Research Repository UCD

Title	Improving the Testing of Java Garbage Collection Through an Efficient Benchmark Generation
Authors(s)	Portillo Dominguez, Andres Omar, Ayala-Rivera, Vanessa
Publication date	2018-10-26
Publication information	Portillo Dominguez, Andres Omar, and Vanessa Ayala-Rivera. “Improving the Testing of Java Garbage Collection Through an Efficient Benchmark Generation.” IEEE, October 26, 2018. https://doi.org/10.1109/conisoft.2018.8645889.
Conference details	CONISOFT 2018: 6th International Conference in Software Engineering Research and Innovation, San Luis Potosí, Mexico, 24-26 October 2018
Publisher	IEEE
Item record/more information	http://hdl.handle.net/10197/10606
Publisher's statement	© 2018 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.
Publisher's version (DOI)	10.1109/conisoft.2018.8645889

Downloaded 2026-05-01 23:35:30

The UCD community has made this article openly available. Please share how this access benefits you. Your story matters! (@ucd_oa)

© Some rights reserved. For more information

Improving the Testing of Java Garbage Collection Through an Efficient Benchmark Generation

A. Omar Portillo-Dominguez, Vanessa Ayala-Rivera
Lero@UCD, School of Computer Science,
University College Dublin, Belfield, Dublin 4, Ireland.
Email: {andres.portillodominguez,vanessa.ayalarivera}@ucd.ie

***Abstract*—Garbage Collection (GC) is a core feature of multiple modern technologies (e.g., Java, Android). On one hand, it offers significant software engineering benefits over explicitly memory management, like preventing most types of memory leaks. On the other hand, GC is a known cause of performance degradation. However, it is considerably challenging to understand its exact impact on the overall application performance. This is because the non-deterministic nature of GC makes very complex to properly model it and evaluate its performance impacts. To help tackling these problems, we present an engine to generate realistic GC benchmarks by enabling to effectively capture the GC/memory behaviours experienced by real-world Java applications. We also demonstrate, through a comprehensive experimental evaluation, how such benchmarks can be useful to strengthen the evaluation of GC-related advancements.**

***Keywords*—Software Testing, Object-Oriented Systems, Java, Garbage Collection, Benchmark Generation, Performance**

I. Introduction

Garbage Collection (GC) is an essential process of many of the most widely-used run-time platforms. For instance, it is a key feature of Java which is a predominant technology at enterprise level [1]. Similarly, mobile platforms like Android, which is one of the most dominant players in the market [2], rely on GC to manage their memory. The popularity of GC comes from the fact that it automates most of the memory-related tasks. This translates into significant software engineering benefits: Since GC releases developers from the burden of manual memory management, it helps to avoid the most common sources of memory leaks [3]. Consequently, GC has a positive impact on the productivity of development teams [4].

Despite its benefits, GC comes with a cost: Whenever triggered, GC impacts the system performance by pausing the involved programs [5]. Even though small pauses (e.g., milliseconds) do not commonly represent a problem, higher delays (e.g., seconds or minutes) can cause trouble for applications that require fast response time and/or high throughput. Research studies have provided evidence of the GC costs. For instance, the work on [6] identified GC as a key factor degrading the throughput of Java application servers (a traditional Java business niche). In their experiments, GC overhead reached 50% of the execution time, involving pauses as high as 5 minutes. Research studies have also shown that no single GC configuration can work well for all scenarios. This is because the GC behaviour is not only influenced by the application workloads, but also by factors such as the chosen GC strategy, and the system configuration [7].

For all these reasons, it is usually agreed that GC plays an important role in the performance of automatic memory management technologies. However, due to the complex interactions between the GC and the many factors that affect its behaviour, the exact influence that GC has on a particular application's performance is hard to define. This manifests in the lack of explicit guidelines for practitioners to properly configure the GC. Instead, vendors typically give recommendations for trial-and-error tuning [8]. Similarly, in the performance engineering community, the GC overhead is often modelled as a constant factor that needs to be calibrated [9]. Thus, the accuracy of such models is at best difficult to assess. This problem is also reflected in the scarcity of GC benchmarks, which indirectly makes the evaluation of GC-related advancements much harder.

To help address these challenges, our research has concentrated on developing techniques to automatically generate realistic GC benchmarks which can be applicable to different use cases. The aim is to help researchers and practitioners (hereinafter referred as users) to increase their productivity by facilitating the creation of suitable GC benchmarks which can be easily re-used during their research and development cycles. Moreover, our techniques foster the reproducibility of experiments as they contribute to obtain reliable results. Specifically, the contributions of this paper are:

1) An automated engine to generate realistic GC benchmarks which can capture the complex and time-varying GC behaviour of Java applications.
2) A comprehensive practical evaluation of the engine. It consisted of a prototype and a series of experiments which prove the benefits it brings to the evaluation of GC-related advancements, as well as its costs.
3) 360 publicly available GC benchmarks [10]. They are based on real-life Java applications that span a wide range of diverse GC/memory behaviours.

II. Background & Related Work

The memory area in Java is known as *heap*. Nowadays, one of the most common types is the *generational heap* [1], where objects are segregated by age into memory regions known as generations. New objects are created in the youngest generation, and the GC in this generation is known as Minor GC (MiGC). It is usually inexpensive and rarely causes a performance concern. MiGC is also in charge of moving the

The final publication is available at IEEE Xplore via https://ieeexplore.ieee.org/document/8645889

live objects, which have become old enough, to the older generations. The GC in the older generations is known as Major GC (MaGC) and it is commonly accepted as the most expensive GC type due to its performance impact [1]. Running out of free memory in a generation triggers its respective type of GC event. Additionally, the heap is managed by the GC strategy configured at the start-up of the Java Virtual Machine (JVM). The availability of GC strategies is usually tied to the heap type. For instance, three of the most widely-used strategies [11] work exclusively on generational heaps: The Serial GC (for client JVMs), the Parallel GC (for server JVMs when throughput is more important than response time), and the Concurrent GC (for server JVMs when response time is more important than throughput).

Benchmarking has proven useful in many domains to enable the fair comparison of rival approaches [12]. For example, the work on [13] proposed a benchmark suite to measure the performance of GPU architectures, while the authors of [14] presented a benchmark for evaluating reconfigurable architectures. In the Java world, there have also been efforts to create realistic benchmarks. For instance, the DaCapo and SPECJVM suites (sponsored by companies such as IBM and Microsoft) are composed of 24 different real-world applications and specialised programs [15], [16]. Meanwhile, the authors of [17] proposed an approach to build clustered test environments by leveraging Java benchmarks. Finally, [18] presented an approach to generate GC benchmarks. However, it does not allow to configure/modify the GC properties (and application behaviours) of interest, a characteristic which considerably limits its practical usage and extensibility.

