2015 IEEE 16th International Symposium on High Assurance Systems Engineering

Adaptive GC-Aware Load Balancing Strategy for High-Assurance Java Distributed Systems

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

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

***Abstract*—High-Assurance applications usually require achieving fast response time and high throughput on a constant basis. To fulfil these stringent quality of service requirements, these applications are commonly deployed in clustered instances. However, how to effectively manage these clusters has become a new challenge. A common approach is to deploy a front-end load balancer to optimise the workload distribution among the clustered applications. Thus, researchers have been studying how to improve the effectiveness of a load balancer. Our previous work presented a novel load balancing strategy which improves the performance of a distributed Java system by avoiding the performance impacts of Major Garbage Collection, which is a common cause of performance degradation in Java applications. However, as that strategy used a static configuration, it could only improve the performance of a system if the strategy was configured with domain expert knowledge. This paper extends our previous work by presenting an adaptive GC-aware load balancing strategy which self-configures according to the GC characteristics of the application. Our results have shown that this adaptive strategy can achieve higher throughput and lower response time, compared to the round-robin load balancing, while also avoiding the burden of manual tuning.**

***Keywords*—High-Assurance Systems, Performance and Reliability, Java Garbage Collection**

I. Introduction

Modern high-assurance applications are commonly deployed in clustered instances to achieve higher system performance. This is particularly common in enterprise-level applications which usually have tight performance requirements, requiring to achieve fast response time and high throughput constantly to meet their service level agreements. These clusters also make wide use of some forms of load balancing to optimise their performance. However, how to effectively distribute the workload among the available clustered instances has become a new challenge. Therefore, many research efforts have aimed to develop effective load balancing algorithms based on different criteria. For example, the authors of [4] enhanced a load balancing algorithm by considering the utilisation of the Java Virtual Machine (JVM) threads, memory and CPU to decide how to distribute the load. Similarly, the work in [10] considered the utilisation of Enterprise JavaBeans (EJB) to balance the load among the available EJB instances.

In Java applications, a particular area of performance concern is Garbage Collection (GC). Even though GC is a key feature of Java which automates most of the tasks related to memory management, it also comes with a cost: Whenever the GC is triggered, it has an impact on the system performance by pausing the involved programs. Research studies have provided evidence of these performance costs. For example, the authors of [26] identified the GC as a major factor affecting the throughput of Java Application Servers (a traditional Java business niche) due to the sensitivity of the GC to the workload. In their experiments, the GC took up to 50% of the JVM execution time (involving pauses as high as 300 seconds), and the Major Garbage Collection (MaGC) represented more than 95% of those pauses on the heaviest workload. Similarly, a survey conducted among practitioners [20] reports that GC is one of the typical areas of performance problems 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 of the application inputs and the system configuration [7], [9], [12]. For example, the authors of [3] showed that the GC is particularly sensitive to the heap size and even small changes could affect the GC behaviour. Due to the multiple environmental factors (such as increases in workload, or non-ideal tuning settings) which can still provoke long MaGC pauses, it is commonly agreed that GC keeps playing an important role in the performance of Java systems.

Under the above conditions, a preferable situation is that the occurrence of the MaGC events do not affect the performance of the cluster. To achieve this goal, our research has centred on enhancing a load balancer so that it selects the nodes which are not close to suffer a MaGC pause as optimal nodes for given workloads. This strategy (shown in Figure 1) can keep the system performance safe from MaGC pauses.

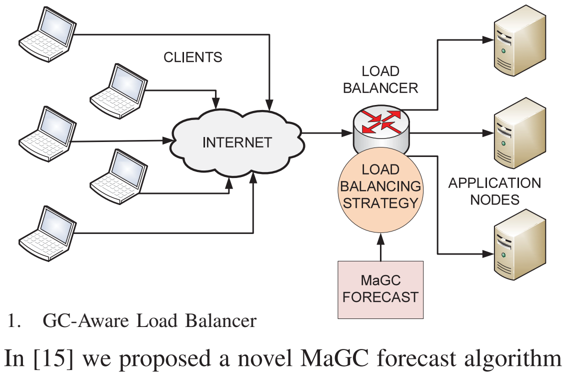

Fig. 1. GC-Aware Load Balancer

In [15] we proposed a novel MaGC forecast algorithm and demonstrated how a load balancing strategy can exploit the MaGC forecast to improve the performance of a distributed system. However, as that strategy uses a static configuration, it mostly relies on human expert knowledge to optimise its

978-1-4799-8111-3/15 $31.00 © 2015 IEEE
DOI 10.1109/HASE.2015.19

settings to achieve the best system performance. In this paper we extend our previous work by proposing an adaptive GC-aware load balancing strategy which utilises JVM data from the underlying application to self-configure according to the GC characteristics of the application, while also using the MaGC forecast to decide on the best way to balance the workload among the clustered applications. Our results showed that this strategy can offer a significant performance gain: The average response time decreased 22.5% across all tested programs, while the average throughput increased 60.7% across all tested programs. The contributions of this paper are:

1) An adaptive GC-aware load balancing strategy that exploits the MaGC forecast information to improve the performance of a distributed Java system.
2) A practical validation of the proposed strategy consisting of a prototype and two experiments to prove the performance benefits of using the strategy.
3) A classification of the programs in the evaluated Java benchmarks based on their GC characteristics.

