Future Generation Computer Systems 100 (2019) 674–688

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

GC-Wise: A Self-adaptive approach for memory-performance efficiency in Java VMs

J. Simão [a,c,*], S. Esteves [a,b], André Pires [a,b], L. Veiga [a,b]

[a] *INESC-ID Lisboa, Portugal*
[b] *Universidade de Lisboa / Instituto Superior Técnico, Portugal*
[c] *Instituto Politécnico de Lisboa / Instituto Superior de Engenharia de Lisboa, Portugal*

HIGHLIGHTS

- Managed runtimes have a key role in different kinds of Cloud deployments and cooperative systems.
- Edge Clouds with large pools of devices should maximize resource usage.
- GC-Wise uses an online classification algorithm to select the best memory parameters.
- The approach can lead to memory savings with low-impact on the throughput.
- A server-class server and a single-board computer were used in the evaluation.

ARTICLE INFO

Article history:
Received 5 March 2018
Received in revised form 28 April 2019
Accepted 8 May 2019
Available online 17 May 2019

Keywords:
Memory management
Machine learning
Java virtual machine

ABSTRACT

High-level language runtimes are ubiquitous in every cloud deployment. From the geo-distributed heavy resources cloud provider to the new Fog and Edge deployment paradigms, all rely on these runtimes for portability, isolation and resource management. Across these clouds, 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. The challenges in this approach rely on the potential scale of systems and the need to make decisions in an application-driven way, because performance degradation can be severe, and therefore it should 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 *GC-Wise*, a system to determine, at run-time, the best values for critical heap management parameters of the OpenJDK JVM, aiming to maximize *memory-performance efficiency*. *GC-Wise* 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 an *oracle* matches the execution parameters of new workloads against those of already seen workloads, and enforces the best heap resizing policy. Distinctly from other works, the *oracle* can also decide upon unknown workloads. Using representative applications and different hardware setting (a resourceful server and a fog-like device), we show that our approach can lead to significant memory savings with low-impact on the throughput of applications. Furthermore, we show that we can predict with high accuracy the best heap resizing configuration in a relatively short period of time.

1. Introduction

Cloud Computing has been successful in providing large amounts of resources, as a utility, to deploy scalable and highly available applications [1]. Most cloud providers today draw their advantages from massive economies of scale resulting from heavy centralization around few large data centres [2–4]. This creates significant barriers-to-entry for providers, and raises lock-in and privacy issues for consumers, and environmental concerns regarding energy demands. Furthermore, these providers make use of metadata and behaviour information, putting citizen-generate data in the hands of a few major actors, which use this data to monetize their "free" infrastructures. Examples include personal devices (e.g., to monitor health-related conditions) or smart appliances, which are part of the increasing family of connected devices, building the Internet of Things (IoT). With large amounts of sensitive data being collected and in need to be processed,

* Corresponding author at: INESC-ID Lisboa, Portugal.
E-mail addresses: jsimao@cc.isel.ipl.pt (J. Simão), lveiga@gsd.inesc-id.pt (L. Veiga).

https://doi.org/10.1016/j.future.2019.05.027
0167-739X/

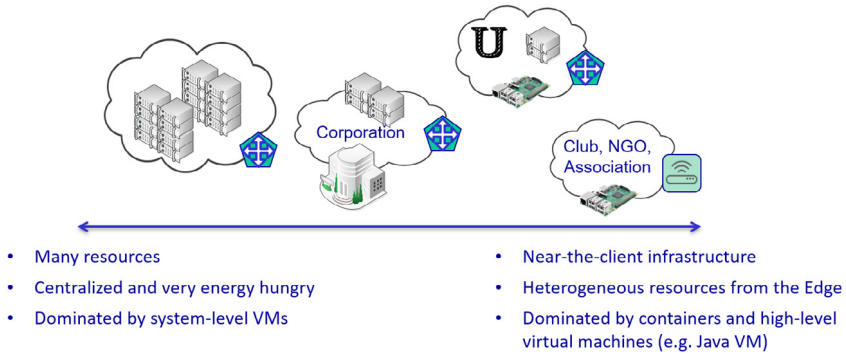


Fig. 1. From heavy-resourced datacenters to near the user fog computing.

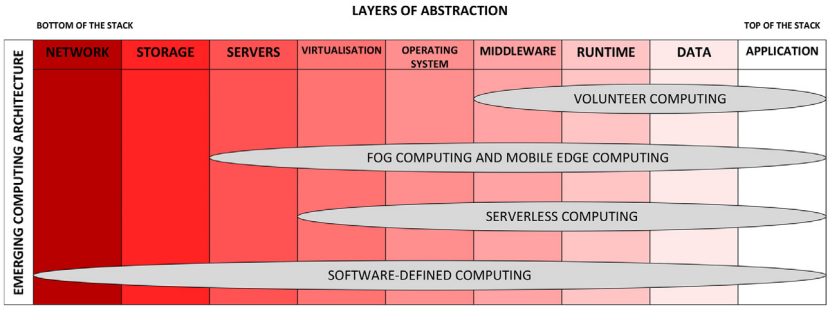


Fig. 2. Emerging computing architectures and the layers of abstraction that need to be adapted. *Source:* From [5].

user-centred devices rely on remote cloud storage and computational services. While this approach is reasonable for many applications, the ones dealing with sensitive data, or in need for low latency responses, would benefit from using computation services in control of the user, or group of users, and closer to the source of the data (the edge).

While IoT is expanding at huge rate, as stated by CISCO [6], it is filling the network's edge with a lot of data production. Trying to push all of this data into the Cloud, in order to process is costly (in terms of bandwidth) and it will saturate the Internet's backbone. The development of community networks and volunteer computing, together with today's low cost of compute and storage devices, is making the Internet's edge filled with a large amount of still under utilized resources [5], such as laptops, desktops, Raspberry Pi [7,8], computing boxes, routers and others. As depicted in Fig. 1, a movement, commercial and academic, is ongoing to leverage this "Edge Power" to provide services with smaller latencies and cheaper bandwidth to the end users. Pre-processing data at the edge would reduce the amount of data needed to be transmitted to the Cloud (to be stored in permanent storage and/or performing heavier computations), thus reducing the transmission cost. These new architectures trends push computation towards the edge, in a paradigm known as Fog computing [5,9–12], which captures this idea of a *cloud continuum* [13]. This computing paradigm expands the classic vision of a *cooperative system* as a set of distributed components, capable of communicate and cooperate to acquire, store, manage, query data and knowledge [14].

1.1. Virtualized and resource management in the edge

While these new environments are less suited for full system virtualization, as done in traditional datacenters, they are well capable of running applications supported by containers [15,16]. Doing so, they continue to benefit from the portability, safety and rich libraries of high-level languages like Python, Java and Scala [17,18], providing similar abstraction of the platform-as-a-service model. Regardless of the language of choice and the development paradigm (more object-oriented or more functional), managed runtimes that support the execution of these languages, have been increasingly used in cloud environments [19–22]. From heavier cloud providers (e.g., Heroku, AppFog and Google App Engine), to the near-the-user devices that make the Fog, all allow the deployments of workloads on high-level language runtimes (or high-level virtual machines), including multi-tenant environments. While the Cloud is getting more decentralized with several types of new deployments, all of them rely on runtimes to support the execution of applications, as depicted in Fig. 2.