Furthermore, multiple research efforts have aimed to improve GC performance. For instance, several works have developed algorithms for particular uses. For example, the authors of [20] proposed an algorithm which exploits the differences between the remote and local objects residing on a Java application server. Similarly, the work on [2] presents a memory manager service for Android to optimise the GC execution across all applications, while the work on [21] proposed a GC-aware load balancing strategy to reduce the GC impact in the overall system performance. In contrast to these works, that have aimed to improve the behaviour of the GC process, our goal has been to develop an easily extendable engine to generate GC benchmarks that properly captures the complexities in GC and memory behaviours that occur in real-life Java applications.

III. Proposed Solution

The goal of this research work has been to develop an engine (HERMES) to generate realistic GC benchmarks that can be useful to better test GC-related advancements. Fig. 1 depicts the conceptual view of the engine. HERMES is composed of two core elements: The *Executor*, which is responsible for performing a controlled execution of a Java program, and the *Collector*, which closely monitors the Java program execution (i.e., its running JVM) to retrieve all needed information (e.g., GC snapshots). Monitoring and collection are periodically carried out to gradually generate a new GC benchmark (in comma-separated value format). This process can be tailored via the *input parameters* provided by the user. For instance, a user might be interested in understanding the impacts of explicitly requesting the JVM to trigger GCs. So, s/he might generate the GC benchmarks with this type of programmatic requests enabled. Alternatively, other users would prefer to explore the GC behaviours without this discouraged programming practice to obtain clearer GC behaviours [22].

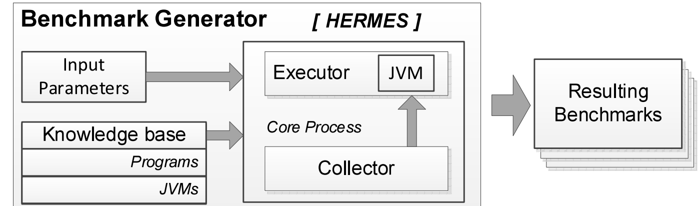


Fig. 1. HERMES' Conceptual View.

With the aim of making the engine easily extensible, we have included a *knowledge base* (KB), where the set of supported programs and JVMs can be modelled. Specifically, the KB encapsulates a set of common characteristics that are shared by Java programs and JVMs (later discussed in Section III-B). This element makes HERMES capable of quickly incorporating multiple use cases (e.g., different JVMs or GC strategies), which might be suitable to fulfil the requirements of different test scenarios. For instance, the Java language specification [23] dictates that most of the monitoring information that a JVM provides is implementation-independent (i.e., one will retrieve the same type of monitoring information regardless of the tested JVM). Therefore, one can abstract the information of a JVM to create configurations that might be reused (with minimal modifications) for the generation of benchmarks based on other JVMs.

A. Core Process

HERMES follows a process composed of three phases. They are depicted in Fig. 2 and explained next:

1) Initialisation: The process starts by setting all the configured input parameters. These consist of six user inputs that define the desired characteristics of the GC benchmark and the program's execution: (1) the JVM over which the program will be run (among the ones supported by HERMES); (2) the GC strategy that will be used; (3) the heap size that will be used; (4) the program that will be run (among the ones modelled in HERMES); (5) how long the program needs to be executed (e.g., 1 hour); and (6) a sampling interval (expressed in milliseconds) to indicate how often GC/memory samples will be collected from the monitored JVM. Additionally, the user can input the set of specific parameters that a particular program might require to be properly executed, or set values that would override those already stored in HERMES' KB. Similarly, a user can set additional JVM arguments to further customise the execution of the program (other than the heap size and the GC strategy). In this phase, a new

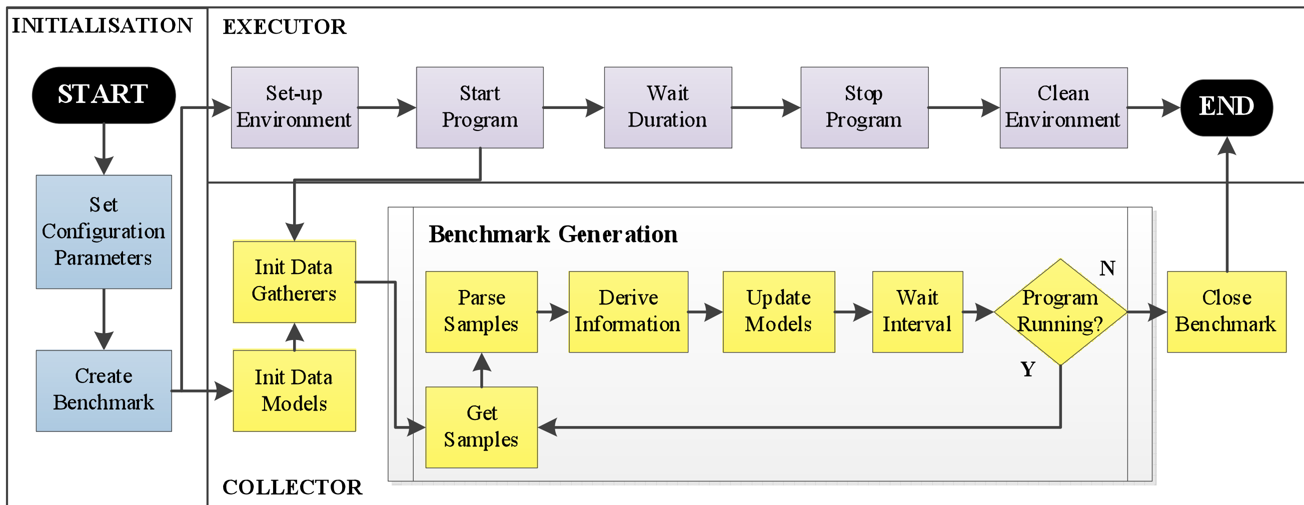


Fig. 2. HERMES' Core Process.

empty GC benchmark is generated (covering all the modelled memory/GC attributes, discussed in Section III-B). Next, the other two phases of the process are concurrently started: The *Executor*, which is responsible for actually running the program; and the *Collector*, which is responsible for gradually producing the GC benchmark by iteratively retrieving samples from the required data sources in the monitored JVM.