The rest of the paper is structured as follows: Section II discusses the background and related work. Section III explains the proposed load balancing strategy. Sections IV and V describe the performed experiments and their results. Finally, Section VI shows the conclusions and future work.

II. Background and Related Work

Memory Management in Java. GC is a core feature of Java which offers significant software engineering benefits over explicit memory management. For example, it frees the programmers from the burden of manual memory management, preventing the most common sources of memory leaks and overwrites [24]. Despite its advantages, the GC comes with a performance cost (as discussed in Section I). It is not possible to programmatically force the execution of a GC [11], only to suggest the JVM to execute it by calling the method *System.gc()* (or *Runtime.getRuntime().gc()*). However, the JVM is not forced to fulfil this request and may choose to ignore it. Moreover, the usage of these methods is discouraged by the JVM vendors[1] because the JVM usually does a much better job on deciding when to do GC.

Nowadays one of the most commonly used heap types in Java is the generational heap[2], where objects are segregated by age into memory regions called generations. The survival rates of younger generations are usually lower than those of older ones, so younger generations are more likely to contain garbage and can be collected more frequently. 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 responsible of moving the live objects which have become old enough to the older generations. The GC in the older generations is known as Major GC (MaGC) and is commonly accepted as the most expensive GC type due to its performance impact [13].

GC Optimisation. Multiple research efforts have focused on improving the performance of the GC: Several works have proposed new parallel [2], [18] and concurrent algorithms [14], [21] that have smaller impacts on the performance of the applications. Another approach has been to develop algorithms that might have predictable GC performance [8]. However this predictability comes in terms of soft-requirements, meaning that the GC might still take more time than expected. Even though all these works have helped to reduce the frequency and impact of GC, it remains a performance concern due to the different factors that can still affect its performance.

Memory Forecasting. It is another active research area which has focused 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 [22] exploits the observation that dead objects tend to cluster together to estimate how much space would be reclaimable to avoid low-yield GCs. Meanwhile, the authors of [25] present an approach to estimate the dead objects at any time, information that a JVM could use to determine 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 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 this information.

Distributed Systems Optimisation. Research has also focused on the optimisation of distributed architectures. For example, the work at [1] proposed a statistical approach for the early detection of QoS deviations at runtime. While [5] presented an architecture for the dynamic detection of security threats in virtualised platforms. Regarding load balancing, the authors of [17] proposed a technique to estimate the global workload of a load balancer to use this information in the balancing of new workload. Meanwhile, the authors of [4] enhanced a load balancing algorithm for Java applications by considering the utilisation of the JVM threads, the heap and the CPU to decide how to distribute the load. In contrast to those works, our work has enhanced a load balancer by considering the MaGC forecasts in its decision layer. In such a case, the load balancer can get extra knowledge of the JVM to control the workload of the system in addition to existing load balancing policies based on other system resources.

MaGC Forecast Algorithm. In our previous work [15], we presented an algorithm to predict MaGC events in generational heaps. The algorithm works by periodically retrieving MiGC and memory samples from a monitored JVM (as per a configurable *Sample Interval*) to build the history of memory allocations 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 is shown in Figure 2. There, the algorithm uses the OldGen historical data within the FWS (represented as a rectangle) to feed a linear regression model (LRM) to predict the amount of memory allocation which is pending to occur in the YoungGen (450MB in our example) before the required allocation in the OldGen memory occurs (200MB in our example, assuming there are currently 100MB of free OldGen memory and 100MB have been previously

[1]http://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/-geninfo/devapps/codeprac.html

[2]http://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf

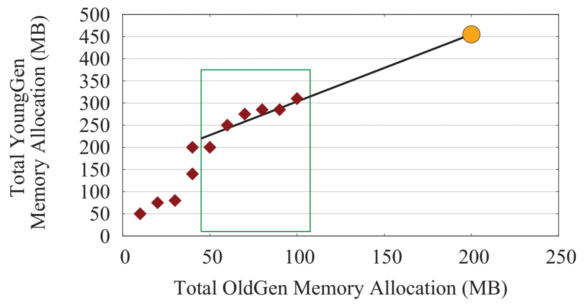

Fig. 2. Forecast of Young Memory Allocation

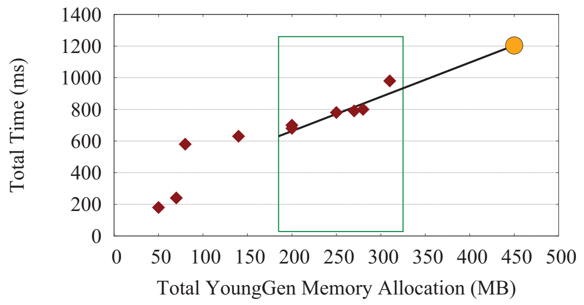

Fig. 3. Forecast of MaGC Event

allocated, as per the historical OldGen data). Secondly, the algorithm forecasts the MaGC event by feeding another LRM with the most recent YoungGen historical data and the result of the previous projection (450MB in our example). This is shown in Figure 3: By using the historical data within our FWS (represented as a rectangle), the algorithm predicts that the next MaGC will occur in the millisecond 1,200.