One of the challenges in such cooperative systems is the configuration of large pools of devices and their managed runtimes, to maximize resource usage. Although these runtimes have a wide range of parameters that can the changed and adapted to different workloads [20,23,24], doing so is cumbersome and in many cases infeasible. To avoid this, it is necessary to use techniques known as *asymptotic* [25] where the application state is controlled externally and driven by its dynamic behaviour. In these resource constrained scenarios, it is paramount to adapt high-impact resource management policies to the running workloads. 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 an impact on the progress rate of the application. However, the best size of the heap remains application-specific. It adds to this point that, when resources are less abundant, taking memory from tenant A can have a different impact than when removing it from tenant B.

Concurrently with the consolidation requirements, we have considered the usage pattern of these platforms. Although clouds support the execution of general purpose applications, each user will most likely use the platform for a specific kind of tasks (e.g., data analytics [26] or epidemic surveillance [27,28]). This

opens the possibility to configure critical parameters of the memory system based on the application's *signature* (i.e., resource usage behaviour profile, or type profile) and the correlation with a set of configurations that are previously known as favouring consolidation.

1.2. Contributions and outline

This paper extends SmartGC [29] to also include support for another production-level runtime, OpenJDK, and new evaluation assessments of edge cloud environments running experiments with the Raspberry Pi hardware. We adapt the heap management policy of managed runtimes based on the identification of an application's execution profile – a *signature* – and the relation of this signature with parameters that are known to maximize the *memory-performance efficiency*, 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 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. *GC-Wise* explores different configurations regarding GC parameters, in a way that exchange throughput for memory, trying to find the most efficient use of memory for the current running application; (iii) deployment at less resourceful servers and hardware widely used in Fog-like deployments.

The next section makes an analysis to the state of the art. Section 3 shows the impact of heap size management policies in memory savings and execution performance, presenting memory-performance efficiency, a metric that relates these observations. Section 4 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 5 describes the design of *GC-Wise*, including the structure of a workload's *signature*, and the two distinct phases of operation (learning and classification). Section 6 makes an extensive evaluation of the major components of *GC-Wise*, including the comparison with a widely deployed JVM, the OpenJDK and Section 7 closes the paper discussing future work.

2. Related work

The platform-as-a-service model, where application owners do not have to manage low-level resources, continues to gain space in the cloud computing landscape [18,20,22,26,30,31], including in Fog deployments [17]. Scheduling of physical resources to application containers in general, is a widely research topic [1, 16,32–34], ranging from parallel processing architectures typically related to high-performance computing [35], to the increasingly form of application deployment and execution virtualization — containers [36].

Resource scheduling improvements in cloud deployment usually target either energy efficiency or higher application throughput. In large scale datacenters the energy utilization is a major concern [3,37–39] because of its impact on the power usage effectiveness [40], i.e., the ratio between total energy consumed by the facility versus the energy consumed by the hardware/software infrastructure. However, hardware used in Fog and Edge deployments are in many cases already resource constrained, and operate under low energy consumption [41]. This leaves to runtimes the configuration choices that maximize gains in application throughput, in an efficient manner, making the best usage of memory for as many applications as possible [41].

2.1. Memory scheduling in Java VMs

Overtime, memory is a resource harder to share than CPU cores. While context switching the CPU across processes creates overhead but enables full utilization of available cycles, memory behaves differently. When memory pages are committed to a process, even if not accessed over a period of time, it is harder and slower to exchange them across processes, as they cannot be shared in an efficient way. This makes optimizing memory usage a relevant resource management target in cloud-like environments, which typically use system-level virtual machines as in [42]. Thus, reducing heap size requirements enables usage of smaller and cheaper VM cloud instances and allows running more workloads in the same physical resources.

This is even more pressing in Edge Computing scenarios where physical nodes are resource constrained devices executing multiple workloads inside containers. This needs to be as memory-efficient as possible, since memory can become the resource bottleneck even during periods of less intensive processing. Memory-efficiency can further enable recent research with orthogonal goals (latency, energy, etc.). When the optimum status is reached, it will take less energy, memory and traffic, with less latency. So, a better performance from the underlying runtime may potentially contribute to edge computing, and improve overall efforts in areas such as low-latency workflow-oriented service functions or efficient traffic engineering [43–45].

Recently, the need to involve the application's runtime has gained more attention [19,46,47], including solutions by industry players such as VMWare's Elastic Memory for Java — EM4J.[1] Uninformed consolidation carries a negative impact on the application performance. With clever choices, providers can therefore consolidate more executions on the same hardware, in terms of memory, and save costs (and reduce prices or increase revenues) without significantly worsen application performance. They may take greater or lower benefit of the physical resources (namely CPU and memory) allocated to the VMs, in a way that the lack in one of them will prevent fruitful use of the other. Extreme situations include trashing (so little available memory, no CPU is used for useful work) and starvation (despite memory available, no CPU is available to process application data).

To determine memory allocation in Java VMs, several garbage collection (GC) algorithms have been developed during the last decades [48–50]. Concurrent approaches promises to improve GC in big data processing, but may require increased specific hardware support [18]. Parallel stop-the-world algorithms are still widely used, since they can efficiently collect moderate sized heaps fully within acceptable pauses, favour GC and application throughput, and do not require constant synchronization with the mutator, such as with concurrent collection, running on devices with any amount of resources and no special hardware support.

Garbage collection performance have been found to be application-dependent [51,52], resulting in several adaptation strategies, ranging from parameters adjustments (e.g., managed heap size [53]) to changing garbage collection algorithm before the first execution [54] or at runtime [51]. Other look for ways to minimize the pause time of stop-the-world garbage collectors [22]. This is an orthogonal effort that can be complemented with *GC-Wise* because these systems are not meant to save memory that can be transferred to other tenants. On the other hand, some system explore compiler-level transformations (e.g., [55]) which are not always possible to use given the massive deployment effort that would be needed.

The adjustment of GC parameters to a given workload has been a topic of intense research [30,46,53], but most of them

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

look for the parameters that give the highest throughput to a single application, regardless of memory usage. With *GC-Wise*, 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. [46] 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 [52] but focus on either CPU or I/O bounded situations, leaving memory management unattained. Rodrigues et al. [35] presents *Helpar* to act upon threshold-based master–slave parallel applications, where horizontal scaling (e.g., changes to the number of virtual machines) or vertical scaling (changes on CPU and memory made available to applications) is controlled by upper and lower bound values. This is done by instrumenting the application code (with necessary albeit reduced execution overhead), enforcing rules to act upon these thresholds. *GC-Wise* targets on mechanisms that are transparent to the application, demanding no code modifications, and requiring no hints/guidelines by the programmer.

2.2. Learning application's patterns

Only a few systems use machine learning techniques to learn the program behaviour and change the runtime algorithms or parameters. Andreasson et al. [56] 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. [57] determine, before execution, which is the best GC for a given program. It uses a J48 classification tree built from a 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 *Mnemonic*, 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 [58]. Xu et at. [59] propose 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 [60] 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. [61] 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, *GC-Wise* 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.

3. Memory-performance efficiency

GC-Wise goal is to reconfigure key parameters driving the behaviour of garbage collection in managed runtimes, in order to save memory while minimizing performance penalties, and so maximizing memory usage efficiency.

