SmartGC: Online Memory Management Prediction for PaaS Cloud Models

José Simão[1,3(✉)], Sérgio Esteves[1,2], and Luís Veiga[1,2]

[1] INESC-ID Lisboa, Lisbon, Portugal
jsimao@cc.isel.ipl.pt
[2] Instituto Superior Técnico, Universidade de Lisboa, Lisbon, Portugal
sesteves@gsd.inesc-id.pt, luis.veiga@inesc-id.pt
[3] Instituto Superior de Engenharia de Lisboa, Instituto Politécnico de Lisboa, Lisbon, Portugal

Abstract. In Platform-as-a-Service clouds (public and private) an efficient resource management of several managed runtimes involves limiting the heap size of some VMs so that extra memory can be assigned to higher priority workloads. However, this should not be done in an application-oblivious way because performance degradation must be minimized. Also, each tenant tends to repeat the execution of applications with similar memory-usage patterns, giving opportunity to reuse parameters known to work well for a given workload. This paper presents *SmartGC*, a system to determine, at runtime, the best values for critical heap management parameters of JVMs. *SmartGC* comprises two main phases: (1) a training phase where it collects, with different heap resizing policies, representative execution metrics during the lifespan of a workload; and (2) an execution phase where it matches the execution parameters of new workloads against those of already seen workloads, and enforces the best heap resizing policy. Distinctly from other works, this is done without a previous analysis of unknown workloads. Using representative applications, we show that our approach can lead to memory savings, even when compared with a state-of-the-art virtual machine - OpenJDK. Furthermore, we show that we can predict with high accuracy the best heap policy in a relatively short period of time and with a negligible runtime overhead. Although we focus on the heap resizing, this same approach could also be used to adapt other parameters or even the GC algorithm.

Keywords: Garbage collection · Machine learning · Shared execution environment · Java Virtual Machine

1 Introduction

Managed runtimes, such as the Java Virtual Machine (JVM), have been increasingly used in large-scale deployments and, particularly, in cloud environments [22]. Platform-as-a-Service providers (e.g., Heroku, AppFog and Google

H. Panetto et al. (Eds.): OTM 2017 Conferences, Part I, LNCS 10573, pp. 370–388, 2017.
https://doi.org/10.1007/978-3-319-69462-7_25

App Engine) allow the deployment of workloads on high-level language virtual machines (HLL-VMs) on a multi-tenant environment. In recent years, several middleware frameworks have been developed for systems that target these runtimes. These frameworks cover several areas, such as, graph processing [9] or bio-informatics [12,14]. Furthermore, regardless of the language of choice and the development paradigm (more object-oriented or more functional), in most cases, the provider will run the resulting components in a JVM-compatible runtime, or in a runtime that has similar code generation and memory management challenges, such as the one that supports node.js - the V8 engine.[1]

Although several resources have elasticity, that is, resources can be removed or assigned without breaking the application execution, memory is one with high impact. To take advantage of the resources available in the cluster supporting the cloud, a managed runtime for these environments needs a synergy of mechanisms and allocation strategies. These mechanisms should include local adaptations to the consumption of resources (where memory must have a prominent attention) and ways to infer application progress (or progress rate).

Uninformed consolidation carries a negative impact on the application performance. With clever choices, providers can therefore consolidate more VMs on the same hardware, in terms of memory, and save costs (and reduce prices or increase revenues) without significantly worsen application performance. A policy with high-impact in the performance of managed runtimes is the one that manages memory. It has a dual effect: a direct impact on the memory allocated to the process but also a impact on the progress rate of the application. This opens the possibility to configure critical parameters of the Garbage Collector (GC) based on the application *signature* (i.e. resource usage behaviour profile, or type profile) and the correlation with a set of configurations that are previously known as favoring consolidation. The *signature* can be determined by metrics obtained from the hardware, operating system and the runtime itself. During applications execution, different access and usage patterns of memory and CPU-related structures can be identified. Previous work has reviewed the performance of programs using low-level performance metrics, namely, hardware performance counters, such as cache misses and instructions per cycle [16].

There is however no previous work that uses such information to categorize applications in terms of memory signatures, so that relevant parameters can be dynamically reconfigured. Because workloads exhibit dynamic patterns, the reconfiguration of the heap management policy should be designed as an adaptive process relying on timely collection of performance counters, identification of an application profile and the choice of the best parameters accordingly.

This paper describes *SmartGC*, a system that adapts the heap management policy of managed runtimes based on the identification of an application execution profile – a *signature* – and the relation of this signature with parameters that are known to maximize the execution yield, i.e., the relation between memory usage and execution time of the application. Although the system can classify unseen workloads, the offline signatures dataset can be easily extended with

[1] http://cloud.google.com/appengine.

new applications. The main contributions include: (i) a low-overhead machine learning-based classification system, based on a small set of performance counters, collected during the execution of applications that target the Java VM; (ii) an adaptive GC control for multi-tenancy to reduce memory footprint of VMs with small performance impacts, hence improving resource effectiveness and provider revenue. *SmartGC* currently supports alternative heap resizing policies with influence in the allocated memory and application performance.

The paper is organized as follows. Section 2 shows the impact of heap size management policies in memory savings and execution performance, presenting a metric that relates these observations. Section 3 describes the most effective metrics to characterize the execution of an application and how this characterization can be represented in what we call a *signature*. Section 4 describes the design of *SmartGC*, including the structure of a workload *signature* and the two distinct phases of operation. Section 5 discusses the details of techniques used in the classification adopted, and Sect. 6 makes an extensive evaluation of the major components of *SmartGC*, including the comparison with a widely deployed JVM, the OpenJDK. Section 7 makes an analysis to the state of the art and Sect. 8 closes the paper discussing future work.

