| | |
|---|---|
| ***Title*** | Towards an Efficient Benchmark Generation Engine for Garbage Collection |
| ***Authors(s)*** | Portillo Dominguez, Andres Omar |
| ***Publication date*** | 2018-04-13 |
| ***Publication information*** | ICPE '18 Companion of the 2018 ACM/SPEC International Conference on Performance Engineering |
| ***Conference details*** | ACM/SPEC International Conference on Performance Engineering Companion, Berlin, Germany, April, 2018 |
| ***Publisher*** | ACM |
| ***Item record/more information*** | http://hdl.handle.net/10197/9959 |
| ***Publisher's statement*** | © ACM, 2018. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in ICPE '18 Companion of the 2018 ACM/SPEC International Conference on Performance Engineering (2018) http://doi.acm.org/10.1145/10.1145/3185768.3186303 |
| ***Publisher's version (DOI)*** | 10.1145/3185768.3186303 |

# Towards an Efficient Benchmark Generation Engine for Garbage Collection

A. Omar Portillo-Dominguez
Lero@UCD, School of Computer Science, University College Dublin, Ireland
andres.portillodominguez@ucd.ie

## ABSTRACT

Garbage Collection (GC) is a key feature of many modern programming technologies. It offers significant software engineering benefits over explicitly memory management. Nonetheless, it is also a major cause of performance degradation. As the rate of adoption of GC-related technologies continues to grow, it is highly relevant to understand its performance impact. However, this is challenging due to the non-deterministic nature of GC. To tackle this problem, we present an engine (HERMES) to create realistic GC benchmarks by effectively capturing the GC/memory behaviours exhibited by real-world Java applications. Our experiments prove how HERMES can be useful to strengthen the evaluation of GC-related advancements. This is achieved by broadening the number and diversity of the test scenarios, as well as reducing the time invested in testing.

## KEYWORDS

Garbage Collection; Benchmark Generation; Java; Object-Oriented Systems

## 1 INTRODUCTION

GC is an essential process of many of the most widely-used runtime platforms (e.g., Java, Android) which automates most of the memory-related tasks. This translates into significant software engineering benefits (e.g., it helps to avoid the most common sources of memory leaks [10]). Despite its benefits, GC has a (potentially significant) impact on the system performance by pausing the involved programs [7]. Due to the complex interactions between the GC and the many factors that affect its behaviour, its exact influence on a particular application's performance is hard to define. For example, this is why the GC overhead is often modelled as a constant factor that needs to be calibrated along with other parameters in the performance engineering community [4].

To help address these challenges, this paper presents a novel automated engine (HERMES) to generate realistic GC benchmarks which can capture the complex and time-varying GC behaviour of Java applications. It also presents a practical evaluation of HERMES and 210 publicly available GC benchmarks [1] (based on real-life Java applications that span a wide range of diverse GC/memory behaviours).

## 2 RELATED WORK

Multiple research efforts have aimed to improve GC performance. For instance, several new concurrent and parallel algorithms with smaller performance impacts have been proposed [9]. Other works have developed solutions for particular scenarios. For example, the authors of [13] proposed a load balancing strategy to address the performance impacts of the Major GC. Regarding the works that have modelled the GC, its overhead is typically captured as a constant factor to be calibrated with other parameters [4, 8]. In contrast to these works, that have aimed to improve the behaviour of the GC process, our goal has been to develop, to the best of our knowledge, the first engine to automatically generate GC benchmarks that properly captures the complexities in GC and memory behaviours that occur in Java applications.

## 3 PROPOSED SOLUTION

HERMES follows a process composed of three phases. They are depicted in Fig. 1 and explained next:

**Initialisation:** It starts by setting all the configured input parameters. These consist of six user inputs that define the desired characteristics of the GC benchmark as well as the program's execution: (1) the JVM over which the program will be run; (2) the GC strategy that will be used; (3) the heap size that will be used; (4) the program that will be run; (5) how long the program needs to be executed; and (6) a sampling interval to indicate how often GC/memory samples will be collected from the monitored JVM. Also, the user can input the set of specific parameters that a particular program might require to be properly executed. Next, the other two phases of the process are concurrently started.

**Executor:** It starts by setting up the experimental environment that the chosen program needs to be executed successfully (e.g., any required dependencies). 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. Once the program has started, the Executor reports the success of this operation to the Collector, so that it can continue its initialisation. 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.