OpenJDK is a production-grade, open-source implementation of the Java Virtual Machine specification, with builds available from server-class hardware to ARM-based system, such as the Raspberry Pi. Across these versions, there is a memory management sub-system that can be configured to regulate how heap grows (the JVM asks more memory to the underlying operating system) or shrinks (memory is returned) which has impact on each instance execution but also on the co-located JVMs

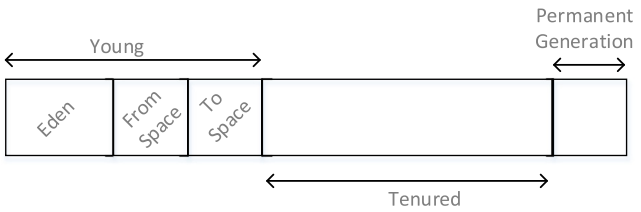


Fig. 3. Simplified heap layout of the Parallel Scavenge.

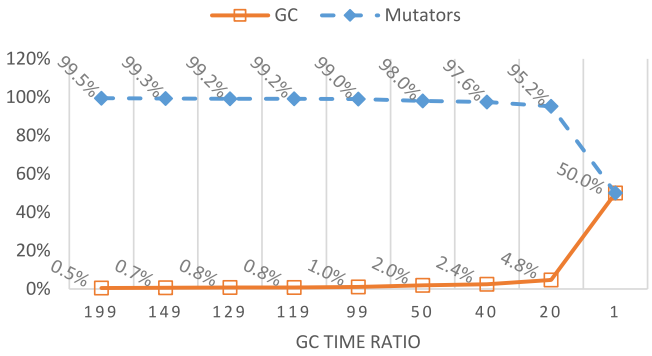


Fig. 4. Time of GC and mutator as a function of *GC time ratio*.

Table 1
9 configurations used to drive generations resize with memory-conserving policies.

Id	$genY_{inc}$, $genT_{inc}$	$factor_{dec}$	GCr
a	10%	8	10
b	10%	8	1
c	10%	10	10
d	10%	10	1
e	5%	8	10
f	5%	8	1
g	5%	10	10
h	5%	10	1

(e.g., within the same physical machine in a cloud data centre or an edge cloud node). In fact, other managed runtimes, such as the Common Language Runtime (CLR) have similar controls.

In this paper we target the Parallel Scavenge (PS), a stop-the-world, parallel and generational collector [62]. In the past, other collectors were proposed for the specific case of devices with less memory [63] but they focus on a single instance and are not widely available. When compared to the current default collector of the OpenJDK when running on a server-class machine (G1[64]), Parallel Scavenge favours throughput over response time. This makes it better suited for long running data-driven applications that process incoming data (e.g., stream processing for data analytics in IoT) as well as better fitting the devices that are increasingly more present in edge clouds. Given the concurrent nature of G1, it also needs a higher number of hardware cores to maximize the benefits when compared with PS collector. Furthermore, G1 is still only released as experimental for fog-class hardware such as the Raspberry Pi ARM system.[2]

The PS garbage collector organizes the memory heap into three major generations, (a) young; (b) tenured and (c) permanent, as depicted in Fig. 3. While the latter one is statically sized, the other two generations have their sizes dynamically adapted based on a strategy called *Ergonomics* [65]. This strategy has three general goals:

[2] https://docs.oracle.com/javase/8/embedded/jdk-arm-8u6/index.html, visited February 24, 2018.

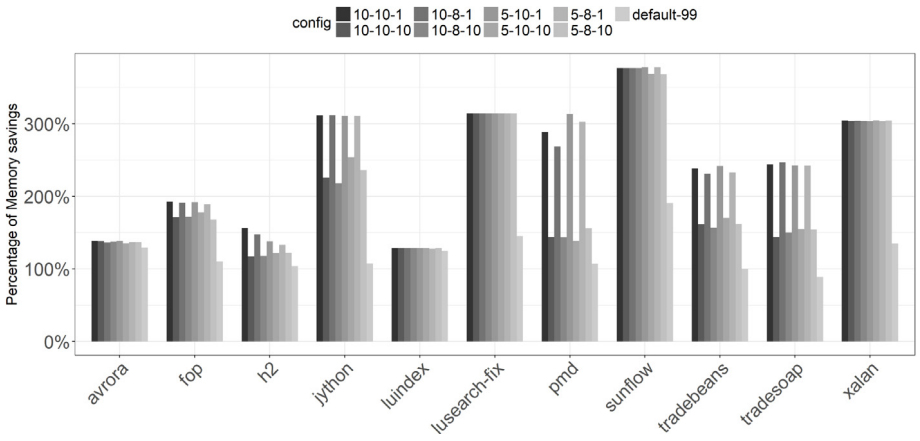


Fig. 5. Memory savings for each of the configurations.

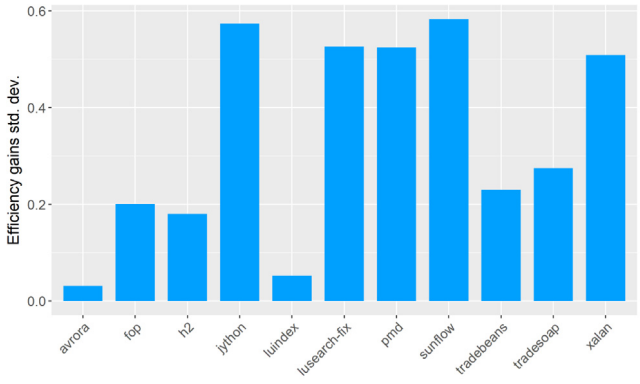


Fig. 6. Standard deviation of the efficiency gained in different applications using distinct heap aggressiveness options.

- do not exceed a given pause time;
- do not take over a certain fraction of overall time in GC operations;
- minimize the use of heap.

When the first goal is met (when undefined, this is presumed) the collector will check the second goal. This is so to keep GC operations under a certain threshold, which will release the CPU enough to guarantee a given level of throughput for the application. When the second goal is not being met, because GC operations are taking too long, the heap will be increased in size.

Several parameters participate in this process of decision, e.g., the ratio between young and old generations. However, the ones that have the highest impact on the throughput of the application are:

- The throughput goal itself, which is inversely proportional to the *GC time ratio* (*GCr*);
- *young* and *tenured* generations perceptual size increment ($genY_{inc}$, $genT_{inc}$) along with a decrement factor common to both generations ($factor_{dec}$).

The *GC time ratio* is a goal used by the JVM to influence the throughput of the application, and determines a target maximum time spent in GC operations, expressed as $\frac{1}{(1+GCr)}$. Fig. 4 shows how the target percentage of time for GC operations varies (for each GC time ratio) and, consequently, the variation of the percentage of time that should be spent in running application threads (i.e., mutators).

When the throughput is not being met, the size of both generations will grow. The new size is determined by the relative contribution of each generation to the total collection time. For example, if collecting the young generation took 15% of the time, and the increment $genY_{inc}$ is set to 25%, then the young generation will grow 3.75%. When the size of a generation can shrink, the new size is a fraction of the growing increment. For example, if the young generation can shrink the current size if reduced by $genY_{inc} \times \frac{1}{factor_{dec}}$ percent.