To assess the performance benefits of adapting the load balancing based on the MaGC forecasts, in [15] we also presented a modified version of the round-robin[3] load balancing algorithm. The main difference of our algorithm (compared against the normal round-robin) is that it performs an additional check in the selection of the next node. If the pre-selected node (as per the normal round-robin logic) would suffer a MaGC within a specified time threshold, that node is skipped and the next node is evaluated. Once the MaGC is over, the affected node is again available for selection.

III. Adaptive GC-Aware Load Balancing Strategy

The objective of this work was to define a GC-aware load balancing strategy which dynamically adjusts to the GC characteristics of the underlying application. This behaviour would allow the load balancer to forecast the occurrence of the MaGC events with enough accuracy to exploit that information and improve the performance of a distributed system, such as a clustered application.

In Figure 4, we depict the contextual view of our solution, where the adaptive GC-aware load balancing strategy periodically retrieves information from the application nodes to select the policy which is most suitable to the GC characteristics of the application running on each node (termed as *program family*). Then, 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 [23], self-adaptation enables a system to adapt itself autonomously to internal and external changes to achieve particular quality goals in the face of uncertainty. To incorporate self-adaptation to our GC-aware load balancing strategy, we have followed the well-known MAPE-K adaptive model [6]. It is composed of 5 elements (depicted in Figure 4): 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. It also has a *Knowledge* element to support the others in their respective tasks.

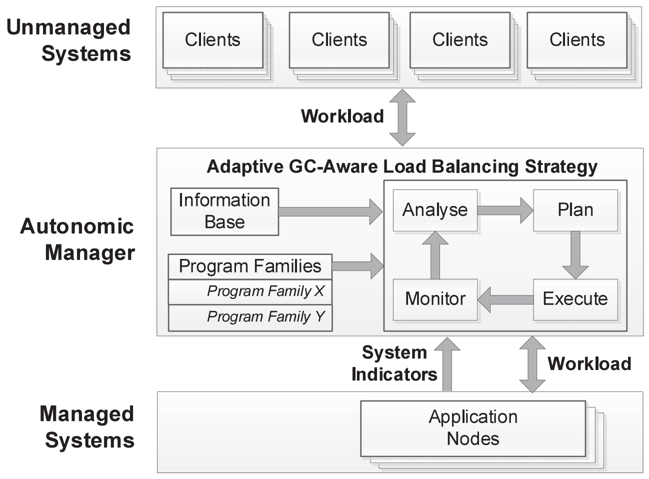

Fig. 4. Adaptive GC-Aware Load Balancing Strategy

The key element of our proposed solution is the set of identified *program families*, which fulfils the role of the *Knowledge* element. 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 based on the duration of the MaGCs so that one policy can be used for those programs which tend to suffer MaGCs of small duration (e.g. few hundreds milliseconds), as these MaGCs do not represent a major performance issue; while a different policy can be used for programs which tend to suffer MaGCs of longer duration.

Additionally, each *program family* has two other attributes: (1) An evaluation criterion to determine if the application behaviour falls within that family. For instance, in our previous example, a possible evaluation criterion might be the comparison of the average MaGC duration of the monitored application against the duration ranges of each defined *program family*. (2) A policy which defines the rules to use for MaGC forecasting and load balancing. For example, families might use different approaches to determine the MiGC history required to forecast the MaGCs which are used to decide how best to balance the workload, or different families might require distinct levels of forecast accuracy.

From a process perspective, the proposed autonomic manager has a core process which coordinates the other MAPE-K

[3]http://publib.boulder.ibm.com/infocenter/wsdatap/4mt/topic/com.ibm.dp.xa.doc/administratorsguide.xa35263.htm