**Collector:** It starts by initialising the data models that will be used to 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., with metrics like maximum, committed, used, and free

The final publication is available at IEEE Xplore Digital Library via https://doi.org/10.1145/3185768.3186303

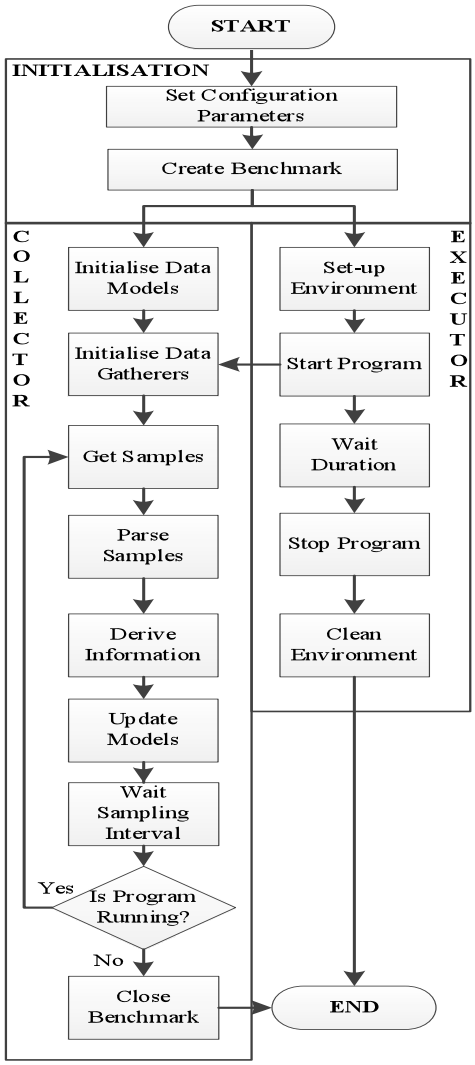



**Figure 1: HERMES' Core Process**

memory). (ii) Another suitable for a GC type, which captures information such as the memory usage before and after its occurrence. Next, the data gathering that is required to collect the information is initialised. 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 and the GC types. 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 derived metrics that complement the observed GC behaviour by properly quantifying the changes in each of the monitored elements. For example, for the MaGC, it involves the calculation of the number of MiGCs that triggered this event. This metric was selected because several works [11, 12] have shown that the MiGC statistics are useful to characterise the GC behaviours experienced by applications. For this same reason, the set of metrics derived from the MiGC information is more extensive (e.g., allocations per memory region). Once all metrics have been calculated, the engine's models are updated. 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.

## 4 EXPERIMENTAL EVALUATION

**Setup:** Our experiments pursued: (1) to prove 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. To achieve this, we conducted two series of experiments. First, a set of GC benchmarks was created using our developed prototype. Then, the benchmarks were used in the evaluation of a GC-related advancement. The prototype was developed in Java. We initially focused on supporting the Oracle HotSpot JVM (as it is one of the leading commercial JVMs [2]) and the generational heaps (as it is the most common type of heap [14]). Regarding Java programs, we selected the DaCapo benchmark [6] as it is one of the Java benchmarks most widely-used in the literature, offering 14 different real-life programs with non-trivial memory loads.

**Bechmark Diversity:** To evaluate HERMES, we generated 210 GC benchmarks. This was achieved by running all the possible combinations derived from the execution of the 14 DaCapo programs with the 3 most widely used GC strategies [14] (i.e., Serial -sGC- , Parallel -pGC- , and Concurrent -cGC- ), and 5 heap sizes (100, 200, 400, 800, and 1600MB). Different heap sizes and GC strategies were chosen as they are major factors affecting the GC behaviour [5]. The program executions were set to 60 mins; the JVM heap initialised to its maximum size with the calls to programmatically request a MaGC disabled. Also, a 100ms sampling interval was used (as it is appropriate for DaCapo [11]).

To assess the diversity among the generated GC benchmarks, we used the Principal Components Analysis (PCA), which is a statistical technique used to assess the dissimilarity among benchmarks [3]. 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 similar to the one used in other works [3, 6]). We ran the PCA on the GC benchmarks (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, 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. 2, which shows how the GC benchmarks differ in a two-dimensional space (PC1 vs. PC2) for the cGC-100MB combination. Similar results were obtained for the other GC strategies and heap sizes.