GC-Wise compares the automatic memory configuration (c_0) of a JVM where a good execution time for the application ($time(c_0)$) is achieved at the expense of more memory ($mem(c_0)$), with other configurations ($c_{\{i..n\}}$) where memory is saved, possibly at the cost of more execution time for the garbage collector. For two configurations, c_a and c_b, the memory usage efficiency is then defined as Eq. (1):

$$\frac{mem(c_a)/mem(c_b)}{time(c_b)/time(c_a)} \tag{1}$$

Not only implementing a correct collector is difficult and error-prone [66], choosing the best values for each of the available parameters is also challenging. Previous work typically only controls the overall heap size as seen from inside the virtual machine [46,47], not the specific sizes of each of the generations. The throughput of the parallel collector is deeply connected with these dimensions and our system takes into consideration the parameters that regulates them.

Thus, *GC-Wise* explores the right-side of Fig. 4, regarding the parameters presented: *GCr*, $genY_{inc}$, $genT_{inc}$ and $factor_{dec}$. The base configuration uses a very high *GCr* (i.e., 199), meaning that the JVM should aim for the highest throughput, leaving the remaining values with the "factory" default base values, 20%, 20% and 4, respectively. The first configuration to compare with this one is called the "default", and only changes one parameter, the *GCr*, to 99, indicating the JVM should keep GC operations below 1%. The remaining 8 configurations to guide the generations resizing process are described in Table 1.

The goal is to correlate each application from a *reference set* to the configuration that maximizes the memory usage efficiency. These applications are from the 2009 version of Dacapo benchmark [67].[3] Each of these applications explores a different issue of the JIT, GC and micro-architecture, as extensively demonstrated in [67]. We rely on these widely used benchmarks to represent the behaviour of full applications (or specific phases during applications' execution), and thus, that the execution patterns they

[3] Maintenance release of Jan. 12, 2018.

exhibit regarding the use of memory are representative of other Java applications.

Fig. 5 presents the memory saved when running 9 different configurations in representative workloads, with different memory patterns, and setting the maximum heap size to 400 MB. We can see that all configurations save memory (except tradesoap with "default-99"). However, when comparing the "default-99" configuration with the remaining configurations, the savings vary not only among workloads, but more relevant, under the same workload. This shows there is significant room to modify (and reduce) the memory usage of applications running on the JVM. The challenge is to do so while hurting performance in a less significant reduced way.

Fig. 6 presents the standard deviation of the memory-performance efficiency obtained for each reference application. The ratio between relative memory savings and relative performance degradation leads to a different memory-performance efficiency variations for each pair of reference application and configuration. This memory-performance efficiency is an opportunity, for the provider, to use tailored memory-saving parameters without imposing a significantly perceivable, or at all, performance (i.e, application progress or throughput) penalty in each workload. Larger numbers reveal that the resources saved are several times higher than any imposed relative penalty. This essentially enables "releasing" 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 off more efficiently.

After establishing the relationship between each type of workload and the best configuration suiting it, we now need to find a set of execution's characteristics, to be used during run-time, to identify the profile of running applications.

4. 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 several metrics that have the potential to characterize a workload, including runtime metrics resource indicators used by *GC-Wise* (Section 4.1.1), and how to characterize and cluster different workloads according to their patterns (Section 4.2).

4.1. Profiling metrics

This section discusses different sensors, at system and application-level, which can be used to learn a profile about a running application.

4.1.1. 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. This type of information is also available in different architectures, including ARM-based devices used to build Fog and Edge solutions.[4] 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).

GC-Wise 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).

4.1.2. Static code analysis

Other metrics can be explored in the future, both static (i.e., structure of the program) and dynamic (i.e., metrics collected at runtime). One of the most cited set of static metrics regarding object-oriented programs are the ones proposed by Chidamber et al. [68]. For example, *lack of cohesion of methods* (LCOM) indicates whether a class represents a single abstraction or multiple abstractions. This is obtained by static analysis of the code, meaning that the application needs not to be executed. Examples include the number of methods defined in a class and the number of immediate sub-classes of a class. However this metrics do not have a direct impact on the working set of the application or its dynamic behaviour. The application working set is determined by how objects are created and left unreferenced.

4.1.3. Garbage collection

Regarding dynamic metrics, three relevant characteristics related to the garbage collection process have been identified [57]:

(i) number of allocated arrays during program execution; (ii) mean size of objects in bytes; (iii) number of allocated bytes during program execution. Some modification is necessary to Jikes RVM code to get (i) and (ii). The metrics used by memory usage analysers are also interesting to determine a profile of the running application. Elephant Tracks [69] produces an in-order trace of (i) object allocation and death; (ii) method entry and exit; (iii) pointer update events. [(i)] It uses a native Java agent that loads along with the JVM. It depends on the availability of the JVMTI and does some bytecode rewriting. Metrics (i) and (ii) should be helpful to characterize the application in a way dependent on the behaviour of the GC. Metric (iii) is used in the Dacapo paper to describe the dynamic characteristics of each benchmark. However, these approaches do not target production running VMs because they impose a significant overhead given their intrusion on the regular execution of a program.

[4] https://community.arm.com/iot/embedded/b/embedded-blog/posts/using-the-arm-performance-monitor-unit-pmu-linux-driver, visited Feb. 24, 2018.

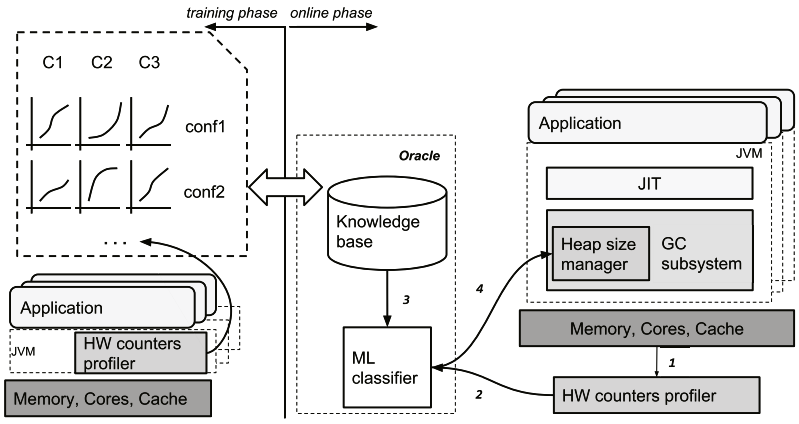


Fig. 7. System design.

4.2. 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 ($pc_1 \ldots pc_N$), between time t_i (initial time) and t_f (final time), as depicted in Eq. (2).

$$S_w(t_i, t_f) = Aggr(pc_1, t_i, t_f), \ldots, Aggr(pc_N, t_i, t_f) \quad (2)$$

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. (3).

$$Diffs(pc_k, t_i, t_f) = \frac{1}{m} \sum_{x=t_i+\Delta t}^{t_f} pc_k(x) - p_k(x-1) \text{ where } k \in [1, N] \quad (3)$$

The other option is a geometric mean, as described in Eq. (4).

$$GMean(pc_k, t_i, t_f) = \left(\prod_{x=t_i+1}^{t_f} pc_k(x) \right)^{1/(tf-ti)} \text{ where } k \in [1, N] \quad (4)$$

After building a representative signature, a *mnemonic* can then be set, associating a signature to the correct best set of parameters (i.e., a configuration), $\{P_1, P_2, \ldots, P_R\}$, for the application. Eq. (5) represents a mnemonic for application W, where signature S_w is associated to the best set of parameters $(P_1, P_2, \ldots, P_R)$.

$$M_w = (S_w \rightarrow \{P_1, P_2, \ldots, P_R\}) \quad (5)$$

An example of parameters are the matrices presented in Section 3. 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 *GC-Wise* , including its two main phases — training and runtime operation.