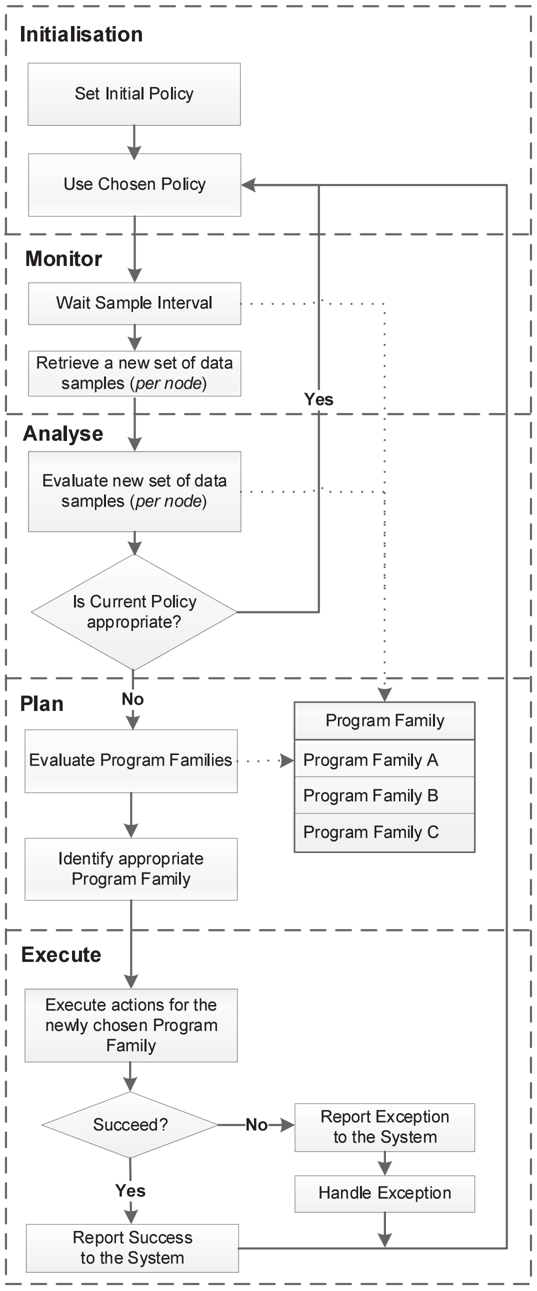
Initialisation
Set Initial Policy
Use Chosen Policy
Monitor
Wait Sample Interval
Retrieve a new set of data samples (per node)
Yes
Analyse
Evaluate new set of data samples (per node)
Is Current Policy appropriate?
Plan
No
Program Family
Evaluate Program Families
Program Family A
Program Family B
Program Family C
Identify appropriate Program Family
Execute
Execute actions for the newly chosen Program Family
Succeed?
No
Report Exception to the System
Handle Exception
Yes
Report Success to the System

Fig. 5. Adaptive GC-Aware Load Balancing Strategy - Process

elements. This process is depicted in Figure 5. It is triggered when the load balancing 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 is used 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 ends: 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 is assessed to identify the new *program family*, which is then used until the next evaluation phase occurs. These actions retrieved their configurations from the database of program families (represented as dashed arrows in Figure 5). Furthermore, any exceptions are internally handled and reported.

IV. Experiment #1: Assessment of Forecast Accuracy

Here, the objective was to identify which program GC characteristics could work better for defining an initial set of program families. To achieve this, we evaluated the effectiveness of our MaGC forecast algorithm (proposed on [15]) against a broad set of Java programs and experimental configurations with the aim of analysing the conditions in which the algorithm performed better. The following sections describe this experiment and its results. Due to space constraint, we only present the most relevant results (as this experiment involved the execution of 69,000 different experimental configurations).

Java Benchmarks. The DaCapo[4] benchmark 9.12 and the SPECJVM [5] benchmark 2008 were chosen as application sets because they offer a wide range of different program behaviours to test (23 between the two benchmarks). Moreover, they are two of the most widely used Java benchmarks in the literature. As the DaCapo benchmark offers different test workloads per program[6], 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 summarized in Table I. 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. 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 [3]).

TABLE I. DaCapo Configurations

Workload Size	Program	#Iters
huge	h2	10
	tomcat	960
	tradebeans	10
	tradesoap	10
large	avrora	4000
	batik	120
	eclipse	10
	jython	10400
	pmd	200
	sunflow	640
	xalan	10400
default	fop	10400
	luindex	10400
	lusearch	10400

GC Strategies. Among the three most commonly used GC strategies in the industry[7], we selected the two which tend to suffer the longest pauses [13]: Serial (sGC) and Parallel (pGC).

MaGC Forecast Algorithm. An extensive range of FWS values was tested: [10..3000] in increments of 10. Moreover, a value of 100ms was selected as *Sampling Interval*, assuming

[4]http://dacapobench.org/
[5]http://www.spec.org/jvm2008/
[6]http://www.dacapobench.org/benchmarks.html
[7]http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html

that no more than one MiGC would occur within that time-frame (hence not missing to sample any MiGC).

Environment. All tests were done in a 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.

Metrics. Three main metrics were calculated: Firstly, the *Forecast Error* (FE) [15], which is the ratio of the absolute forecasting error (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. Secondly, the average number of MiGCs that occurred between MaGCs ($MiGC_{AVG}$). This metric captures the relationship between the allocation needs of an application and the heap size (major factors influencing the GC, as proved by [12] and [19] 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 less), the application is close to an Out-Of-Memory exception. On the contrary, a value far from zero (e.g., 1,000 or more) indicates that the *Old Generation* is infrequently exhausted. Finally, the coefficient of variation[8] ($MiGC_{CV}$) was also calculated. This metric, which is the standard deviation of the $MiGC_{AVG}$ expressed as a percentage of the average, allows the comparison of different applications in terms of variability in memory usage.

Experimental Results. Among the possible alternatives to develop policies to test our load balancing strategy, we have initially concentrated on automating the selection of the FWS for two reasons: (1) The results of our previous work [15] showed that the accuracy of our MaGC forecast algorithm is particularly sensitive to this configuration, as the FWS delimits the range of historical data which is used in the forecast process (as explained in Section II). (2) Even though we achieved a FE below 10% for all the tested combinations of programs and heap sizes in this experiment (with an average FE of 4.56% and a standard deviation of 2.70%), no single FWS achieved the lowest FE for all programs.