2 The Case for Execution Yield

SmartGC goal is to reconfigure parameters of managed runtimes, saving memory while minimizing performance penalties. So, in our system we consider the ratio between the savings in memory, as a resource (ΔR), and the degradation in execution performance (ΔP), when compared with a large enough heap with a fixed size. We call this ratio the *execution yield*: $\frac{\Delta R}{\Delta P}$.

The parameters to be changed at runtime are JVM-specific. In this work we focus on experiences with the Jikes RVM [3]. This JVM uses a matrix to control how the heap size is modified as the application progresses. The matrix relates the percentage of live objects and the time used in GC operations to determine whether the heap will grow or shrink to a new size. The default matrix is the first presented in Fig. 1. It relates the time spent in GC activity (versus mutator activity) to the ratio of live objects. For each pair of these values, the heap will grow (positive values) or shrink (negative values) a certain percentage.

We have proposed new alternative matrices, depicted in Fig. 1 [25,26]. These new matrices explore different shrink/grow percentages across an imaginary four-quadrant space. For example, $Q2$ of M_1 is more heap conservative, meaning that, when a small percentage of time is spent on GC and live objects remain below 30%, the heap will decrease between 30% and 45%, while in M_0 the same situation implies a resizing between -10% and $+10\%$.

Previous work typically only takes into account the heap size as seen from inside the virtual machine [7,10]. Since we target consolidated execution environments, we consider a lower-level metric, the proportional set size (PSS). This metric is deeply connected with the effective use of physical memory pages, the actual resource virtual machines compete for in cloud environments. The PSS

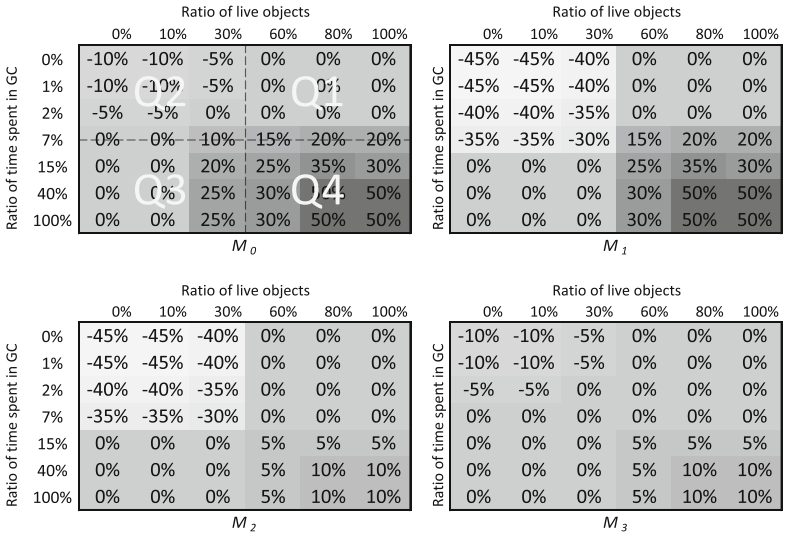
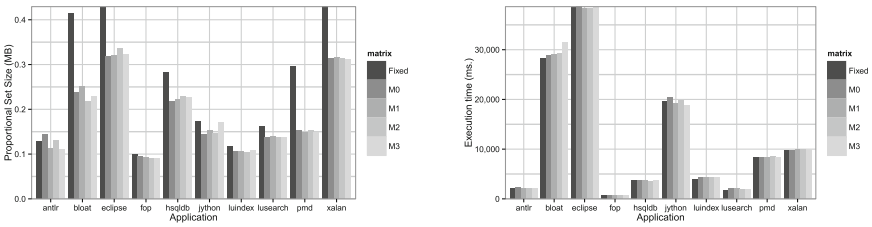


Fig. 1. Matrices options space

is reported by the virtual memory management sub-system. It is described, in Linux virtual memory documentation, as the set of pages a process has in memory, where each shared page is divided by the number of processes sharing it.

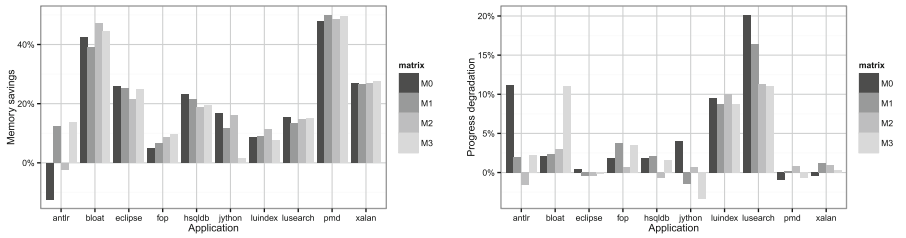
To find the best matrix, we executed a *reference set* of applications which use memory with different patterns. The goal is to correlate each of these reference applications to the matrix that maximizes the execution yield. These applications are from the Dacapo benchmark [5]. Each of these applications explore a different issue of the JIT, GC and micro-architecture, as extensively demonstrated in [5]. We expect these benchmarks represent full applications or their phases, and that the execution patterns they have regarding the use of memory are representative of other Java applications.

Figure 2a presents the memory saved by each of the four matrices, which are used within a dynamically-sized heap, when compared to a heap with a fixed size.



a. Proportional Set Size b. Execution time

Fig. 2. (a) Proportional set size and (b) Execution time, of several representative applications using different heap management policies



a. Proportional Set Size b. Execution time

Fig. 3. Percentage of (a) Memory savings and (b) Progress degradation, of several representative applications using different heap management policies

Figure 2b presents the correspondent progress degradation. Figure 3a details the percentages of saved memory by each of the four matrices, which are used within a dynamically-sized heap, when compared to a heap with a fixed size. Figure 3b shows the correspondent progress degradation. We can see that 4 applications can save 20% or more of memory, while the vast majority can save between 10% and 20%. At cloud data centers scale, the potential cumulative savings are very relevant. We also note that, for example, *pmd*, a code analysis workload, can save more than 40%. Regarding progress degradation, it is limited to 20%, and in most cases it is below 5%, almost at the level of variance in execution times of several runs. In some few cases, there are a negative value, which means that the corresponding matrix actually results in a progress gain.