2) Executor: This phase starts by setting up the experimental environment that the chosen program needs to be executed successfully (e.g., any required dependencies). For instance, the program might require the existence of a temporal directory, or the presence of some workload input data. Next, the execution of the configured program starts. This step involves dynamically constructing the command required to run the program (as each modelled program might have a different syntax and set of options available), enabling the applicable JVM sampling strategy, and handling the program's execution. This level of abstraction ensures that the engine can be easily extensible for other Java programs, as they usually share the same type of starting steps (e.g., they have as entry point the main method of a class). Once the program has started, the Executor reports it to the Collector (including the configuration needed to connect to the program's JVM), so that the Collector can continue its initialisation. This step also involves saving the process ID of the program (so that the Executor can access it later on), as well as parsing/processing the standard input and output streams of the program. Then, the process waits the time configured as planned duration. Once this has elapsed, the Executor stops the program (indirectly stopping the Collector too). Finally, the *clean environment* step is carried out and all the dependencies previously set are rolled back (e.g., anything that has been copied into a temporal directory is erased). The aim is to make the engine self-sustainable by releasing resources (e.g., hard disk) that had been temporarily allocated to successfully execute the program (as many programs can be configured to run in sequence). Moreover, any exceptions and errors are internally handled.

3) Collector: This phase starts by initialising the data models that will capture the memory/GC behaviours of the chosen JVM type. It involves two types of models: (i) One suitable for a memory region (e.g., a generation), which is composed of metrics such as maximum, committed, used, and free memory; (ii) Another suitable for a GC type (e.g., MiGC), which captures information such as its duration, and the memory usage before and after its occurrence. Next, the data gathering that is required to collect the information is initialised (its technical details are explained in Section III-C). This step has the additional constraint of waiting for the monitored JVM to start (as explained in the Executor section). Later, the following steps are iteratively performed until the program execution finishes: First, an updated snapshot of the current memory/GC usage in the monitored JVM is obtained. It involves getting samples of diverse sources such as all the involved memory regions (i.e., generations) and the GC types (e.g., MiGC and MaGC for generational heaps). The memory samples are composed of information such as a timestamp and the amounts of memory usage of the corresponding memory generation from the monitored JVM. The GC samples are composed of information such as the start time, the end time, and the complete memory usage (i.e., initial, committed, used, and free memories) before and after the latest event of the corresponding GC type. Once there are new samples from all the data gatherers, they are parsed to save their attributes, as well as processed to calculate a set of delta GC-related metrics that complement the observed GC behaviour by properly quantifying the changes in each of the monitored elements (i.e., GC and memory). In the case of the MaGC, it involves the calculation of the number of MiGCs that triggered this event. This is because the MiGC statistics are useful to characterise the GC behaviours experienced by applications (e.g., they can be used to forecast MaGC events [24]). For this same reason, the set of metrics derived from the MiGC information is more extensive, including metrics such as *YoungGen memory allocation* (how much memory was used

to create new objects between the latest two MiGCs); *OldGen memory allocation* (how much OldGen allocation occurred due to the latest MiGC, meaning that some objects have become old enough to be moved to the OldGen by the latest MiGC); *OldGen allocation between MiGCs* (how much OldGen allocation occurred between the latest two MiGCs); and *time between MiGCs* (how much time elapsed between the latest two MiGCs). Once all metrics have been calculated, the engine's models are updated (i.e., memory and GC ones) and saved to disk. This step is performed in its own thread, so that the Collector does not need to wait for the saving to complete (which might have put at risk sampling at the desired frequency). Next, the logic awaits the configured sampling interval before starting the next round of the iterative steps of the phase. This loop continues until the program finishes its controlled execution and the new GC benchmark has been generated. Similarly to the Executor, the Collector internally handles any errors or exceptions that might occur during its execution (e.g., a data gatherer failing to report).

B. Architecture

HERMES is implemented with the architecture presented in Fig. 3, which shows a simplified view of the engine. Internally, HERMES follows a Model-View-Controller (MVC) architecture in which every GC component is modelled as an object and interacts with each other (through interfaces) within the engine. This design decision was taken to make the architecture highly extensible and lowly maintainable, as one of our middle-term goals is to extend HERMES to support generating other types of benchmarks. For instance, based on other computational resources like CPU utilisation, network usage, or even hybrid scenarios (which might be useful to realistically represent multi-factorial performance issues).

In Fig. 3, it can be seen how the *MainClient* handles all tasks associated with the initialisation of the core process (e.g., setting the configuration parameters), and the saving of the results (i.e., the GC benchmark). The client also initialises the *Executor* and *Collector* components, which implement the logic of their corresponding phases to control the execution of the programs and the GC benchmark generation. Besides, the *Collector* class is responsible for the instantiation of the different model objects, which are later used to capture the GC behaviour. In this part, it was important to identify shared characteristics among the investigated elements (e.g., GC strategies and memory regions) to model them into a reusable hierarchy of classes. This is important not only to reduce the code repetition, but also to efficiently support additional GC strategies. Following the same line of thought, the characteristics of the Java programs (e.g., DaCapo) and JVMs (e.g., Oracle HotSpot) were abstracted into a knowledge base. This is important because a key design goal of this architecture is to make it general enough to support other programs and JVMs. The two types of elements currently supported by HERMES in its knowledge base are:

1) *Java programs.* A program is defined by a name that uniquely identifies it within the engine (so that the engine can support different versions or configurations of the same program); the path where the program's executable is located; the name of the executable (typically a Java ARchive); the program's dependencies (if any); the name of the program's main class (used by the engine to start the program through reflexion invocation); any default input parameters for the program (e.g., input workloads); and the type of execution that the program natively supports. Two types of execution are currently supported: Time-bound (where the engine propagates the planned duration to the program) and the iteration-bound (where the engine translates the planned duration to the maximum number of iterations supported by the program). This is performed to allow the program to run the planned duration (as its actual execution will be stopped by the Executor).
2) *JVMs.* A JVM is defined by a name that uniquely identifies it (e.g., HotSpot7); the path where the JVM is installed; any JVM startup arguments (e.g., some data gathering strategies, such as the generation of Javacores used by the IBM JVM, require extra configurations to work because this functionality is disabled by default); the gathering type (e.g., JMX, which is the standard monitoring approach in Java); and the GC strategies and memory regions that the JVM supports.