**GC-related Advancement:** To assess if the GC benchmarks generated by HERMES could improve the testing of GC-related

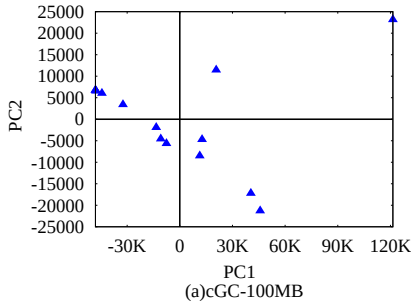



**Figure 2: PC1 vs PC2 of GC Benchmarks**

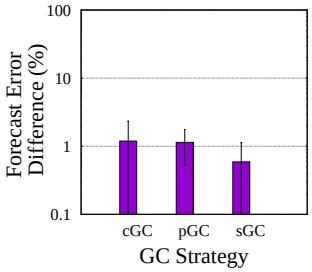



**Figure 3: FE Difference between Testing Strategies (real-time vs. offline)**

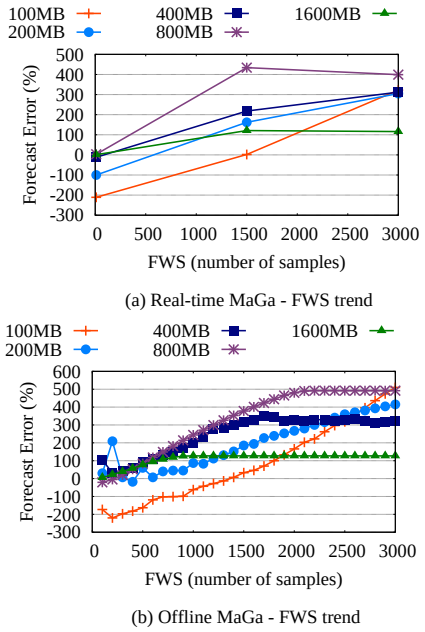


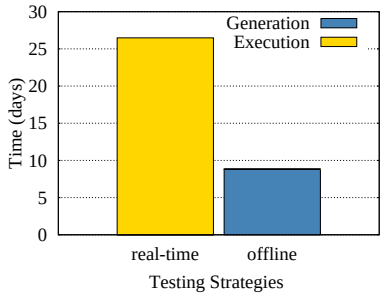

**Figure 4: Examples of Extended Evaluation and Analysis of GC Advancement: MaGa Algorithm**

advancements, we applied them in the evaluation of a forecast algorithm (denominated MaGa) which predicts MaGC events [12]. 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 CSV 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. Moreover, the same set of Java programs, heap sizes, GC strategies, and sampling interval used in the GC benchmark generation were also used here. Finally, to measure MaGa's accuracy, we utilised the forecast error (FE) [12], metric which captures the difference between the forecasted time and the real time of a MaGC event (represented as a proportion of the time elapsed since the previous MaGC).

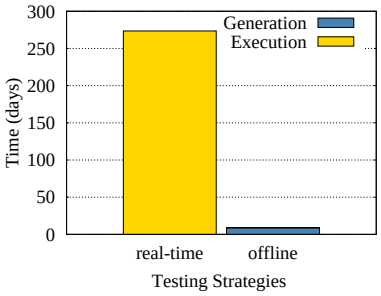
Once all test runs were executed, the FEs of the MaGC predictions were calculated and the differences between the two types of runs were compared (i.e., the FWS values of 100, 1500, and 3000). Although the results varied among the GC strategies, this analysis showed that the offline test runs were able to achieve similar FEs (compared to their real-time counterparts). These results are depicted in Fig. 3, 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. 4.a and 4.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 sensitive to the chosen FWS, it is important to know which is the "best FWS" for a particular case. In this context, the best FWS is defined as the FWS that achieves the highest accuracy (i.e., a FE of zero, or the closest to zero). If we try to use the information of Fig. 4.a to analyse this, there are no FWS values that fit well the definition of best FWS 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 number of test results (e.g., to determine whether an increment in the FWS improves or deteriorates the accuracy of the algorithm). In contrast, Fig. 4.b would allow performing a more detailed analysis, as the FWS trends for each of the heap sizes can be clearly identified. For instance, if we review the behaviour of the 100MB heap, it can be observed how the accuracy is considerably poor (i.e., FE $> \pm 100\%$) when FWS $\leq 1000$ or FWS $\geq 2000$). A similar analysis can be done for the other heaps, where it is always possible to find an FWS (or range of FWS) where the FE is significantly close to 0%.