5. System design

Analytical modelling is a common approach to inform resource-allocation systems [70,71]. 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 centres [70]. *GC-Wise* uses this strategy, and could easily be plugged into such systems. Fig. 7 shows the overall system view. *GC-Wise* 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.

In order to address the challenges of having a large number of elements in the network to configure, we based our approach on asymptotic configuration techniques [9,25], using an *oracle* which asserts the desired configuration of the memory management system.

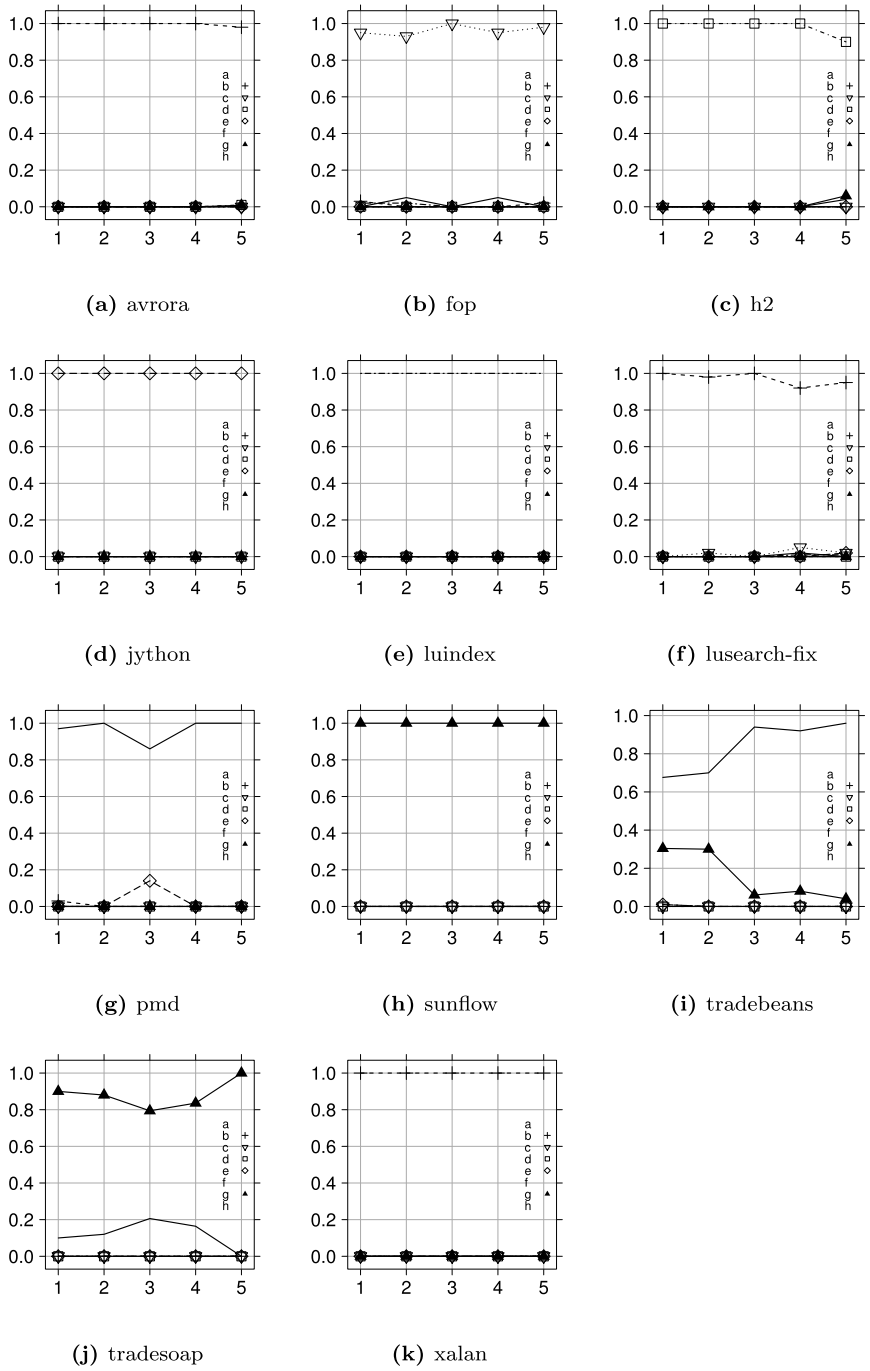
5.1. Training phase

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's *signatures*, where each *signature* is associated with the heap management configurations that maximizes the memory-performance efficiency. 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. *GC-Wise* 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. *GC-Wise* relies on a GC system that can be instructed to change its parameters during the application runtime.

Instead of a strictly analytic model, *GC-Wise* 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. *GC-Wise* explores the heap management policy described in Section 3, 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.