It is important to highlight that HERMES internally leverages multi-threading to perform many of its tasks concurrently (within the limits of the machine where it runs). This design decision was taken to reduce the time complexity of the engine. To illustrate the need for these types of optimisations, consider, for example, the Executor. As the execution of the program occurs in an independent process, a thread needs to be triggered to track the execution of the program and stop it once the planned duration has been reached. Similarly, the Collector uses a multi-threading approach to collect the GC and memory samples. Even though the collection of these samples is typically very fast (e.g., few milliseconds), the use of threads is helpful to reduce the total time of the activity and make sure that the Collector can satisfy the configured sampling interval. A similar scenario occurs with the disk writing. Even though the information to save is minimal (hence, it is very unlikely that the hard disk will become a bottleneck), this step is done in its own thread to avoid interfering with the rest of the phase and be certain that the monitored GC behaviours are correctly captured (i.e., they reflect their real-time counterparts).

C. Prototype

It was developed external to the JVM with the aim of maximising the portability and overall adoption of our solution. Also, this design facilitated the experimental evaluation, as the prototype could work with any JVM currently available in the industry. From a technical perspective, the prototype was developed in Java 7, using Java Management Extension (JMX) to interact with the monitored JVM. This technology

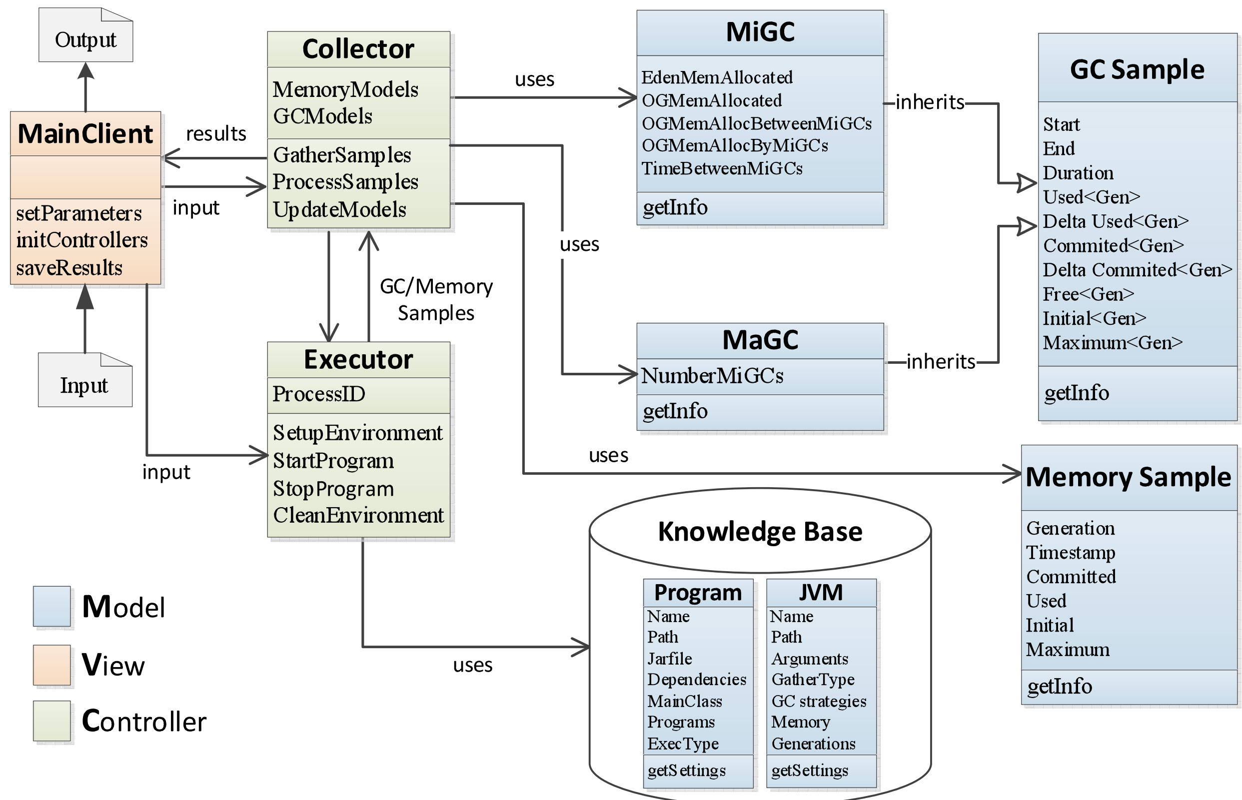


Fig. 3. HERMES' Architecture.

was chosen as it is a standard component of Java which allows the information retrieval from all the needed JVM resources.

Regarding the knowledge base, we initially focused on supporting the Oracle HotSpot JVM (one of the leading commercial JVMs [25]) and the generational heap (the most common heap type [1]). In terms of programs, we selected DaCapo and SPECJVM [15], [16] because they are two of the Java benchmarks most widely-used in the literature, offering an ample range of 24 different programs.

IV. Experimental Evaluation

In this section, we describe the experimental setup used in our evaluation, as well as discuss the results obtained.

A. Experimental Setup

Below, we present our methodology, test environment, experimental configurations, and evaluation criteria used.

1) Methodology: Our experiments pursued three objectives: (1) to demonstrate the capability of HERMES to generate GC benchmarks of diverse characteristics; (2) to investigate the effectiveness of the generated benchmarks for improving GC-related evaluations; (3) to analyse the costs of using HERMES in terms of the computational resources consumed. To achieve this, we conducted two series of experiments. First, a set of GC benchmarks was created using our prototype. Then, the benchmarks were used in the evaluation of a GC-related advancement. All the experiments were performed in an isolated test environment composed of a computer with 4 CPUs (at 2.20GHz) and 3GB of RAM, running Linux Ubuntu 12.04L and Oracle HotSpot JVM 7.

2) Generated GC Benchmarks: To comprehensively evaluate HERMES, we generated 360 GC benchmarks. This was achieved by running all the possible combinations derived from the execution of the 24 DaCapo/SPECJVM programs with the 3 most widely used GC strategies [1] (i.e., Serial -sGC- , Parallel -pGC- , and Concurrent -cGC-), and 5 different heap sizes (100, 200, 400, 800, and 1600MB). It is important to highlight that different heap sizes and GC strategies were chosen because they are major factors affecting the GC behaviour [7]. The duration of the program executions was set to 60 minutes; the JVM was configured to initialise the heap to its maximum size, with the calls to programmatically request a MaGC disabled (as per common tuning practices [22]). Also, a 100ms sampling interval was used (as it has been proven appropriate for both Dacapo and SPECJVM [21]).

