balancer node. From the above results, we conclude that a GC-aware load balance strategy can bring significant benefits to a clustered Java system.

In our future work, we will continue investigating which other GC characteristics might be suitable in order to deepen our classification of program behaviours into families. We plan to use this additional knowledge to develop more portable load balancing policies. We will also explore how to extend TRINI to address other types of performance issues and build a more sophisticated load balancing solution. As a first step in that direction, we plan to explore the feasibility of using the outputs of performance diagnosis tools (e.g., the IBM WAIT [50]) to monitor the health of the application nodes. Then, leverage that information (in addition to the MaGC forecasts), to decide how best to balance the workload distribution within a cluster.

ACKNOWLEDGEMENTS

This work was supported, in part, by Science Foundation Ireland grant 10/CE/I1855 to Lero - the Irish Software Engineering Research Centre (www.lero.ie). The authors thank the editor and the anonymous reviewers for their helpful comments and suggestions. Likewise, the authors appreciate the many discussions about this paper with Vanessa Ayala-Rivera.

REFERENCES

1. Lee WY, Hong SJ, Kim J, Lee S. Dynamic load balancing for switch-based networks. *Journal of Parallel and Distributed Computing* 2003; **63**(3):286–298.
2. Bahi J, Couturier R, Vernier F. Synchronous distributed load balancing on dynamic networks. *Journal of Parallel and Distributed Computing* 2005; **65**(11):1397–1405.
3. Carmona AB, Roca-piera J, Capel CH, Álvarez bermejo JA. Adaptive Load Balancing between Static and Dynamic Layers in J2EE Applications. *Next Generation Web Services Practices* 2011; :61–66.
4. Rupprecht L, Reiser A, Kemper A. Dynamic load balancing in data grids by global load estimation. *International Symposium on Parallel and Distributed Computing* 2012; :243–250.
5. Liu Y, Wang L, Li S. Research on self-adaptive load balancing in EJB clustering system. *Intelligent System and Knowledge Engineering* 2008; :1388–1392.
6. Java: 2.5 Years After the Acquisition. *International Data Corporation (2012)* 2012; .
7. Memory Management in the Java HotSpot Virtual Machine. *Sun Microsystems (2006)* 2006; .
8. Xian F, Srisa-an W, Jiang H, Hall A. Garbage Collection : Java Application Servers' Achilles Heel. *Science of Computer Programming* Feb 2008; **70**(2):89–110.
9. Snatzke RG. Performance survey. *Codecentric AG (2009)* 2009; .
10. Mao F, Zhang EZ, Shen X. Influence of program inputs on the selection of garbage collectors. *SIGPLAN Virtual Execution Environments* 2009; :91–100.
11. Lengauer P, Mössenböck H. The taming of the shrew: increasing performance by automatic parameter tuning for java garbage collectors. *International Conference on Performance Engineering* 2014; :111–122.
12. Blackburn SM, Cheng P, Mckinley KS. Myths and Realities: The Performance Impact of Garbage Collection. *SIGMETRICS Performance Evaluation Review* 2004; **32**(1):25–36.
13. Portillo-Dominguez AO, Wang M, Murphy J, Magoni D. Adaptive gc-aware load balancing strategy for high-assurance java distributed systems. *International Symposium on High Assurance Systems Engineering*, IEEE, 2015; 68–75.
14. Willebeek-LeMair MH, Reeves AP. Strategies for dynamic load balancing on highly parallel computers. *IEEE Transactions on Parallel and Distributed Systems* 1993; **4**(9):979–993.
15. Wilson PR. Uniprocessor Garbage Collection Techniques. *International Workshop of Memory Management*, 1992; 1–42.
16. Phipps G. Comparing Observed Bug and Productivity Rates for Java and C++. *Software Practice and Experience* 1999; **29**(4):345–358.
17. Manning W. Scjp sun certified programmer for java 6 exam. *Emereo Pty Ltd, London* 2009; .
18. Developing Java Applications. URL `http://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/devapps/codeprac.html`, last accessed: 2015-11-10.
19. Memory Management in the Java HotSpot Virtual Machine. URL `http://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf`, last accessed: 2015-11-10.
20. Java SE 6 HotSpot Virtual Machine Garbage Collection Tuning. URL `http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html`, last accessed: 2015-11-10.
21. Beniwal P, Garg A. A comparative study of static and dynamic load balancing algorithms. *International Journal of Advance Research in Computer Science and Management Studies* 2014; **2**(12):1–7.
22. Pizlo F, Petrank E, Steensgaard B. A study of concurrent real-time garbage collectors. *ACM SIGPLAN Notices* 2008; **43**(6):33–44.
23. Siebert F. Limits of parallel Garbage Collection. *International Symposium of Memory Management* 2008; :21–29.