(a) avrora **(b)** fop **(c)** h2

(d) jython **(e)** luindex **(f)** lusearch-fix

(g) pmd **(h)** sunflow **(i)** tradebeans

(j) tradesoap **(k)** xalan

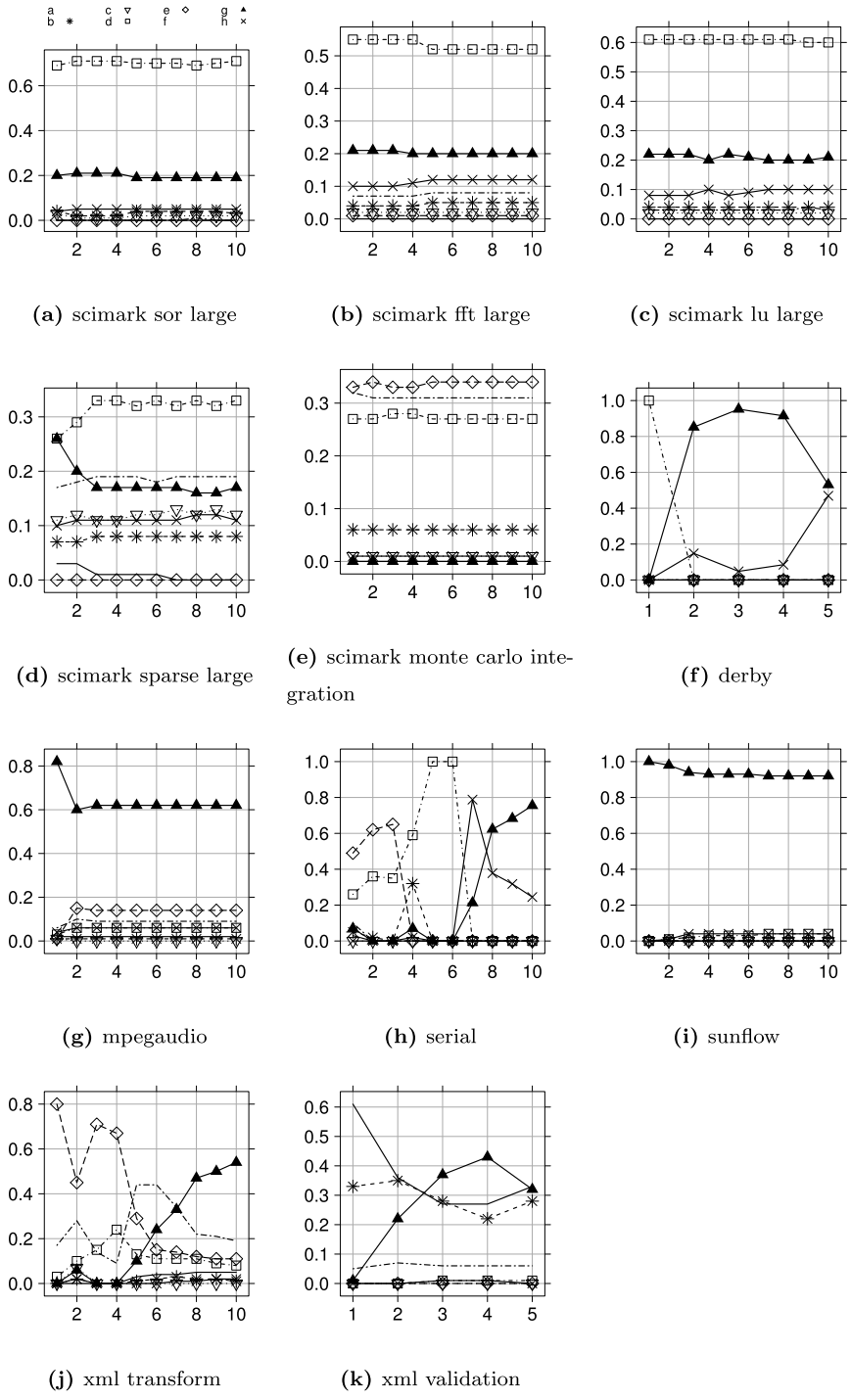
Fig. 8. Confidence levels for the Dacapo 9 benchmarks.

5.2. Execution phase

After constructing a representative set of mnemonics, *GC-Wise* 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, *GC-Wise* performs the following operations, as numbered in Fig. 7:

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 Section 4.2, 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, *GC-Wise* will use the default



(a) scimark sor large **(b)** scimark fft large **(c)** scimark lu large

(d) scimark sparse large **(e)** scimark monte carlo integration **(f)** derby

(g) mpegaudio **(h)** serial **(i)** sunflow

(j) xml transform **(k)** xml validation

Fig. 9. Confidence levels for the SPECJVM 2008 benchmarks.

parameters and schedule a new training phase for the current application.

5.3. 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 configuration, 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 memory-performance efficiency; (ii) include the values of program counters collected from the workload, along with the corresponding optimal configuration, 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 configuration yielding the highest confidence (regardless of any defined thresholds).

5.4. Classification

To achieve the best reconfiguration, *GC-Wise* 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.

The classification process defines how the training set was built and the classification technology used. *GC-Wise* classification system is organized around different classes, corresponding to the configuration of parameters described in Section 3. Having a known key set of workloads with the corresponding optimal parameters for each of them, we classify new workloads runs, which were never seen before, and select the most appropriate configuration.

5.4.1. Building the training set

The training set is built during the so called training phase, as depicted in Fig. 7. For each execution of an application, we collect 12 performance counters (cf. Section 4.1.1) 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 behaviour 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 JVM's default configuration 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 parameters (which was assessed beforehand in different trials).

5.4.2. Online classifier

We used a polynomial kernel as a parameter of a Support Vector Machine (SVM) classifier [72], and a classification based on Random Forests [73]. The *oracle* runs an SVM according to the John Platt et al. algorithm. To implement the *oracle*, we use the open source Weka framework.[5]

Random Forests [73] 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.

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 start. 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 Section 4.2.

6. Evaluation

This section presents the evaluation of two key aspects of *GC-Wise* : the machine learning classification process and the benefits of the choices made for different types of applications. 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.

Our execution environment is based on two distinct hardware profiles. The following is the specific hardware used for each of these profiles:

- Profile 1 – Intel Core i7-6700 processors (4 cores with hyper threading, 8M Cache and 3.40 GHz) with 16 GB of RAM. The OS is Ubuntu 16.04.3 with kernel version 4.4.0–112.
- Profile 2 – A Raspberry Pi 3 (ARMv7 Processor rev 4), 4 cores each with 1.2 GHz and 1 GB of RAM. The OS is Raspbian GNU/Linux 8.0 (jessie).

Regarding the JVM, both systems use a OpenJDK-based distribution. The Intel-based system uses a tailored build of the OpenJDK 9, 64-bit. The ARM-based system runs with the default JVM for the Raspbian system, the OpenJDK Zero VM.

6.1. Unseen executions

Fig. 8 shows how the classification confidence of the *oracle* evolves while analysing unseen performance counters values from the execution of 11 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 collected 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 always very high, in most cases above 80% confidence about the best configuration. Furthermore, there is always one classification that dominates the remaining ones. Also, the high confidence levels are reached early in the execution, which is of critical importance for adaptiveness, fast change of parameters. For the majority of the workloads' executions the *oracle* classifies them correctly, choosing the class corresponding to the configuration that maximizes the memory-performance efficiency.

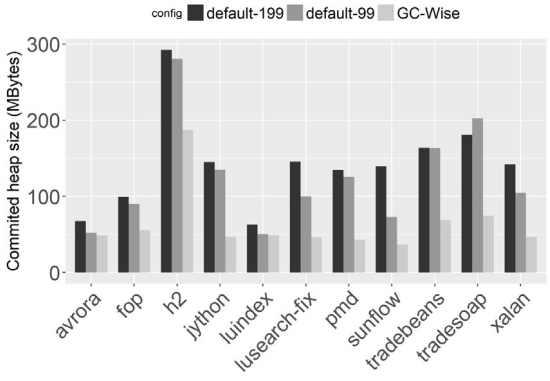
6.2. Unseen applications

Fig. 9 also presents the confidence levels for each configuration but when running 11 unseen applications, i.e., applications which where not used during the offline training phase. These applications are a mix of high performance computing workloads, a database and XML schema transformations, all from SPECjvm2008.[6]

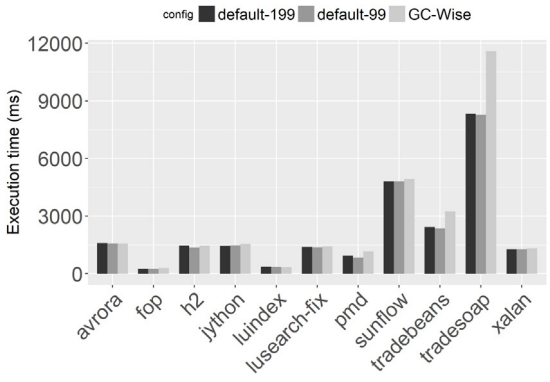
The results show that for 9 of the 11 applications there is always a dominant configuration chosen by the classifier. The XML transformation and validation benchmarks are the exceptions. This indicates that the classification model is not trained enough with this class of applications — XML transformers. In fact, the base training set includes only 1 application of this type, `fop`, and it is the one which produces the smaller number of samples because of its execution time.