The ratio between saving and degradation results in a different yield for each pair of reference application and matrix. This yield is an opportunity, for the provider, to use tailored memory-saving parameters without imposing a significantly perceivable, or at all, progress penalty in each workload. Larger numbers reveal that the resources saved are several times higher than any imposed penalty. This essentially "releases" resources to other workloads that will be able to make better progress, allowing to effectively channel resources, at each moment, to where they will pay out more efficiently. The mapping between application and best heap resizing matrix is presented next: (**antlr**, M_1), (**bloat**, M_0), (**eclipse**, M_3), (**fop**, M_2), (**hsqldb**, M_0), (**jython**, M_2), (**luindex**, M_2), (**lusearch**, M_3), (**pmd**, M_1), (**xalan**, M_3). This was established by running each application with a fixed heap of 350 MiBytes (a value for which these applications exhibit low number of GCs [29]) and with each of the depicted matrices. Common practices to avoid the non-determinism inherent to the adaptive compiler were used (e.g. replay compilation).[2]

After establishing the relationship between each type of workload and the best matrix, we now need to find a set of execution characteristics to be used during the runtime identification of any given application.

[2] http://jikesrvm.org/Experimental+Guidelines.

3 Transparent Profiling of Workloads

There are several runtime characteristics that can be explored to build the profile of an application. Common indicators include: (i) hardware performance counters; (ii) operating system performance counters; (iii) managed runtime specific metrics; (iv) and application specific metrics. From these four indicators, application specific metrics are the less reliable. They can typically be either related to the organization of classes, or to the nature of operations performed (e.g., rate of transactions processed). Collecting these metrics at the application level is a cumbersome task and makes difficult the correlation of memory usage patterns across different applications. Following, we describe the runtime resource indicators used by *SmartGC* (Sect. 3.1) to characterize and cluster different workloads according to their memory usage patterns (Sect. 3.2).

3.1 Performance Counters

Modern CPUs support a large set of performance counters, including instructions per cycle, branch misses and L1 cache misses. Operating systems also report performance related information, such as page faults and context switches.

Using performance counters introduces some difficulties, namely: (i) selecting the appropriate number and types of performance counters for our purposes; (ii) normalizing their values across different workloads.

Regarding the first issue, we must avoid using very processor-specific hardware performance counters so that the profile to be built can be reused with new hardware. When considering Intel's© processor families, the group of *architecture performance events* ensure consistent values across different processor implementations. This group includes counters such as the number of cycles and the last level cache references. Regarding performance events supported by operating systems, it is common that counters such as page faults as exported to be easily consumed in user-space.

Regarding the second issue, normalization of performance counters across different workloads, it is necessary to capture tendencies and perform regression so that workloads can be clustered based not on the magnitude of the PCs values, but on composed relative values (e.g., growth rate).

SmartGC uses several performance counters, collected periodically, mixing both memory-related and computation-related counters, including, computation-related counters (instructions, cpu-cycles, ref-cycles); cache statistics (cache-references, cache-misses); major and minor page faults (major-faults, minor-faults) and translation lookaside buffer statistics (dTLB-stores, dTLB-loads, dTLB-load-misses, dTLB-store-misses).

3.2 Workload Signature and Mnemonics

Workloads are clustered using what we call a *signature*. A *signature* is an aggregated description computed from the considered PCs that identifies a given workload, W, in terms of its resource usage. We assume that performance counters,

regardless of their nature, report a single value in each read. So, a signature contains an aggregated value for each performance counter that is computed during a period of time (e.g., lifespan of an application or a shorter period). This aggregated value represents the growth rate of a set of relevant performance counters ($pc_1 \dots pc_N$), between time t_i (initial time) and t_f (final time), as depicted in Eq. 1.

$$S_w(t_i, t_f) = Aggr(pc_1, t_i, t_f), \dots, Aggr(pc_N, t_i, t_f) \tag{1}$$

Currently, we support 2 different forms for aggregating sequences of performance counters, both taking into account the values that are available at Δt intervals. The first one is the mean of differences, as described in Eq. 2.

$$Diffs(pc_k, t_i, t_f) = \frac{1}{m} \sum_{x=t_i+\Delta t}^{t_f} pc_k(x) - p_k(x-1) \text{ where } k \in [1, N] \tag{2}$$

The other option is a geometric mean, as described in Eq. 3.

$$GMean(pc_k, t_i, t_f) = \Big(\prod_{x=t_i+1}^{t_f} pc_k(x) \Big)^{1/(tf-ti)} \text{ where } k \in [1, N] \tag{3}$$

A *mnemonic* can then be built, associating a signature to the correct best set of parameters, $\{P_1, P_2, \dots, P_R\}$, for the executing application. Equation 4 represents a mnemonic for application W, where signature S_w is associated to the best set of parameters.

$$M_w = (S_w \rightarrow \{P_1, P_2, \dots, P_R\}) \tag{4}$$

An example of parameters are the matrices presented in Sect. 2. In this case, a single parameter, i.e., a matrix number, can capture a multi-dimensional associations between ratio of live objects and ratio of time spent in GC operations. The next section will detail the system design of *SmartGC*, including its two main phases – training and runtime operation.

4 System Design

Analytical modelling is a common approach to inform resource-allocation systems [18,24]. Models can be used to predict the impact of management decisions on performance, availability, and/or energy consumption. However, constructing a model of a real system is a complex task. As a consequence, not rarely models are made with over-simplified assumptions so that they can be mathematically manageable. To overcome this, researchers have developed systems for experiment-based management of virtualized data centers [18]. *SmartGC* uses this strategy, and could easily be plugged into such systems. Figure 4 shows the overall system view. *SmartGC* acts in two distinct phases: the *training* phase and the *execution* phase. In some cases, a third phase can be used to test and reinforce the quality of the training.