For these reasons, our initial analysis focused on identifying the *optimal FWS* (the FWS which achieved the FE closest to zero) per combination of program and heap size. This analysis showed us an interesting trending: In general, our 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, our 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), we analysed the $MiGC_{CV}$ of the programs. This analysis showed us 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 our MaGC forecast algorithm). Whenever those changes are "too drastic" (reflected in a high value of $MiGC_{CV}$), 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. Examples of this behaviour are depicted in Figures 6, 7, and 8: In Figure 6, eclipse-100MB with sGC, which experienced a high $MiGC_{CV}$ (1.816), shows multiple troughs during its execution. Meanwhile, batik-200MB with sGC, which experienced a considerable lower $MiGC_{CV}$ (0.221), shows a more steady FWS growth trend (with minimum troughs) in Figure 7. Finally, avrora-100MB with pGC, which experienced a $MiGC_{CV}$ very close to zero (0.022), practically shows a steady FWS growth trend in Figure 8.

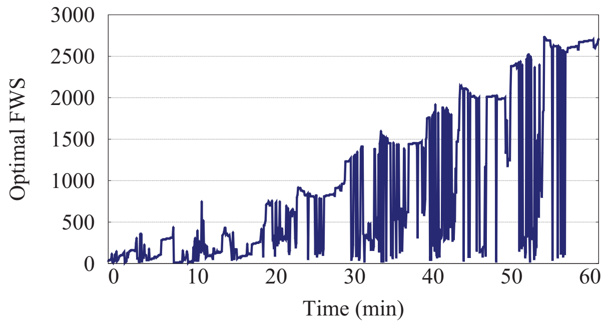

Fig. 6. sGC Eclipse-100MB: FWS vs. Time

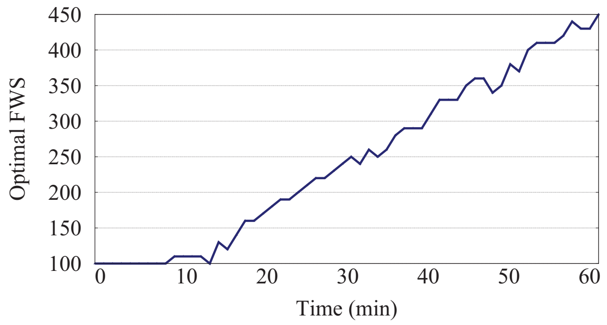

Fig. 7. sGC Batik-200MB: FWS vs. Time

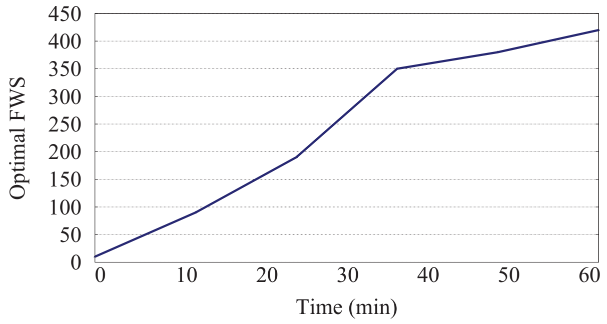

Fig. 8. pGC Avrora-100MB: FWS vs. Time

The above finding led us 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 $MiGC_{CV}$), and be used as a first set of policies within our

[8]http://ncalculators.com/statistics/coefficient-of-variance-calculator.htm

adaptive load balancing strategy. These function-based policies would then allow the automatic selection of an appropriate FWS based on the optimal FWS trend.

We identified three *program families* based on their $MiGC_{CV}$ values: *Low* ($MiGC_{CV}$<=0.1), *medium* (0.1<$MiGC_{CV}$<1.0), and *high* ($MiGC_{CV}$>=1.0). Then, for each program and heap size combination, we derived a trending function 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[9] 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 us to adjust our scope: Instead of concentrating on achieving high forecast accuracy for all the MaGC events, we 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. Our 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 our load balancing strategy to improve a system's performance.

After removing the outliers, we were able to successfully derive FWS trending functions for all the programs using the tested heap sizes. An example of these functions (for eclipse-100MB with sGC) is presented in Figure 9. As it can be noticed, this function has a R2 value very close to 1.0 (meaning that it is very representative of the modelled data).

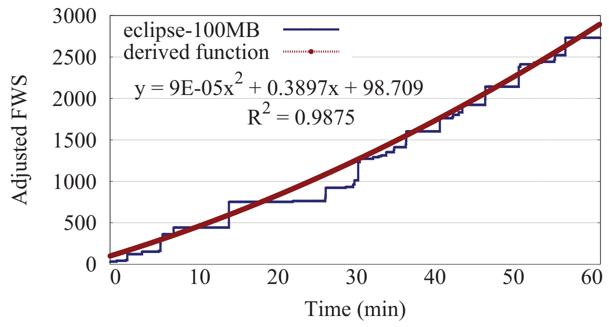

Fig. 9. sGC Eclipse-100: Derived FWS Trending Function

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 $MiGC_{CV}$) 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 our chosen strategy of skipping the outliers, as their number decreases when the variability decreases. In contrast, the $MiGC_{AVG}$ 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 10, which shows the $MiGC_{CV}$ and $MiGC_{AVG}$ trends for the *sunflow* program with pGC when using the different heap sizes.