3) GC-related Advancement: To assess if the GC benchmarks generated by HERMES could improve the testing of GC-related advancements, we applied them in the evaluation of a forecast algorithm (denominated MaGa) which predicts MaGC events [26]. We conducted two types of test runs: The first type (referred as *real-time*) used the original algorithm's implementation which interfaces with a real JVM. The second type (referred as *offline*) used a modified version of the algorithm's implementation which retrieves its data from a text file (i.e., our GC benchmarks). The MaGa algorithm is particularly sensitive to the forecast windows size (FWS),

a configuration parameter which delimits the historical data used by the algorithm to forecast MaGC. Hence, we tested the algorithm's accuracy with various FWS values. For its original implementation (i.e., real-time), we tested 3 representative values of FWS $\in$ [10, 1500, 3000]. For the modified implementation (i.e., offline), we tested a larger range, varying the FWS $\in$ [10..3000] in increments of 10. Finally, the same set of Java programs, heap sizes, GC strategies, and sampling interval used in the GC benchmark generation were used here.

4) Evaluation Criteria: To evaluate the diversity among the generated GC benchmarks, we used the Principal Components Analysis (PCA), which is a statistical technique commonly used to assess the dissimilarity among benchmarks [27]. PCA works by reducing an X-dimensional space into a lower dimensional uncorrelated space composed of Y principal components. The chosen constituent metrics were the average and standard deviations of the frequencies of all the attributes in the GC benchmarks (a strategy used in other works [27]). To measure MaGa's accuracy, we utilised the forecast error (FE) [21], a metric which captures the difference between the forecasted and the real time of a MaGC event. In terms of computational costs, our metrics were the CPU (%) and memory (MB) utilisations, and the execution time. GC overhead was also monitored due to its relevance in Java's performance.

B. Experimental Results

Here, we discuss our results in terms of the applicable metrics. Due to space constraints, we only present the most relevant results (as our experiments involved the execution of more than 12000 different experimental configurations).

1) Benchmark Diversity: We ran the PCA on the GC benchmarks generated by HERMES (grouped by GC strategy and heap size) to assess their diversity. Typically, most of the variance is explained by the first few principal components. In our case, it was PC1 and PC2 (accounting at least 80% in all cases), so our analysis centred on them. Our hypothesis was that our selection of heap sizes, GC strategies, and programs should lead to a varied range of GC/memory behaviours. This was confirmed by the results of this analysis, as a fair diversity was achieved for all GC strategies and heap sizes. This behaviour is visually illustrated in Fig. 4, which shows how the GC benchmarks differ in a two-dimensional space (PC1 vs. PC2) for the 3 GC strategies using 100, 400, and 1600MB heap sizes. Intuitively, the farther the distance that separates the benchmarks (i.e., the triangles), the more different they are with respect to their constituent metrics. Thus, as the benchmarks are well dispersed in all figures (and across them), this proves that the benchmarks differ independently of the GC strategy (or heap size). This can be clearly observed by comparing the figures vertically (i.e., by varying the GC strategy) as well as horizontally (i.e., by varying the heap size).

In conclusion, this analysis confirmed that we achieved the goal of generating a diverse set of GC benchmarks. Thus, various scenarios of GC behaviours could be captured. This also indicates that our selection of experimental configurations (e.g., programs and heap sizes) was appropriate.

2) Forecast Accuracy: Next, we assessed if the generated benchmarks properly captured the memory/GC behaviours of the programs. For this purpose, we tested the GC benchmarks applied to the MaGa algorithm. It is important to remark that our intention was not to exhaustively investigate the forecast accuracy of the algorithm, but to document how the resulting benchmarks were useful to broaden, and simplify, the testing of GC-related techniques. Firstly, all test runs were executed (both real-time and offline). Next, the FE of the MaGC predictions were calculated and the differences between the FEs of the two types of runs compared. This involved comparing the test runs using the FWS values of 100, 1500, and 3000. Although the results varied among the different GC strategies, this analysis showed that the offline test runs were able to achieve similar FEs when compared to those of their real-time counterparts. This is because the differences between the two types of runs were minimal (e.g., the average difference in FE was 1%). Thus, these results demonstrated that HERMES worked well, as the offline test results adequately mimicked the real-time results. Besides, this confirmed that the generated benchmarks properly captured the GC/memory behaviours of the programs. These results are depicted in Fig. 5, which shows the average difference in FE per GC strategy.

To further exemplify how HERMES can be useful to extend the validation of GC advancements, we present, in Figs. 6.a and 6.b, a comparison of the FWS trends that were obtained with the results of the real-time MaGa (i.e., the limited range of 3 FWS) against the results of the offline MaGa (i.e., the extended range of 31 FWS) for the different heap sizes tested on Eclipse-sGC (one of the programs within DaCapo). As the forecast accuracy of MaGa is highly sensitive to the chosen FWS, it is important to know which FWS achieves the highest accuracy (i.e., the FE closest to zero) for a particular case. If we try to use the information of Fig. 6.a to analyse this, it can be seen that there are not FWS values that achieve such high accuracy for all the heap sizes (e.g., the 200MB scenario). Also, it is hard to see any clear trend with respect to the FWS values with such limited results (e.g., to determine if an increment in FWS improves or deteriorates the accuracy). In contrast, Fig. 6.b allows a more detailed analysis (as the FWS trends per heap size can be clearly identified). For instance, if we review the behaviour of the 100MB heap, the accuracy is considerably poor (i.e., FE $> \pm$ 100%) when FWS $\leq$ 1000 or FWS $\geq$ 2000, while the highest accuracy was achieved when 1400 $\leq$ FWS $\leq$ 1600. A similar analysis can be done for the other heaps, where it is always possible to find a range of FWS where the FE is significantly close to 0%.

Besides, the extended results can enable other analyses. For instance, to investigate the relationship between the FWS and the GC/memory behaviours experienced by Eclipse to understand the changes in the FWS which achieved the highest accuracy (per heap size). This relationship is explained by the changes in behaviour exhibited by Eclipse in terms of the average number of MiGCs that occur between MaGCs ($MiGC_{AVG}$). This metric captures the relationship that exists between the allocation needs of an application

(a)cGC-100MB (b)cGC-400MB (c)cGC-1600MB

(d)pGC-100MB (e)pGC-400MB (f)pGC-1600MB

(g)sGC-100MB (h)sGC-400MB (i)sGC-1600MB

Fig. 4. PC1 vs PC2 of Generated GC Benchmarks per GC Strategy - Heap Size.