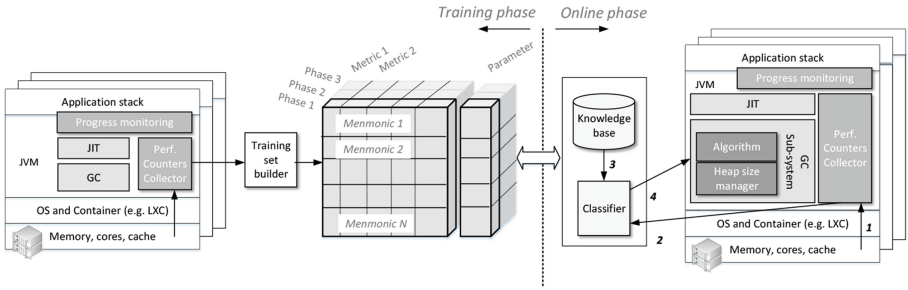


Fig. 4. System design

4.1 Training Phase

In the training phase, a set of representative applications are executed while system metrics are collected (memory and computation-related). This information is aggregated to build a training set, i.e., a set of application *signatures*, where each *signature* is associated with the heap management matrix that maximizes the *execution yield*. During the second phase, also known as online or execution phase, information about the running workload is collected to determine, using the training set, which type of application is being executed. *SmartGC* then changes the relevant parameters of the JVM according to what was predicted to be the best case for the running application, if a best parameter with a distinct confidence level is found. *SmartGC* relies on a GC system that can be instructed to change its parameters during the application runtime.

Instead of a strictly analytic model, *SmartGC* has two challenges for the training phase: (i) determine the *best* memory configuration parameters for a set of representative applications; (ii) use a set of system-related values to characterize the running application.

Regarding the first issue, resource allocation is in most cases a trade-off between two or more variables. Memory management in managed runtime is no exception. The system is designed to act on a given parameter. *SmartGC* explores the heap management policy described in Sect. 2, but structural components, such as the GC algorithm itself, can also be targeted.

The training phase can be periodically repeated when new hardware is acquired or a significant update to the runtime is made. The knowledge base built during this step can also be extended to incorporate more examples of *signatures*, so that online decisions can be made with a higher degree of confidence.

4.2 Execution Phase

After constructing a representative set of mnemonics, *SmartGC* is now ready to be plugged into a resource management middleware. It assumes that the managed runtime exposes mechanisms to dynamically reconfigure relevant systems, in our case, the garbage collector. In the online phase of the system, when a new runtime is started, *SmartGC* performs the following operations:

1. collects performance counters, asynchronously with GC and with the execution of regular threads. We use a separate VM internal thread to periodically monitor a set of performance counters;
2. each VM reports these values to the *oracle* (which can be located in another machine of the cluster);
3. the *oracle* aggregates these values between consecutive samples through one of the two metrics described in Sect. 3.2, and uses this information to find a signature, if that is already possible;
4. if a signature is found, this information is sent to the corresponding heap size manager which changes the GC-related parameter accordingly, and a GC execution is forced.

The *oracle* will make a prediction as good as the size and diversity of the applications used during the training phase. It may be the case that the *oracle* cannot predict a set of parameters with high confidence. In that case, *SmartGC* will use the default parameters and schedule a new training phase for the current application.

4.3 Online Phase

If the matrices confidence level, returned by the classifier for a given workload, are below a certain threshold (e.g., 50%), it means that the signature of that workload is not well known to the predictor. In these cases, we consider to incorporate the new workloads in our learning model, so that we are able to accurately predict the optimal matrix, should the same workload be exposed to the system at future times.

To this end, we: (i) run the (before-unseen) workload with each of the available matrices and assess which one leads to a better execution yield; (ii) include the values of program counters collected from the workload, along with the corresponding optimal matrix, in our knowledge base; and (iii) train a new learning model with an updated training-set containing the new workload footprint. This process is similar to our training phase, but it only considers executions of a single workload/application. Further, the system can be parametrized on whether to train a new workload or to use the matrix yielding the highest confidence (regardless of any defined thresholds).

5 Implementation

To achieve the best reconfiguration, *SmartGC* has to look for key runtime metrics, identify a signature and change the parameters to the appropriate values. This section describes the details of the runtime metrics collected and the machine learning algorithms used to classify a running application.

5.1 Classification Methods and Technologies

The classification process defines how the training set was built and the classification technology used. *SmartGC* classification system is organized around four classes, corresponding to the four matrices described previously. Having a known key set of workloads with the corresponding optimal matrix for each of them, we classify new workload runs, which were never seen before, and select the most appropriate matrices.

The training set is built during the so called training phase, as depicted in Fig. 4. For each execution of an application, we collect 12 performance counters (cf. Sect. 3.1) at every specified time interval (currently 100 ms). After a single run of an application we aggregate the values of the 12 counters that were observed over time thereby averaging the differences, or calculating the geometric mean, between consecutive observed values for every counter (i.e., thus having a single value for each performance counter and application execution). To stress application behavior in terms of counters variance, we run each of the 10 applications 50 times, resulting in 500 training instances. For this phase, we use the same matrix (M_0) for each execution, so that we have the same reference when obtaining counters at runtime. In the training set we also include for each instance of each application the corresponding optimal matrix (which was assessed beforehand in different trials).

We used a polynomial kernel as a parameter of a Support Vector Machine (SVM) classifier. Other kernels revealed to be less accurate. The *oracle* runs an SVM according to the Platt et al. algorithm [20]. To implement the *oracle*, we use the open source Weka framework.[3] Random Forests [8] consist of an ensemble learning, that is, a combination of various learning models to obtain better predictive performance. In this method, a large collection of tree predictors are built based on random subsets sampled from a set of training examples. The generalization capability depends on the strength of individual trees and the correlation between them. With bootstrap aggregating (also known as *bagging*) and a random selection of features to split each node it is possible to control the variance of the trees.