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

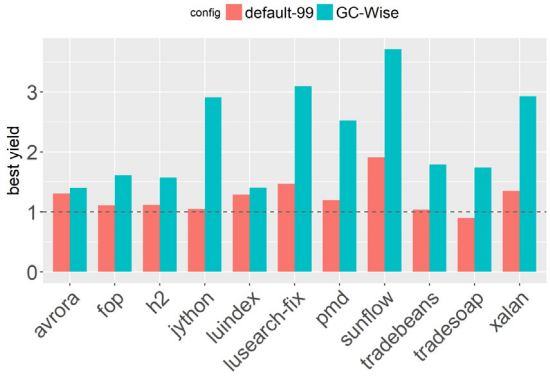
[6] https://www.spec.org/jvm2008/, vistied Feb. 24, 2018.



(a) Commited heap size, lower is better



(b) Execution time



(c) Memory-performance efficiency , higher is better

Fig. 10. Dacapo applications running on server-class hardware.

Although in same cases the confidence is below 0.5, there is however a clear and stable winning choice during the execution of these unseen applications. Another point taken from the observations is that the `scimark` family of benchmarks are all classified as configuration **g**. This family of benchmarks deals with matrix operations, except for the `monte carlo integration` benchmarks.[7] Finally, note that SPECjvm also uses part of the `sunflow` code as a benchmark (sub-figure (i) of Fig. 9) and the classifier determines the same class of Fig. 8, with very high confidence.

[7] https://math.nist.gov/scimark2/about.html, visited Feb. 24, 2018.

6.3. Measuring the memory-performance efficiency

The first set of results were obtained with the hardware from Profile 1. Fig. 10.a and b present the memory savings and execution time, respectively, of Dacapo benchmarks when running on the class identified by the classification system. The JVM was parametrized with a 400 MB maximum heap across all configurations. The series `default-199` and `default-99` differ in the GC time ratio (i.e. *GCr*, Section 3), using the values 199 and 99. In the latter case, the JVM will try to minimize the GC operation time to less than 1% of the overall time, freely expanding the heap as necessary, eventually until the maximum value defined. In both series, the generations resizing parameters are the JVM's default ones. Fig. 10.c depicts the corresponding memory-performance efficiency comparing the two competitive configuration (default-99 and `GC-Wise`) with the base one (`default-199`). *GC-Wise* can improve memory savings across all applications, in most case having at least 50% more memory-performance efficiency than the `default-199` base configuration, while in 5 of them it more than doubles.

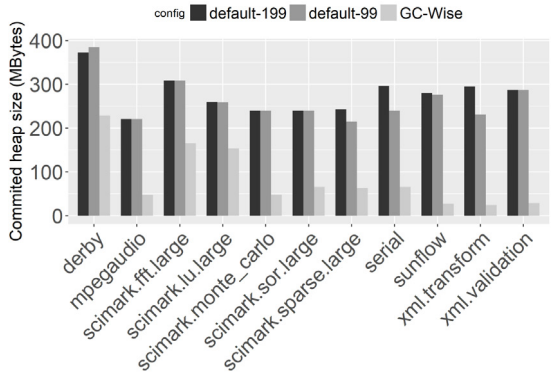
Fig. 11 does the same study but using the set of application not used to train the model. In this set of applications the saved heap is larger because the default parameters are too *generous* regarding the memory committed to each process, unnecessarily using resources that would either have to be overcommitted or simply refused to new tenants sharing the same hardware.

To conclude the evaluation we run a sub-set of the application in an hardware of Profile 2. The OpenJDK Zero VM was set with a maximum heap size of 200 MB, and an initial size of 25 MB. Before each run, the system had approximately 320 MB of RAM available for user space processes. Fig. 12 shows the results of using the best values identified by the classifier for each of the tested application. In this case, *GC-Wise* operates as well as the default configuration for 4 applications, but more than doubles the memory-performance efficiency for the remaining four. In a device with so many memory limitations, having the possibility to choose a more memory-efficient solution, even for a sub-set of applications, without harming performance or the other applications, is extremely important. This is a key enabling result for Fog and Edge computing with Java.

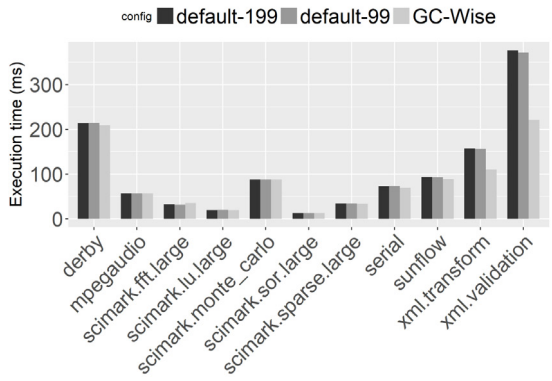
7. Conclusions

In recent years we have seen a growing interest on the process of migrating from heavy-resourced datacenters to the edge of the network. This vision of edge clouds, when fulfilled, is a complex *cooperative system* where parties communicate and cooperate in order to process and manage both data and knowledge. While these new deployments will continue to enjoy the portability, safety, and high-level programming simplicity of managed runtimes, they face new resource scheduling challenges because of the number of devices involved and their less resourceful hardware, particularly memory [17,41].

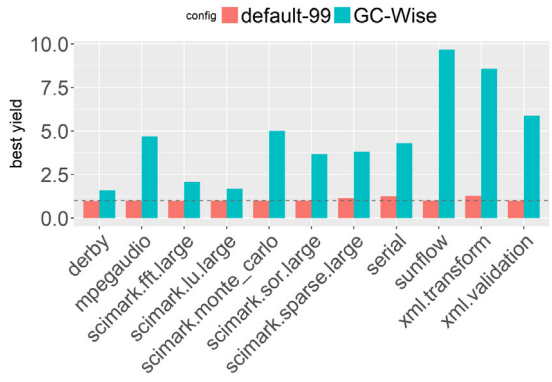
We developed a low-overhead machine learning-based application classification that drives an adaptive memory management control, useful across different cloud deployment. Our goal is to show that it is possible to choose at run-time, based on previous executions, the best GC parameters for a given application, 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 managed runtimes, most notably memory. This paper describes the design rationale to build *GC-Wise*, a system to guide the management of memory in systems where certain