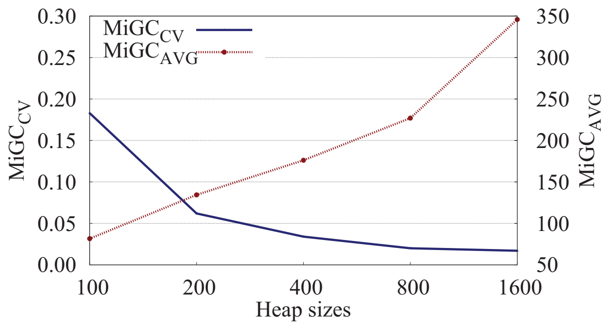

Fig. 10. pGC Sunflow: $\mathrm{MiGC_{CV}}$ vs. $\mathrm{MiGC_{AVG}}$ Trends

V. Experiment #2: Assessment of Performance Improvements

Here the objective was to evaluate if our strategy of program families (currently based on the $MiGC_{CV}$ and using the function-based policies described in our previous experiment) could improve the performance of a distributed system without the need of manual tuning. Due to space constraint, we only present the most relevant results (as this experiment involved the execution of 184 different experimental configurations).

Prototype. To validate our proposed solution, we implemented a prototype using Java 7. Our adaptive load balancing strategy is built on top of the round-robin algorithm (leveraging on the modified version that we proposed on [15]) and the Central Directory[10] load balancer. We selected the round-robin algorithm because it is widely used in the industry. For example, it is the only load balancing algorithm commonly supported across all the top public Cloud vendors [16]. Central Directory was chosen because it is a light-weight load balancer which is open source and developed in Java, characteristics which facilitated its integration with our MaGC forecast logic. Finally, the forecast logic uses Java Management Extension (JMX)[11]. This technology was chosen because it is a standard Java component which can retrieve all the required information (e.g., GC and memory snapshots).

Experimental Setup. This experiment used a distributed environment composed of seven VMs: Five application nodes, one load balancer and one load tester (using Apache JMeter 2.9[12]). All VMs had the characteristics described in Section IV. The application nodes ran an Apache Tomcat 6.0.35.

All the 23 previously tested programs were used in conjunction with the two biggest heap sizes (800 and 1600MB) and the two GC strategies used in Experiment #1 (Section IV). Regarding the MaGC forecast algorithm, its configuration was also similar to the one used in Experiment #1 except the FWS, as our program families took the place of this configuration. Moreover, two types of runs were performed: The first type used the normal round-robin algorithm and was considered the

[9]http://www.businessdictionary.com/definition/coefficient-of-determination-r2.html