24. Kalibera T. Replicating real-time Garbage Collection for Java. *International Workshop on Java Technologies for Real-Time and Embedded Systems* 2009; :100–109.
25. Xian F, Srisa-an W, Jia C, Jiang H. AS-GC : An Efficient Generational Garbage Collector for Java Application Servers. *European Conference on Object-Oriented Programming*, 2007.
26. Wegiel M, Krintz C. Dynamic prediction of collection yield for managed runtimes. *SIGPLAN Notices* Feb 2009; **44**(3):289–300.
27. Xian F, Srisa-an W, Jiang H. Fortune Teller: Improving Garbage Collection Performance in Server Environment using Live Objects Prediction. *Object-Oriented Programming, Systems, Languages, and Applications* 2005; :246–248.
28. Mateos C, Zunino A, Campo M. m-jgrim: a novel middleware for gridifying java applications into mobile grid services. *Software: Practice and Experience* 2010; **40**(4):331–362.
29. Aghdaie N, Tamir Y. Coral: A transparent fault-tolerant web service. *Journal of Systems and Software* 2009; **82**(1):131–143.
30. Kalogeraki V, Melliar-Smith P, Moser LE, Drougas Y. Resource management using multiple feedback loops in soft real-time distributed object systems. *Journal of Systems and Software* 2008; **81**(7):1144–1162.
31. Boone B, Van Hoecke S, Van Seghbroeck G, Joncheere N, Jonckers V, De Turck F, Develder C, Dhoedt B. Salsa: Qos-aware load balancing for autonomous service brokering. *Journal of Systems and Software* 2010; **83**(3):446–456.
32. Hui CC, Chanson ST. Flexible and extensible load balancing. *Software: Practice and Experience* 1997; **27**(11):1283–1306.
33. Ho KS, Leong HV. Improving the scalability of the corba event service with a multi-agent load balancing algorithm. *Software: Practice and Experience* 2002; **32**(5):417–441.
34. Sanghi D, Jalote P, Agarwal P, Jain N, Bose S. A testbed for performance evaluation of load-balancing strategies for web server systems. *Software: Practice and Experience* 2004; **34**(4):339–353.
35. Chae HS, Park JG, Cui JF, Lee JS. An adaptive load balancing management technique for rfid middleware systems. *Software: Practice and Experience* 2010; **40**(6):485–506.
36. Mohamed N, Al-Jaroodi J. Midcloud: an agent-based middleware for effective utilization of replicated cloud services. *Software: Practice and Experience* 2015; **45**(3):343–363.
37. Weyns, Danny J M Usman Iftikhar. Do external feedback loops improve the design of self-adaptive systems? a controlled experiment. *International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, 2013; 3–12.
38. Kephart JO, Chess DM. The vision of autonomic computing. *Computer* 2003; **36**(1):41–50.
39. Portillo-Dominguez AO, Wang M, Magoni D, Perry P, Murphy J. Load balancing of java applications by forecasting garbage collections. *International Symposium on Parallel and Distributed Computing*, 2014; 127–134.
40. Coefficient of Determination (R2). URL `http://www.businessdictionary.com/definition/coefficient-of-determination-r2.html`, last accessed: 2015-11-10.
41. Apache Camel. URL `http://camel.apache.org/`, last accessed: 2015-11-10.
42. Altman E, Arnold M, Fink S, Mitchell N. Performance analysis of idle programs. *SIGPLAN Notices* 2010; **45**(10):739–753.
43. Java Management Extensions (JMX) Technology. URL `http://www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html`, last accessed: 2015-11-10.
44. Apache JMeter. URL `http://jmeter.apache.org/`, last accessed: 2015-11-10.
45. Apache Tomcat. URL `http://tomcat.apache.org/`, last accessed: 2015-11-10.
46. The DaCapo Benchmark Suite. URL `http://dacapobench.org/`, last accessed: 2015-11-10.
47. SPECjvm 2008. URL `http://www.spec.org/jvm2008/`, last accessed: 2015-11-10.
48. Linux Ubuntu Manual - Top Command. URL `http://manpages.ubuntu.com/manpages/oneiric/man1/top.1.html`, last accessed: 2015-11-10.
49. Singer J, Jones RE, Brown G, Luján M. The economics of garbage collection. *International Symposium on Memory Management*, 2010; 103–112.
50. IBM WAIT Tool. URL `https://wait.ibm.com/`, last accessed: 2015-11-10.

APPENDIX A: 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 this paper.

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 [46]. Table II lists these programs and briefly describes their functionality.

Table II. DaCapo Programs

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

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 [47]. Table III lists these programs and briefly describes their functionality.

Table III. SPECJVM Programs

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