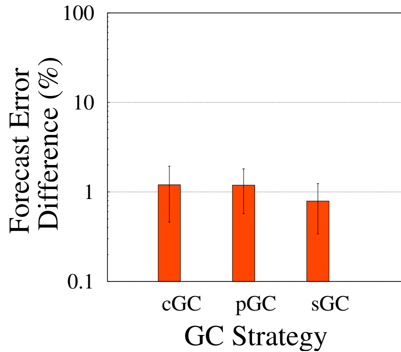
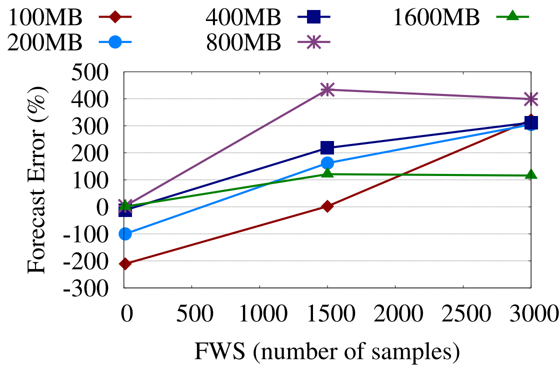


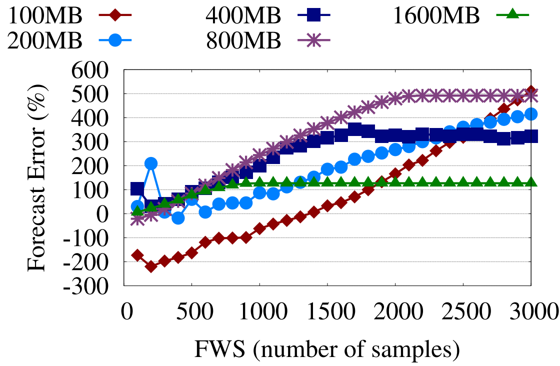
Fig. 5. FE Difference between Testing Strategies (real-time vs. offline).

and the heap size (major factors influencing the GC). The smaller the $MiGC_{AVG}$ is, the more frequent the application tends to exhaust its OldGen memory. If the value is near to zero (e.g., $\leq$10), the application is close to an Out-Of-Memory exception. On the contrary, a value far from zero (e.g., $\geq$ 1000) suggests that the OldGen is rarely exhausted. Precisely, this change of behaviour (i.e., gradually moving from a low $MiGC_{AVG}$ to a high value) is exhibited by Eclipse in Fig. 6.c: The smallest heap size (i.e., 100MB) is barely enough for its successful execution. This is reflected in its very low $MiGC_{AVG}$ (i.e., 2.99). Under these conditions, the lowest FE was achieved by the 1400 FWS (obtained from the extended offline results), while the second lowest FE was achieved by 1500 FWS (obtained from the real-time results). This means that, for this particular experimental configuration, we were lucky to have selected 1500 as one of the FWS evaluated in real-time. However, as observed in Fig. 6.a, 1500 was not a good FWS value for the other heap sizes. This was due to the changes of GC behaviour caused by the use of different (larger) heap sizes. This is because the increment in available memory provoked that the $MiGC_{AVG}$ steadily increased.

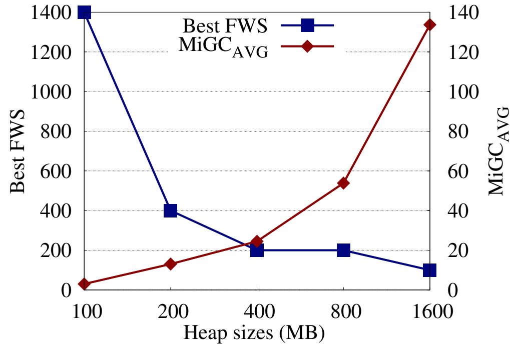
Based on the discussion presented, it is clear that this deepness of analysis could not have been performed with the small set of FWS values used in the real-time test runs. Thus, the use of an offline test strategy has proven useful to strengthen the testing of GC developments and derive more general conclusions. This is important because a major reason for only using a reduced set of experimental configurations is that, typically, GC-related advancements can only be evaluated in real-time (due to the lack of GC benchmarks). So, under those conditions, it might be unfeasible to do a large number of experiments due to the considerable amount of time required



(a) Real-time MaGa - FWS trend



(b) Offline MaGa - FWS trend



(c) BestFWS vs $MiGC_{AVG}$ trade-off

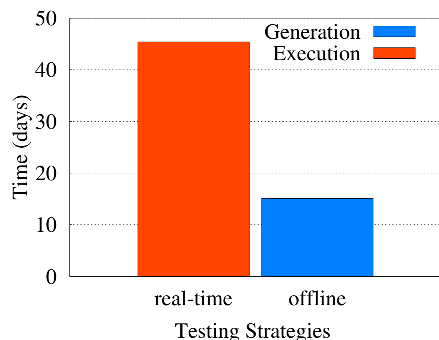
Fig. 6. Examples of Extended Evaluation and Analysis of GC Advancement.

to execute them. This analysis perspective is explored next.

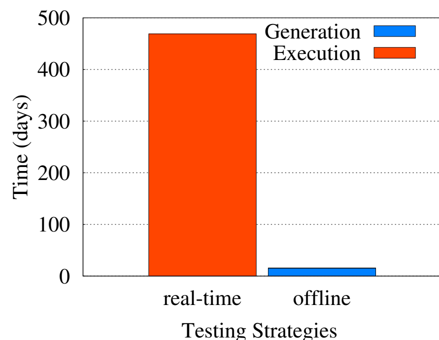
3) Time Savings: Next, we compared the time invested in running each test strategies (i.e., real-time and offline) to understand the time savings that HERMES achieved. This is an important analysis perspective because a key motivation of our work is to improve users' productivity. Fig. 7.a presents the time invested in running MaGa in real-time and offline mode with a small range of FWS. In the case of real-time MaGa, the only time component is the actual execution time of the test runs. In contrast, the offline MaGa's time is composed of two different types of time: The generation of the GC benchmark and the simulated execution of the test runs. The benchmark generation task was considered in this analysis to make a conservative comparison of the test runs. Another option is to assume that the GC benchmarks have already been created (e.g., all GC benchmarks generated for this paper have been made publicly available). By comparing the times in Fig. 7.a, it can be noted how significant time savings can be obtained by using the offline testing strategy: It required 30.26 fewer days (i.e., a 67% improvement) with respect to the real-time testing, while obtaining the same quality of results. Finally, the execution time, for the offline testing, is not observable in the figure due to being marginal (i.e., $<1\%$).