For pattern recognition, a classifier tries to estimate a function that, given a set with N-dimensional input data, predicts which of two possible classes form the output ($f : \mathbf{R}^N \rightarrow \pm 1$). It is also possible to classify examples in more than two classes (multiclass classification, as in our case with four classes), by using strategies like the "One-vs-Rest" that compare confidence values between pairs of binary-class classifications. The estimation of the SVM function, which corresponds to the construction of a SVM model, is based on a supplied set of training examples encompassing tuples with known correct values of input and corresponding output (i.e., supervised learning). After the model is constructed, the SVM is then able to assign new unseen examples to one of the possible multiple classes.

[3] http://www.cs.waikato.ac.nz/~ml/weka/index.html.

During the execution of an application, the oracle is queried every time interval with an aggregated value of each PC, at each time instance, since the application starts. This aggregated value captures the growth rate of a PC, and corresponds to the mean of the differences between every two consecutive time instances, as discussed in Sect. 3.2.

5.2 Collecting Information

The Jikes RVM can be built with support for reading performance counters.[4] It calls the PerfMon2 native Linux API.[5] In the codebase, performance events can be collected if the VM is executed in harness mode. This incurs in extra overhead so we have avoided some of the core code of the readings, and use a dedicated thread for this monitoring activity, so that the harnessing code path can be avoided. Periodically (default to $100ms$), the JVM collects and reports the performance counters to the oracle (in another physical machine of our testbed cluster) and waits for the classification response. If a new matrix is determined for the current signature, it will be used from then on. In the presence of jittering, we have a configurable threshold of n equal decisions (currently 5) necessary to make the change.

6 Evaluation

This section presents the evaluation of the three key aspects that have major impact in GC and application performance, as well as in the extent of the benefits of our approach. First it discusses the overheads of continuously monitoring the values of selected performance counters. We then focus on the classification process. The choices of a classification technology are presented followed by the confidence levels and stability of the classifier with unseen executions from applications used in the training set, and executions from applications never presented to the system. It concludes with memory savings obtained when compared with a widely used JVM, the OpenJDK.

Our execution environment is based on Jikes RVM 3.1.3, built in `production` mode (both in training and online phase), using the generational version of Immix [6]. We run on Intel Core i7-2600K processors (4 cores with hyper threading, 8M Cache and 3.40 GHz) with 12 GiBytes of RAM. The OS is Ubuntu 12.04.4 with kernel version 3.2.0-58.

We report on *SmartGC* overhead when compared with the JikeRVM 3.1.3 codebase (hereafter named as *baseline*). For each benchmark, both the *baseline* and *SmartGC* were executed 5 times to determine the run with the smallest execution. To measure this we used the record and replay infrastructure of Jikes RVM.[6] The advice files from the best run were then used in replay compilation. Each benchmark was executed 10 times, and the execution time was collected

[4] Using `--with-perfevents` parameter in `buildit` script.
[5] http://perfmon2.sourceforge.net/.
[6] http://jikesrvm.org/Experimental+Guidelines.

Table 1. Negligible impact on periodic collection of 12 performance counters. Mean execution time with confidence interval of 95%. Δ_{Mean} is $(SmartGC/Baseline) - 1$

	Base (ms)	CI (%)	*SmartGC* (ms)	CI (%)	Δ_{Mean}
xalan	9760.30	±0.04%	9863.7	±0.03%	1.06%
lusearch	1867.9	±0.04%	1887.6	±0.04%	1.05%
luindex	4304.8	±0.02%	4324.2	±0.01%	0.45%
fop	766.3	±0.12%	765.8	±0.04%	−0.07%
jython	19871.4	±0.08%	20030.9	±0.13%	0.80%
pmd	8459.7	±0.08%	8461.6	±0.05%	0.02%
bloat	28454.1	±0.24%	28397.0	±0.22%	−0.20%
antlr	2142.4	±0.09%	2263.1	±0.12%	5.63%
hsqldb	3646.1	±0.04%	3777.6	±0.03%	3.61%
eclipse	38110.8	±0.03%	38106.4	±0.02%	−0.01%

from the second iteration. Table 1 shows the mean of these executions along with the percentage value for a confidence interval of 95%.[7] The last column reports the percentage difference between the means of both systems. This value is very low for most cases, thus revealing that the proposed approach has a negligible impact in the execution time of these applications.

To select a good learning approach for our particular problem, we compared different widely-deployed Machine Learning algorithms thereby using the metrics: (i) accuracy; (ii) precision, proportion of instances that are truly of a class divided by the total instances classified as that class; (iii) recall, proportion of instances classified as a given class divided by the actual total in that class; (iv) F-measure, a weighted average of the precision and recall; and (v) ROC area, performance of the classifier where values approaching 1 mean optimal classifier and 0.5 being comparable to random guessing (Table 2). We used a data set with 10 applications and 500 examples of different executions, where 66% were used as the training set, and the remaining as our test set (while preserving instance order in the split). The parameters of all algorithms were tuned to get the best possible results. We decided to adopt SVM and Random Forest as our classification models to conduct all the experiences in this paper.

We have established that reading and reporting performance counters to the *oracle* (i.e., the online classifier) has a negligible impact on the performance of applications. We have also shown that, based on values collected in the training phase, the classifier which provides the best results is the one based on support vector machines. We now evaluate the *confidence* the *oracle* has that an unseen fragment of execution, either from a known or unknown application, represents one of the previously identified signatures, so that an appropriate heap resizing policy m will provide the best *execution yield* results.

[7] Because we have 10 final average samples we used the Students t-distribution to calculate the confidence interval [13].