Based on the above, 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 about their performance. This is important as a major reason for only using a reduced set of experimental configurations is that, typically, GC-related advancements




(a) Time Savings (3 FWS range)




(b) Time Savings (extended FWS range)

**Figure 5: Time Savings of HERMES**

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 to execute them.

**Time-Savings:** To understand the time savings that HERMES achieved, we compared the time invested in running each of the two test strategies. Fig. 5.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 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 more conservative comparison of the test runs. Alternatively, one could consider that the GC benchmarks have already been created (e.g., the 210 GC benchmarks generated for this paper have been made publicly available). By comparing the times in Fig. 5.a, it can be noted how significant time savings can be obtained by using the offline testing strategy: It required 17.63 fewer days (i.e., a 67% improvement) with respect to the real-time testing, while obtaining the same quality of results. It is worth mentioning that the biggest part of the total offline time was the generation of the GC benchmarks (i.e., 99%); while the GC execution time was marginal (only 0.5 hours), so it is not observable in the figure.

Regarding the generation of a GC benchmark, since this occurs in real-time, it has the same time cost that a real-time test. However, 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. 5.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 extended range of FWS would have required an excessive amount of time. Hence, these combinations were exclusively tested with offline MaGa. This is illustrated in Fig. 5.b, which shows the extrapolated time needed to test the extended 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 250 days of continuous test executions. Thus, 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 if the GC-related advancements (e.g., MaGa) are tested in real-time, that strategy involves a linear growth. In contrast, by using the generated GC benchmarks, the testing time is practically independent of the range of tested FWS (as the only relevant time is the generation of the GC benchmarks). After that, any additional test runs have a marginal time cost (e.g., 4 secs in our experiments).

## 5 CONCLUSIONS AND FUTURE WORK

This paper presented HERMES, an engine to generate realistic GC benchmarks that properly captures the GC behaviours exhibited by Java applications. Results have shown how HERMES can broaden the diversity of testing scenarios. Also, our tests illustrated the time savings that HERMES can bring to the evaluation of GC-related technologies. As future work, we plan to continue strengthening HERMES' capabilities (e.g., by diversifying the tested application behaviours, and the modelled memory/GC attributes).

## 6 ACKNOWLEDGMENTS

This work was supported, in part, by Science Foundation Ireland grant 13/RC/2094.

## REFERENCES

[1] Benchmarks. https://github.com/ucd-pel/GCBenchmark. Last accessed: 2018-02-10.
[2] Java SE HotSpot at a Glance. http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136373.html. Last accessed: 2018-02-10.
[3] V. Ayala-Rivera, A. O. Portillo-Dominguez, L. Murphy, and C. Thorpe. COCOA: A synthetic data generator for testing anonymization techniques. *PSD*, 2016.
[4] S. Becker, H. Koziolek, and R. Reussner. The palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82(1):3–22, 2009.
[5] S. M. Blackburn, P. Cheng, and K. S. Mckinley. Myths and Realities: The Performance Impact of Garbage Collection. *SIGMETRICS PER*, Jan. 2004.
[6] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *ACM Sigplan Notices*, volume 41, pages 169–190, 2006.
[7] P. Lengauer and H. Mössenböck. The taming of the shrew: Increasing performance by automatic parameter tuning for Java GC. In *ICPE*, 2014.
[8] P. Libic, L. Bulej, V. Horky, and P. Tuma. On the limits of modeling generational garbage collector performance. In *ICPE*, pages 15–26, 2014.
[9] F. Pizlo, E. Petrank, and B. Steensgaard. A study of concurrent real-time garbage collectors. *ACM SIGPLAN Notices*, 43(6), 2008.
[10] A. O. Portillo Dominguez. Performance optimisation of clustered java systems. 2016.
[11] 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. *Softw. Pract. Exp.*, 2016.
[12] A. O. Portillo-Dominguez, M. Wang, D. Magoni, P. Perry, and J. Murphy. Load balancing of java applications by forecasting garbage collections. In *ISPDC*, 2014.
[13] A. O. Portillo-Dominguez, M. Wang, J. Murphy, and D. Magoni. Adaptive GC-aware load balancing strategy for high-assurance java distributed systems. In *HASE*, 2015.
[14] Sun Microsystems. Memory Management in the Java HotSpot Virtual Machine. *April*, 2006.