Regarding the generation of a GC benchmark, since this occurs in real-time, it has the same time cost that a real-time test. Nonetheless, it only needs to be performed once per combination of program, GC strategy, and heap size. In contrast, a real-time test needs to be repeated per FWS, which drastically increases the required time (as depicted in Fig. 7.a). The total time of testing real-time MaGa is directly related to the number of test runs (e.g., the range of FWS in our case). Thus, evaluating the lengthened range of FWS would have needed an excessive amount of time. Therefore, these combinations were exclusively tested with offline MaGa. This is illustrated in Fig. 7.b, which shows the extrapolated time required to test the lengthened range of 31 FWS values with the real-time MaGa. It can be observed how that test strategy is impractical, as it would require more than 450 days of continuous test executions. Consequently, the time savings of using HERMES are even bigger under these conditions (i.e., a 97% reduction in this case). This scenario also illustrates how the gains are higher when the GC benchmarks are reused. This is because the real-time test strategy involves a linear time growth (as shown in the figures). In contrast, the offline testing time is practically independent of the range of tested FWS (as the only significant time is the generation of the GC benchmarks). After that, any additional test runs (e.g., to evaluate other FWS values) have a marginal time cost (e.g., each offline test run using a GC benchmark, representing 1-hour of real-time execution, lasted an average of 4 seconds).

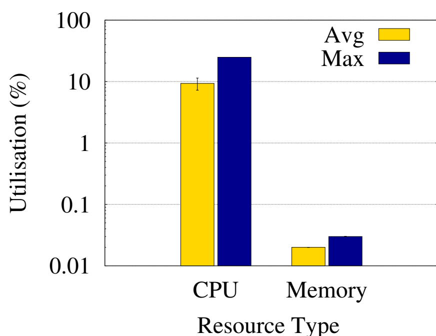
4) Cost Analysis: Finally, we also assessed the computational costs of using HERMES. The results are depicted in Fig. 7.c, which presents the percentages of CPU and memory utilisations of the GC benchmark generation process. HERMES proved to be lightweight in terms of CPU. For example, the CPU usage never surpassed 25% (indicating that there was a considerable amount of idle resources). Likewise, HERMES proved to be highly memory efficient. For instance, it never used more than 0.03% of the JVM memory (i.e., 33MB). This low usage of resources is the result of a carefully developed design, which purposely uses the minimal set of internal structures to keep track of the memory/GC behaviours, and a moderate number of Java classes loaded by the JVM to support the engine (e.g., the initial memory footprint was 5.5MB, which represents 0.01% of the JVM memory). It is important to highlight that HERMES itself never triggered a MaGC, which was an additional indicator that the memory settings were appropriate. The costs were also analysed per GC strategy and program. Although marginal differences were observed, the costs were very similar across the three GC



(a) Time Savings (3 FWS range)



(b) Time Savings (extended FWS range)



(c) Computational Costs

Fig. 7. Time Savings and Computational Costs of HERMES.

strategies, as well as the 24 tested programs. This is because the major factor contributing to the computational costs of HERMES is the sampling interval, which is configurable and independent of the chosen GC strategy or program. Based on the obtained results, it can be concluded that the costs of HERMES were acceptable, as our test environment was far from exhausting its resources (despite its modest characteristics).

C. Final Discussion

The results of our experiments have illustrated how HERMES can help to optimise and strengthen the evaluation of GC-related advancements. For example, our experiments showed how HERMES can help to better evaluate the accuracy of a MaGC forecast algorithm (MaGa) with a minimal time investment (while real-time experiments will still be necessary to understand its computational costs). HERMES can also be useful for early research phases, so that one can prove the validity of an idea without the cost of building (and running) a fully-functional prototype. That is, before the level of maturity justifies the costs involved in testing in real test environments (i.e., using real-time experimental configurations).

For our experimental evaluation, we used three of the GC strategies most widely-used in the industry. As the experimental results have shown, HERMES worked well for the three GC strategies, so, it is expected that it can produce similar results when used with other GC strategies. It is also possible to conclude that HERMES can be applicable to other technologies which rely on GC strategies and principles similar to those used by Java (e.g., generational heaps). For example, Python or C#. The same criterion applies to Java programs. In this work, we selected the DaCapo and SPECJVM benchmark suites because they are widely used in the literature and offer a varied set of program behaviours. However, our solution is not tied to them as HERMES can be used in conjunction with any other Java programs and JVMs (e.g., OpenJDK and IBM JVM). The current results are encouraging, as they have provided some evidence that the engine generalises across the tested experimental configurations. Nevertheless, additional experiments would provide more certainty about the broader applicability of our engine.

Regarding the overhead introduced by HERMES, the results have shown that the CPU and memory utilisations are very low. Nonetheless, if this level of overhead is intolerable for a particular usage scenario, there are ways this can be reduced. This decrement can be achieved by modifying the sampling interval to a higher value (as a 100ms value was used in this work). This configuration adjustment would have the effect of decreasing the frequency of the GC/memory sampling in the monitored Java program; hence, reducing the amount of computational resources required by HERMES. This would be done at the expense of increasing the likelihood of missing to sample a GC event. For this reason, we recommend that the sampling interval should not be higher than the average time elapsed between MiGC events if the aim is to precisely mimic the behaviour of the Java program (e.g., a MiGC sample should normally represent a single MiGC event). Alternatively, increasing the sampling interval might be useful to diversify (even more) the tested behaviours. This is because the higher the sampling interval is, the more likely that the sample will represent bigger chunks of memory allocation (i.e., appearing to be a more memory-intensive application). For instance, a gathered MiGC sample might represent the memory increment behaviour of multiple MiGCs (those which have been skipped between samples), instead of a single MiGC event.

Regarding the test strategy followed to demonstrate the benefits that HERMES can bring to the evaluation of GC-related advancements, we decided to illustrate it by strengthening the evaluation of MaGa's benefits (i.e., its forecast accuracy). This is because, in our own experience, a major challenge experienced when testing GC-related advancements is the non-deterministic nature of the GC, scenario which could be illustrated with the forecast accuracy of MaGa. In this use

case, HERMES can help to remove (for reliability/consistency testing purposes) the non-deterministic nature of the GC by taking snapshots of GC/memory behaviours (i.e., the generated GC benchmarks). Later, they can be reused to be certain that the same GC behaviour is tested across all the test runs that used a particular GC benchmark (implicitly implying a specific combination of a GC strategy, heap size, and Java program). However, other test strategies might be used. For instance, if a user wants to keep some amount of (controlled) variability, s/he might run the same experimental configurations multiple times (in HERMES) to create different GC benchmarks based on the same combination of GC strategy, heap size, and Java program. Then, s/he can use the generated sets of GC benchmarks (e.g., 100 different GC benchmarks per experimental configuration) to test his/her GC-related advancements offline. Despite the differences in the particular test strategy utilised, it is expected that the time savings gained by using HERMES would be still significative. This is because the offline test runs that use the GC benchmarks generated by HERMES do not require the need of spending costly calendar (i.e., real) time.