Table 2. ML algorithms comparison

Algorithm	Accuracy	Precision	Recall	F-measure	ROC area
Bayes	20.58%	0.686	0.206	0.317	0.85
Logistic	52.35%	0.53	0.5224	0.475	0.806
Neuronal	99.41%	0.994	0.994	0.994	0.992
Rnd Forest	97.64%	0.978	0.976	0.976	0.990
J48 Tree	99.41%	0.994	0.994	0.994	0.992
SVM	99.41%	0.994	0.994	0.994	0.998

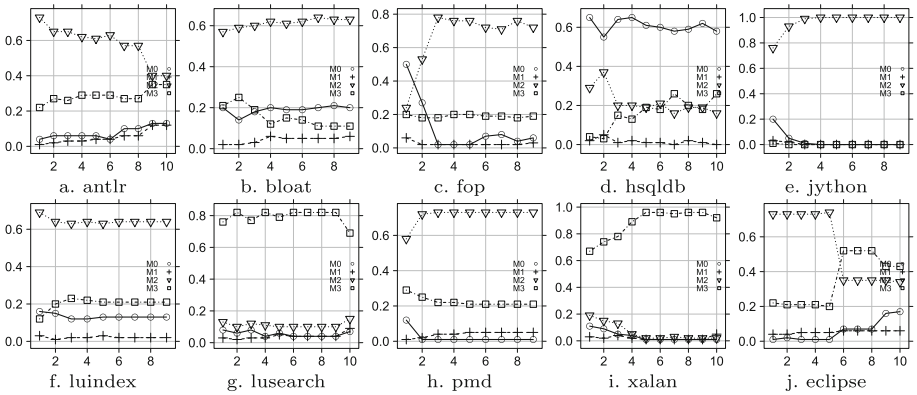


Fig. 5. Evolution of the classifiers confidence level of the unseen executions of 5 iterations of Dacapo applications

Figure 5 shows how the classification confidence of the *oracle* evolves while analyzing unseen performance counters values from the execution of 12 workloads. In each chart, the x-axis is time dependent, with a point for each classification. A classification is made based on the values of performance counters sent from a given JVM.

In the y-axis we plot the confidence value (between 0.0 and 1.0) the classifier reports for each possible matrix. This value is higher than 0.6 and there is always one classification that dominates the remaining ones. Furthermore, this high confidence levels are reached early in the execution, which is of critical importance for adaptiveness, fast change of parameters, and maximizing the overall execution yield.

For the majority of the workloads executions the *oracle* classifies them correctly, choosing the matrix that maximizes the execution yield. In cases where this is not so, like the case of `bloat`, the classifier classifies it with M_2 which is the one that represents the second best execution yield, very close to M_0, and far from M_3 which gives the worst yield for this workload.

Figure 6 also presents the confidence levels for each matrix but when running eight unseen applications, i.e., application which where not used during the

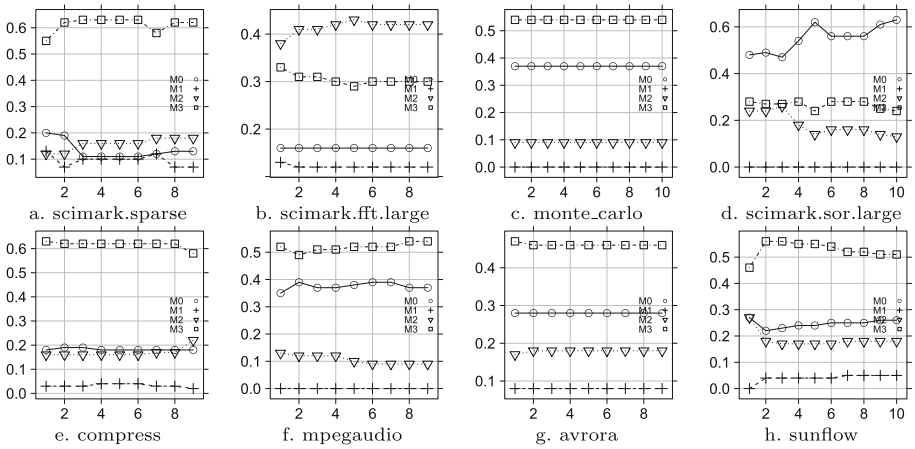


Fig. 6. Evolution of classifiers confidence level with unseen applications from SPECJVM2008 and Dacapo 2009

offline training phase. These applications are a mix of high performance computing workloads from SPECjvm2008[8] and two new benchmarks that where introduced in Dacapo 2009[9]. The results have a similar pattern to the ones discussed regarding Fig. 5, i.e., there is always a dominant matrix chosen by the classifier. Although in same cases the confidence is around 0.5, there is however a clear and stable winning choice during the execution of these unseen applications.

Finally we compare the memory saved when running reference applications with a mainstream OpenJDK 1.8.0, 64 bits, installed from the Ubuntu repository, and with *SmartGC*. OpenJDK was executed in `server` mode, the one that according to the documentation provides the best performance. OpenJDK also has a heap resizing policy, known as *Ergonomics*. This policy takes into account the application throughput, a maximum GC pause time and the minimum heap size.

Figure 7 shows the average memory used by OpenJDK and *SmartGC*. For these results, we assume that *SmartGC* chooses the correct matrix with the highest confidence. Each execution was iterated 3 times to promote the warm-up of the JVM. For 56% of the workloads there is a memory saving, which can reach 23%. This can even be underestimated because JikesRVM uses part of the heap for its own internal structures, given its meta-circular nature. This shows that our approach could be integrated with advantage in a widely deployed JVM, with limited and small changes to its codebase.

[8] https://www.spec.org/jvm2008.
[9] http://www.dacapobench.org.

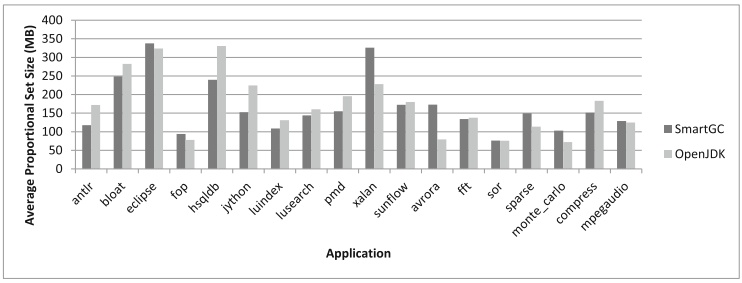


Fig. 7. Average memory used by OpenJDK and *SmartGC*. Lower is better.