[10]http://javalb.sourceforge.net/

[11]http://www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html

[12]http://jmeter.apache.org/

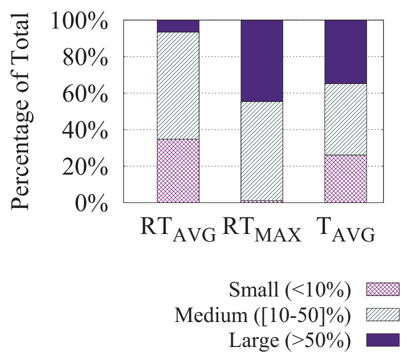

Fig. 11. Breakdown of Improvements

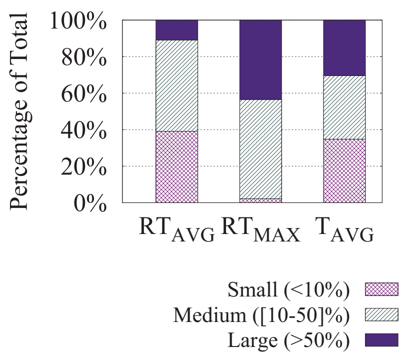

Fig. 12. sGC: Breakdown of Improvements

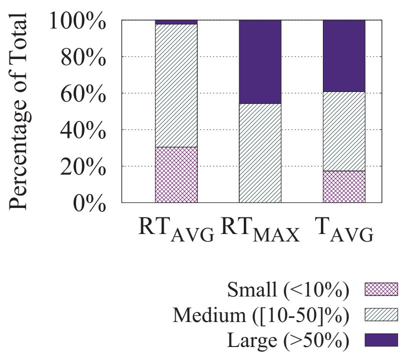

Fig. 13. pGC: Breakdown of Improvements

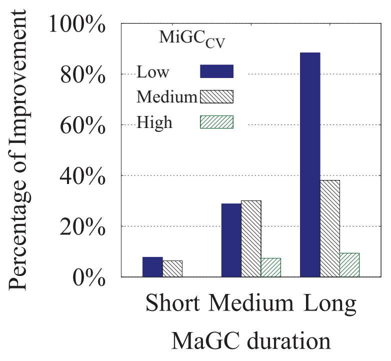

Fig. 14. RT_{AVG}: Performance Improvements

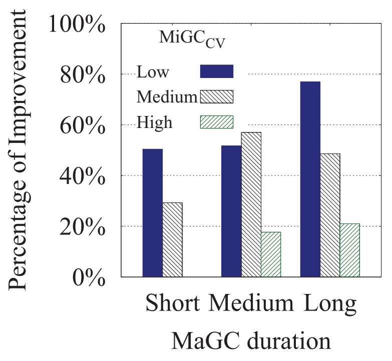

Fig. 15. RT_{MAX}: Performance Improvements

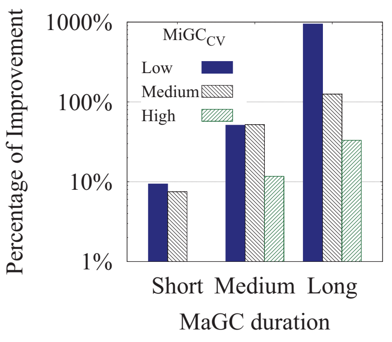

Fig. 16. T_{AVG}: Performance Improvements

baseline. The second type of run used our proposed adaptive GC-aware load balancing strategy.

To be able to call a program from within a JMeter HTTP test script (so that multiple concurrent calls can be invoked), 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 among the calls; while for the SPECJVM programs, it involved varying the execution time. Each JMeter test run lasted 60 minutes and used 50 concurrent users. Also, each individual program call was considered a transaction. Finally, as we were interested in performance, the throughput (tps) and the response time (ms) were our main metrics and were collected with JMeter. Additionally, the CPU (%) and memory (MB) utilisations of the load balancer were collected with the "top" command.

Experimental Results. In general terms, our proposed solution worked well because all experimental configurations achieved a performance improvement. To have a better understanding of the results, we categorised them in three groups, based on the level of performance improvement achieved: *Small* (<=10%), *medium* (>10% and <=50%) and *high* (>50%). This analysis showed that most of the experimental configurations obtained a performance improvement higher than 10%. More importantly, our solution achieved a *high* performance improvement in a fair percentage of the cases: 4% in terms of RT_{AVG}, 45% in terms of RT_{MAX}, and 34% in terms of T_{AVG}. This breakdown is shown in Figure 11. The same exercise was done per GC strategy. In general, similar results were obtained, meaning that our solution worked irrespective of the used GC strategy. These breakdowns are presented in Figures 12 and 13.

Next, our analysis focused on understanding the conditions under which our solution works the best. Figures 14, 15 and 16 show the results of this analysis for each performance metric. In general, the biggest performance gains were obtained in those programs which suffered longer MaGCs (hence having the biggest potential performance gains) and low $MiGC_{CV}$ (condition under which the solution achieved a higher forecast accuracy and was able to successfully skip all the MaGCs). On the contrary, the smallest performance gains were obtained in the following two cases: (1) When the MaGC duration was short (as there was little potential performance gain to win) or (2) when the $MiGC_{CV}$ was high (condition under which the solution missed to predict accurately some MaGCs and it was not able to skip all the MaGCs). In this analysis, a program was considered to have short MaGC duration ($MaGC_D$) if it spent less than 1% of its execution time doing GC work, long $MaGC_D$ if it spent more than 10%, and medium $MaGC_D$ if it did not fall into the previous categories.

An additional analysis was done to evaluate the influence of the $MiGC_{CV}$ in the achieved performance improvements. Figure 17 shows the results of this analysis. There, it can be observed how the percentage of skipped MaGCs gradually increases (while the $MiGC_{CV}$ decreases) until reaching practically 100% of skipped MaGCs around a $MiGC_{CV}$ of 0.1. These results confirm that $MiGC_{CV}$ is an appropriate metric to be used for classifying the different program behaviours into families. In this context, a MaGC was considered skipped if it was forecasted accurately enough that it was possible to prevent sending transactions to the affected node during the duration of the MaGC. Under these conditions, the only transactions affected by the MaGC event were those in the pipeline to be processed by the node which suffered the MaGC, transactions which eventually led to the triggering of the MaGC event.

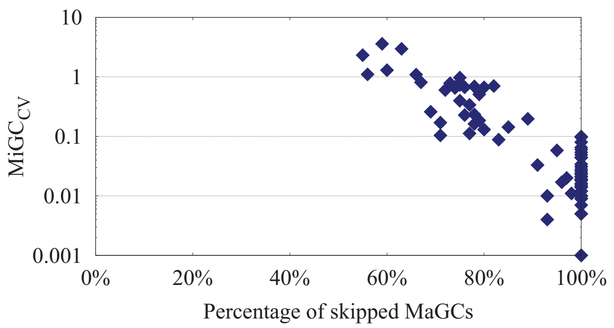

Fig. 17. Comparison of Skipped MaGC vs. Experienced $\mathrm{MiGC_{CV}}$

As a minor contribution of this work, the results of this experiment also allowed us 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 II.

TABLE II. Program Classification

$MiGC_{CV}$	$MaGC_D$		
	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

Finally, we also compared the resource usages in the load balancer node to understand the costs of our solution (compared against the normal round-robin): The average CPU usage (CPU_{AVG}) increased in the range of 3% and 7%, while the average memory usage (MEM_{AVG}) increased in the range of 0.11 and 0.35GB. This memory increase was caused by the historical data that the forecast logic maintained. These increments were considered tolerable because the load balancer node was far from exhausting its resources (reporting a CPU_{AVG} of 27.3% and a MEM_{AVG} of 1.42GB).