V. Conclusions and Future Work

This paper presented HERMES, an engine for the generation of realistic Garbage Collection (GC) benchmarks that allows to properly capture the different GC/memory behaviours exhibited by Java applications. The aim was to offer researchers a useful tool to optimise and strengthen the validation of GC-related solutions. Our results showed HERMES can help to broaden the number and diversity of GC testing scenarios (the generated GC benchmarks are freely-available [10]). Furthermore, our tests demonstrated the time savings that HERMES can bring to the evaluation of GC-related technologies. Finally, results proved that HERMES is lightweight in computational resources, making it practical for real-world usage.

As future work, we plan to extend the capabilities of HERMES by supporting other types of snapshot resources (e.g., CPU utilisation) and GC-based technologies (e.g., .NET or Python). We will also strengthen the evaluation of HERMES (e.g., by diversifying the tested application behaviours and JVMs). Finally, we plan to enhance the quality of our source code, implement a graphical user interface, and extend the set of supported Java programs and JVMs in order to release HERMES as a publicly-available tool.

Acknowledgments

This work was supported, in part, by Science Foundation Ireland grant 13/RC/2094 and co-funded under the European Regional Development Fund through the Southern & Eastern Regional Operational Programme to Lero - the Irish Software Research Centre (www.lero.ie).

References

[1] Oracle, “Memory Management in the Java HotSpot Virtual Machine,” *April*, 2006.

[2] A. Hussein, M. Payer, A. L. Hosking, and C. Vick, “One process to reap them all: Garbage collection as-a-service,” in *International Conference on Virtual Execution Environments*, 2017, pp. 171–186.

[3] P. R. Wilson, “Uniprocessor GC Techniques,” in *International Workshop of Memory Management*, 1992.

[4] G. Phipps, “Comparing Observed Bug and Productivity Rates for Java and C++,” *Software: Practice and Experience*, April 1999.

[5] A. O. Portillo Dominguez, “Performance optimisation of clustered Java systems,” Ph.D. dissertation, University College Dublin, 2016.

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

[7] S. M. Blackburn, P. Cheng, and K. S. McKinley, “Myths and realities: The performance impact of garbage collection,” *ACM SIGMETRICS Performance Evaluation Review*, vol. 32, no. 1, pp. 25–36, 2004.

[8] S. B. Joshi, “Apache hadoop performance-tuning methodologies and best practices,” in *International Conference on Performance Engineering*, 2012, pp. 241–242.

[9] S. Becker, H. Koziolek, and R. Reussner, “The palladio component model for model-driven performance prediction,” *Journal of Systems and Software*, vol. 82, no. 1, pp. 3–22, 2009.

[10] “HERMES’ GC Benchmarks,” Last accessed: 2018-07-20. [Online]. Available: https://github.com/ucd-pel/GCBenchmark

[11] “Java SE 6 HotSpot Virtual Machine Garbage Collection Tuning,” Last accessed: 2018-07-20. [Online]. Available: http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html

[12] H. Kasture and D. Sanchez, “Tailbench: a benchmark suite and evaluation methodology for latency-critical applications,” in *International Symposium on Workload Characterization*, 2016.

[13] Y. Sun, X. Gong, A. K. Ziabari, L. Yu, X. Li, S. Mukherjee, C. McCardwell, A. Villegas, and D. Kaeli, “Hetero-mark, a benchmark suite for CPU-GPU collaborative computing,” in *International Symposium on Workload Characterization*, 2016, pp. 1–10.

[14] J. Wadden, V. Dang, N. Brunelle, T. Tracy II, D. Guo, E. Sadredini, K. Wang, C. Bo, G. Robins, M. Stan *et al.*, “ANMLzoo: a benchmark suite for exploring bottlenecks in automata processing engines and architectures,” in *International Symposium on Workload Characterization*, 2016, pp. 1–12.

[15] “The DaCapo Benchmark Suite,” Last accessed: 2018-07-20. [Online]. Available: http://dacapobench.org/

[16] “SPECjvm 2008,” Last accessed: 2018-07-20. [Online]. Available: http://www.spec.org/jvm2008/

[17] A. O. Portillo-Dominguez and V. Ayala-Rivera, “Improving the testing of clustered systems through the effective usage of java benchmarks,” in *International Conference in Software Engineering Research and Innovation*, 2017.

[18] A. O. Portillo-Dominguez, “Towards an efficient benchmark generation engine for garbage collection,” in *International Conference on Performance Engineering*, 2018.

[19] F. Pizlo, E. Petrank, and B. Steensgaard, “A study of concurrent real-time garbage collectors,” *ACM SIGPLAN Notices*, vol. 43, no. 6, 2008.

[20] F. Xian, W. Srisa-an, C. Jia, and H. Jiang, “AS-GC : An Efficient Generational GC for Java Application Servers,” in *European Conference on Object-Oriented Programming*, 2007.

[21] A. O. Portillo-Dominguez, M. Wang, J. Murphy, and D. Magoni, “Adaptive GC-Aware load balancing strategy for high-assurance java distributed systems,” in *International Symposium on High Assurance Systems Engineering*, 2015.

[22] “Tuning JVMs,” Last accessed: 2018-07-20. [Online]. Available: https://docs.oracle.com/cd/E15523_01/web.1111/e13814/jvm_tuning.htm

[23] “Java Language and Virtual Machine Specifications,” Last accessed: 2018-07-20. [Online]. Available: https://docs.oracle.com/javase/specs/

[24] A. O. Portillo-Dominguez, M. Wang, D. Magoni, P. Perry, and J. Murphy, “Load balancing of java applications by forecasting garbage collections,” in *International Symposium on Parallel and Distributed Computing*, 2014, pp. 127–134.

[25] “Java SE HotSpot at a Glance,” Last accessed: 2018-07-20. [Online]. Available: http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136373.html

[26] A. O. Portillo-Dominguez, P. Perry, D. Magoni, M. Wang, and J. Murphy, “TRINI: an adaptive load balancing strategy based on garbage collection for clustered java systems,” *Software: Practice and Experience*, 2016.

[27] V. Ayala-Rivera, A. O. Portillo-Dominguez, L. Murphy, and C. Thorpe, “COCOA: A synthetic data generator for testing anonymization techniques,” *International Conference on Privacy in Statistical Databases*, 2016.