7 Related Work

Memory is a relevant resource management target in cloud-like environments, typically using system-level virtual machines as in [2,27]. Recently, the need to involve the application runtime has gained more attention [7,10,22], including solutions by industry players such as VMWare's Elastic Memory for Java - EM4J.[10]

Researchers have analyzed garbage collection performance and found it to be application-dependent [11,30]. Based on these observations, several adaptation strategies have been proposed, ranging from parameters adjustments (e.g. the current size of the managed heap [15]) to changing the algorithm itself before the first execution or at runtime [30]. Other look for ways to minimize the pause time of stop-the-world garbage collectors [19]. This is an orthogonal effort that can be complemented with *SmartGC* operation because these systems are not meant to save memory that can be transferred to other tenants.

The adjustment of GC parameters to a given workload has been a topic of intense research [7,17,31], but most of them look for the parameters that give the highest throughput to a single application, regardless of memory usage. With *SmartGC* we explore a trade-off more relevant in consolidated environments - one that can reduce memory usage of applications without significantly hindering their usage of available CPU, i.e., without incurring in longer execution times.

Bobroff et al. [7] propose to investigate active memory sharing (AMS) in virtual environments. It distributes memory fairly to several running HLL-VMs. Chen et al. study the effect of consolidating HLL-VMs [11] but focus on either CPU or I/O bounded situations, leaving memory management unattained.

Only a few systems use machine learning techniques to learn the program behaviour and change the runtime algorithms or parameters. Andreasson et al. [4] used reinforcement learning techniques, to dynamically learn from GC collections. The system receives good or bad reinforcements by looking at the throughput after a GC collection. Singer et al. [28] determine, before execution, which is the best GC for a given program. It uses a J48 classification tree built from a

[10] https://www.vmware.com/support/pubs/vfabric-em4j.html.

long series of offline executions. Experimentations were made based just on full-heap collectors. The classification is done only offline, missing the opportunity for a finer grain adjustment to the different phases that each application might exhibit. Differently from *SmartGC*, these systems do not consider generational collectors with dynamic heap sizes or look at performance counters as a source of information.

Performance counters are typically used to detect bottlenecks and guide optimizations. Xu et al. [21] proposes to detect resource bottlenecks of multi-tier web servers, using low-level performance counters, such as, cache misses and instructions execution rate. Schneider and Gross [23] instrumented a JVM to feed the JIT compiler with performance counters information, so that more decisions can be further tailored to the underlying architecture and not only based on the program behaviour. Adl-Tabatabai et al. [1] uses performance counters to determine where to insert prefetch instructions in a JVM. This is done automatically within the same execution of the JVM. To the best of our knowledge, *SmartGC* is the first system that successfully guides the heap resizing policy of managed runtimes based on signatures learned from the observation of hardware performance counters.

8 Conclusion

We developed a low-overhead machine learning-based application classification that drives an adaptive GC control for cloud multi-tenancy. Our goal is to show that it is possible to choose at runtime, based on previous executions, the best GC parameter for a given application, obtaining reductions in the memory footprint of virtual machines with limited performance impacts, improving resource effectiveness and revenue.

The use of workload-aware policies to distribute resources among different tenants needs specific allocation mechanisms and strategies that can act upon the assets controlled by the managed runtimes, most notably memory. This paper describes the design rationale to build *SmartGC*, a system to guide the management of memory, to be used by PaaS service providers. The goal is to identify an application type, during its execution, and reconfigure relevant memory-related parameters based on the observation of current and performance counter values. We show that this can be done with low-overhead, applying parameters that can save physical memory, with minimum impact on the overall progress of applications. Future work should explore the use of *SmartGC* to target other tradeoffs in resource management, including the energy efficiency of different GC algorithms.

Acknowledgements. This work was supported by national funds through Fundação para a Ciência e a Tecnologia with reference PTDC/EEI-SCR/6945/2014, and by the ERDF through COMPETE 2020 Programme, within project POCI-01-0145-FEDER-016883. This work was partially supported by Instituto Superior de Engenharia de Lisboa and Instituto Politécnico de Lisboa. This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UID/CEC/50021/2013.