In summary, this experiment proved the performance gains that our solution can achieve: By avoiding the impact of most of the MaGC events in the individual nodes, the performance of the clustered applications was improved (in terms of response time and throughput), without human intervention.

VI. Conclusions and Future Work

This paper proposes a new adaptive GC-aware load balancing strategy to improve the performance of high-assurance Java distributed systems. This solution extends our previously proposed GC-based load balancing strategy (which uses a static configuration), by adapting the load balancing strategy to the GC characteristics of the underlying application (hence avoiding the need of manual configuration). This behaviour allows our solution to use the MaGC forecasts to prevent that the MaGC events on the individual nodes affect the performance of the overall system. The results showed that such adaptive strategy can improve the response time and throughput of a distributed system: The average response time decreased 22.5% (across all programs), while the average throughput increased 60.7% (across all programs), both metrics compared to the round-robin load balancing. Our future work will focus on exploring other program GC characteristics to broaden our classification of program behaviours into families, then use that knowledge to develop more portable load balancing policies which exploit the behaviour similarities of each family.

Acknowledgments

Supported, in part, by Science Foundation Ireland grant 10/CE/I1855.

References

[1] A. Amin, A. Colman, and L. Grunske. Using automated control charts for the runtime evaluation of qos attributes. *HASE*, 2011.

[2] K. Barabash, O. R. I. Ben-yitzhak, I. Goft, E. K. Kolod, V. Leikehman, Y. Ossia, and A. V. I. Owshanko. A Parallel, Incremental, Mostly Concurrent Garbage Collection for Servers. *TPLS*, 2005.

[3] S. M. Blackburn, P. Cheng, and K. S. Mckinley. Myths and Realities: The Performance Impact of GC. *SIGMETRICS PER*, Jan. 2004.

[4] A. B. Carmona, J. Roca-piera, C. H. Capel, and J. A. Álvarez bermejo. Adaptive Load Balancing between Static and Dynamic Layers in J2EE Applications. In *NWeSP*, 2011.

[5] J. González, A. Muñoz, and A. Maña. Multi-layer monitoring for cloud computing. *HASE*, 2011.

[6] D. J. Kephart. The vision of autonomic computing. *Computer*, Jan. 2003.

[7] J. C. Jeremy Singer, Gavin Brown, Ian Watson. Intelligent Selection of Application-Specific Garbage Collectors. In *ISMM*, 2007.

[8] T. Kalibera. Replicating real-time GC for Java. *JTRES*, 2009.

[9] P. Lengauer and H. Mössenböck. The taming of the shrew: increasing performance by automatic parameter tuning for java garbage collectors. *ICPE*, 2014.

[10] Y. Liu, L. Wang, and S. Li. Research on self-adaptive load balancing in EJB clustering system. *ISKE*, 2008.

[11] W. Manning. Scjp sun certified programmer for java 6 exam. *Emereo Pty Ltd, London*, 2009.

[12] F. Mao, E. Z. Zhang, and X. Shen. Influence of program inputs on the selection of garbage collectors. *SIGPLAN VEE*, 2009.

[13] S. Microsystems. Memory Management in the Java HotSpot Virtual Machine. *April*, 2006.

[14] F. Pizlo, E. Petrank, and B. Steensgaard. A study of concurrent real-time Garbage Collection. *PLDI*, 2008.

[15] A. Portillo-Dominguez, M. Wang, D. Magoni, P. Perry, and J. Murphy. Load balancing of java applications by forecasting garbage collections. *ISPDC*, 2014.

[16] M. Rahman, S. Iqbal, and J. Gao. Load balancer as a service in cloud computing. *SOSE*, 2014.

[17] L. Rupprecht, A. Reiser, and A. Kemper. Dynamic load balancing in data grids by global load estimation. *ISPDC*, 2012.

[18] F. Siebert. Limits of parallel GC. *ISMM*, 2008.

[19] J. Singer, R. E. Jones, G. Brown, and M. Luján. The economics of garbage collection. In *ISMM*, 2010.

[20] R. G. Snatzke. Performance survey. *Codecentric AG*, 2008.

[21] M. T. Vechev, E. Yahav, and D. F. Bacon. Correctness-preserving derivation of concurrent garbage collection algorithms. *PLDI*, 2006.

[22] M. Wegiel and C. Krintz. Dynamic prediction of collection yield for managed runtimes. *SIGPLAN Notices*, Feb. 2009.

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

[24] P. R. Wilson. Uniprocessor GC Techniques. In *IWMM*, 1992.

[25] F. Xian, W. Srisa-an, and H. Jiang. Fortune Teller: Improving Garbage Collection Performance in Server Environment using Live Objects Prediction. *OOPSLA*, 2005.

[26] F. Xian, W. Srisa-an, H. Jiang, and A. Hall. GC: Java Application Servers' Achilles Heel. *Science of Computer Programming*, Feb. 2008.