References

1. Adl-Tabatabai, A.R., Hudson, R.L., Serrano, M.J., Subramoney, S.: Prefetch injection based on hardware monitoring and object metadata. In: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation, PLDI 2004, pp. 267–276. ACM, New York (2004)
2. Ben-Yehuda, O.A., Posener, E., Ben-Yehuda, M., Schuster, A., Mu'alem, A.: Ginseng: market-driven memory allocation. In: Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE 2014, pp. 41–52. ACM, New York (2014)
3. Alpern, B., Augart, S., Blackburn, S.M., Butrico, M., Cocchi, A., Cheng, P., Dolby, J., Fink, S., Grove, D., Hind, M., McKinley, K.S., Mergen, M., Moss, J.E.B., Ngo, T., Sarkar, V.: The Jikes research virtual machine project: building an open-source research community. IBM Syst. J. **44**, 399–417 (2005)
4. Andreasson, E., Hoffmann, F., Lindholm, O.: To collect or not to collect? Machine learning for memory management. In: Proceedings of the 2nd Java Virtual Machine Research and Technology Symposium, pp. 27–39. USENIX Association, Berkeley (2002)
5. Blackburn, S.M., Garner, R., Hoffmann, C., Khang, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J.E.B., Moss, B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., Wiedermann, B.: The DaCapo benchmarks: Java benchmarking development and analysis. In: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA 2006, pp. 169–190. ACM, New York (2006)
6. Blackburn, S.M., McKinley, K.S.: Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. In: Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2008, pp. 22–32. ACM, New York (2008)
7. Bobroff, N., Westerink, P., Fong, L.: Active control of memory for Java virtual machines and applications. In: 11th International Conference on Autonomic Computing (ICAC 2014), pp. 97–103. USENIX Association, Philadelphia, June 2014
8. Breiman, L.: Random forests. Mach. Learn. **45**(1), 5–32 (2001)
9. Bu, Y., Borkar, V., Xu, G., Carey, M.J.: A bloat-aware design for big data applications. In: Proceedings of the 2013 International Symposium on Memory Management, ISMM 2013, pp. 119–130. ACM, New York (2013)
10. Cameron, C., Singer, J.: We are all economists now: economic utility for multiple heap sizing. In: Proceeding of Implementation, Compilation, Optimization of OO Languages, Programs and Systems (ICOOOLPS) (2014)
11. Chen, L., Serazzi, G., Ansaloni, D., Smirni, E., Binder, W.: What to expect when you are consolidating: effective prediction models of application performance on multicores. Cluster Comput. **17**(1), 19–37 (2014)
12. Francisco, A., Vaz, C., Monteiro, P., Melo-Cristino, J., Ramirez, M., Carrico, J.: Phyloviz: phylogenetic inference and data visualization for sequence based typing methods. BMC Bioinform. **13**(1), 87 (2012)
13. Georges, A., Buytaert, D., Eeckhout, L.: Statistically rigorous Java performance evaluation. In: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications, OOPSLA 2007, pp. 57–76. ACM, New York (2007)

14. Gront, D., Kolinski, A.: Utility library for structural bioinformatics. Bioinformatics **24**(4), 584–585 (2008)
15. Guan, X., Srisa-an, W., Jia, C.: Investigating the effects of using different nursery sizing policies on performance. In: Proceedings of the 2009 International Symposium on Memory Management, ISMM 2009, pp. 59–68. ACM, New York (2009). http://doi.acm.org/10.1145/1542431.1542441
16. Hauswirth, M., Sweeney, P.F., Diwan, A.: Temporal vertical profiling. Softw. Pract. Exp. **40**(8), 627–654 (2010)
17. Hertz, M., Kane, S., Keudel, E., Bai, T., Ding, C., Gu, X., Bard, J.E.: Waste not, want not: resource-based garbage collection in a shared environment. In: Proceedings of the International Symposium on Memory Management, ISMM 2011, pp. 65–76. ACM, New York (2011)
18. Janakiraman, G.J., Santos, J.R., Turner, Y.: JustRunit: experiment-based management of virtualized data centers. In: Voelker, G.M., Wolman, A. (eds.) 2009 USENIX Annual Technical Conference, San Diego, CA, USA. USENIX Association, 14–19 June 2009
19. Maas, M., Asanović, K., Harris, T., Kubiatowicz, J.: Taurus: a holistic language runtime system for coordinating distributed managed-language applications. In: Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2016, pp. 457–471. ACM, New York (2016)
20. Platt, J.: Fast training of support vector machines using sequential minimal optimization. In: Schoelkopf, B., Burges, C., Smola, A. (eds.) Advances in Kernel Methods - Support Vector Learning. MIT Press, Cambridge (1998)
21. Rao, J., Xu, C.Z.: Online capacity identification of multitier websites using hardware performance counters. IEEE Trans. Parallel Distrib. Syst. **22**(3), 426–438 (2011)
22. Salomie, T.I., Alonso, G., Roscoe, T., Elphinstone, K.: Application level ballooning for efficient server consolidation. In: Proceedings of the 8th ACM European Conference on Computer Systems, EuroSys 2013, pp. 337–350. ACM, New York (2013)
23. Schneider, F.T., Payer, M., Gross, T.R.: Online optimizations driven by hardware performance monitoring. In: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 373–382 (2007)
24. Sharifi, L., Rameshan, N., Freitag, F., Veiga, L.: Energy efficiency dilemma: P2P-cloud vs. datacenter. In: IEEE 6th International Conference on Cloud Computing Technology and Science, CloudCom 2014, Singapore, pp. 611–619. IEEE, 15–18 December 2014
25. Simão, J., Veiga, L.: QoE-JVM: an adaptive and resource-aware Java runtime for cloud computing. In: Meersman, R., et al. (eds.) OTM 2012. LNCS, vol. 7566, pp. 566–583. Springer, Heidelberg (2012). doi:10.1007/978-3-642-33615-7_8
26. Simão, J., Veiga, L.: Adaptability driven by quality of execution in high level virtual machines for shared cloud environments. Int. J. Comput. Syst. Sci. Eng. **29**(6), 413–426 (2013)
27. Simão, J., Veiga, L.: A taxonomy of adaptive resource management mechanisms in virtual machines: recent progress and challenges. In: Antonopoulos, N., Gillam, L. (eds.) Cloud Computing, pp. 59–98. Springer, Cham (2017). doi:10.1007/978-3-319-54645-2_3
28. Singer, J., Brown, G., Watson, I., Cavazos, J.: Intelligent selection of application-specific garbage collectors. In: Proceedings of the 6th International Symposium on Memory Management, ISMM 2007, pp. 91–102. ACM, New York (2007)

29. Singer, J., Jones, R.E., Brown, G., Luján, M.: The economics of garbage collection. In: Proceedings of the 2010 International Symposium on Memory Management, ISMM 2010, pp. 103–112. ACM, New York (2010)
30. Soman, S., Krintz, C.: Application-specific garbage collection. J. Syst. Softw. **80**, 1037–1056 (2007)
31. White, D.R., Singer, J., Aitken, J.M., Jones, R.E.: Control theory for principled heap sizing. In: Proceedings of the 2013 International Symposium on Memory Management, ISMM 2013, pp. 27–38. ACM, New York (2013)