(a) Commited heap size, lower is better

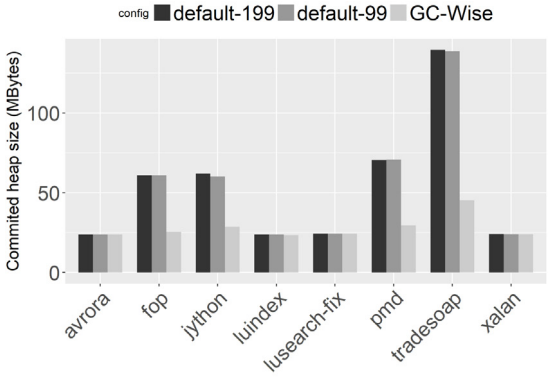


(b) Execution time

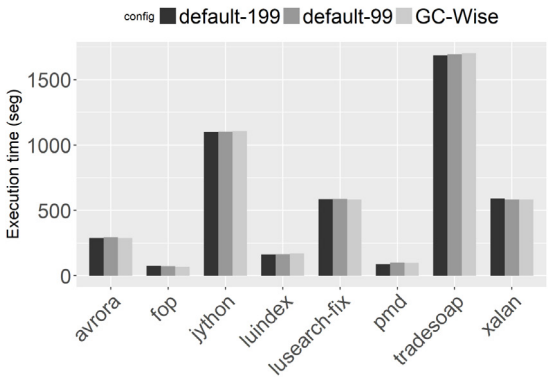


(c) Memory-performance efficiency , higher is better

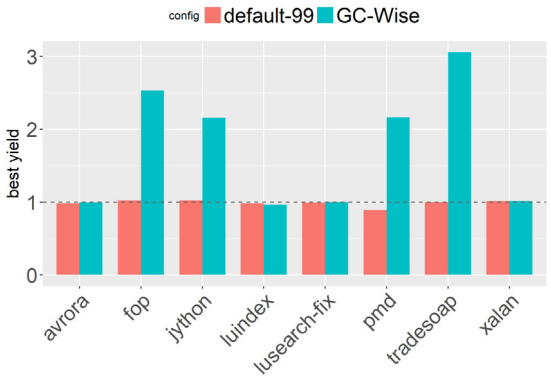
Fig. 11. SPECjvm application running on server-class hardware.



(a) Commited heap size, lower is better



(b) Execution time



(c) Dacapo memory saving

Fig. 12. Dacapo applications running on Raspberry Pi class hardware.

classes of application are frequently executed given the chance to learn the best parameters and use that knowledge in the future. *GC-Wise* identifies an application type and reconfigures relevant memory-related parameters based on the observation of 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. The techniques presented in this paper can be applied to other adaptation points, if the manage runtime exports them to be used by an autonomous adaptation middleware. Examples include the garbage collection algorithm itself. Furthermore, future work should explore the use of *GC-Wise* to target other trade-offs 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, Portugal with reference PTDC/EEI-SCR/6945/2014 and PTDC/EEI-COM/30644/2017, and by the ERDF through COMPETE 2020 Programme, within project POCI-01-0145-FEDER-016883. This work was supported by national funds through Fundação para a Ciência e a Tecnologia,

Portugal with reference UID/CEC/50021/2019. This work was partially supported by Instituto Superior de Engenharia de Lisboa and Instituto Politécnico de Lisboa.

Declaration of competing interest

None

References

[1] R. Buyya, C.S. Yeo, S. Venugopal, J. Broberg, I. Brandic, Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility, Future Gener. Comput. Syst. 25 (6) (2009) 599–616.

[2] R. Buyya, S.K. Garg, R.N. Calheiros, SLA-Oriented resource provisioning for cloud computing: Challenges, architecture, and solutions, in: Proceedings of the 2011 International Conference on Cloud and Service Computing, in: CSC '11, IEEE Computer Society, Washington, DC, USA, 2011, pp. 1–10.

[3] A. Khosravi, S.K. Garg, R. Buyya, Energy and Carbon-efficient placement of virtual machines in distributed cloud data centers, in: F. Wolf, B. Mohr, D. an Mey (Eds.), Euro-Par, in: Lecture Notes in Computer Science, vol. 8097, Springer, 2013, pp. 317–328.

[4] S. Singh, I. Chana, A survey on resource scheduling in cloud computing: Issues and challenges, J. Grid Comput. 14 (2) (2016) 217–264, http://dx.doi.org/10.1007/s10723-015-9359-2.

[5] B. Varghese, R. Buyya, Next generation cloud computing: New trends and research directions, Future Gener. Comput. Syst. 79 (2018) 849–861, http://dx.doi.org/10.1016/j.future.2017.09.020, http://www.sciencedirect.com/science/article/pii/S0167739X17302224.

[6] Cisco Systems, Fog Computing and the Internet of Things: Extend the Cloud to Where the Things Are, White Paper, 2016, 6, http://dx.doi.org/10.1109/HotWeb.2015.22, http://www.cisco.com/c/dam/en_us/solutions/trends/iot/docs/computing-overview.pdf.

[7] C. Pahl, S. Helmer, L. Miori, J. Sanin, B. Lee, A container-based edge cloud paas architecture based on raspberry pi clusters, in: Proceedings - 2016 4th International Conference on Future Internet of Things and Cloud Workshops, W-FiCloud 2016, 2016, pp. 117–124, http://dx.doi.org/10.1109/W-FiCloud.2016.36.

[8] P. Bellavista, A. Zanni, Feasibility of fog computing deployment based on docker containerization over raspberrypi, in: Proceedings of the 18th International Conference on Distributed Computing and Networking - ICDCN '17, 2017, pp. 1–10, http://dx.doi.org/10.1145/3007748.3007777, http://dl.acm.org/citation.cfm?doid=3007748.3007777.

[9] L.M. Vaquero, L. Rodero-Merino, Finding your way in the fog: Towards a comprehensive definition of fog computing, ACM SIGCOMM Comput. Commun. Rev. 44 (5) (2014) 27–32, http://dx.doi.org/10.1145/2677046.2677052, http://dl.acm.org/citation.cfm?id=2677046.2677052.

[10] P. Varshney, Y. Simmhan, Demystifying fog computing: Characterizing architectures, applications and abstractions, in: 2017 IEEE 1st International Conference on Fog and Edge Computing (ICFEC), 100, 2017, http://dx.doi.org/10.1109/ICFEC.2017.20, http://arxiv.org/abs/1702.06331.

[11] B. Varghese, N. Wang, S. Barbhuiya, P. Kilpatrick, D.S. Nikolopoulos, Challenges and opportunities in edge computing, in: Proceedings - 2016 IEEE International Conference on Smart Cloud, SmartCloud 2016, 2016, pp. 20–26, http://dx.doi.org/10.1109/SmartCloud.2016.18.

[12] R. Buyya, S.N. Srirama, G. Casale, R.N. Calheiros, Y. Simmhan, B. Varghese, E. Gelenbe, B. Javadi, L.M. Vaquero, M.A.S. Netto, A.N. Toosi, M.A. Rodriguez, I.M. Llorente, S.D.C. di Vimercati, P. Samarati, D.S. Milojicic, C.A. Varela, R. Bahsoon, M.D. de Assunção, O.F. Rana, W. Zhou, H. Jin, W. Gentzsch, A.Y. Zomaya, H. Shen, A manifesto for future generation cloud computing: Research directions for the next decade, CoRR abs/1711.09123 (2017) http://arxiv.org/abs/1711.09123, arXiv:1711.09123.

[13] L. Bittencourt, R. Immich, R. Sakellariou, N. Fonseca, E. Madeira, M. Curado, L. Villas, L. DaSilva, C. Lee, O. Rana, The internet of things, fog and cloud continuum: Integration and challenges, Internet Things 3–4 (2018) 134–155, http://dx.doi.org/10.1016/j.iot.2018.09.005, http://www.sciencedirect.com/science/article/pii/S2542660518300635.

[14] E. Bertino, B. Catania, V. Gervasi, A. Raffaet, A logical approach to cooperative information systems, J. Logic Programm. 43 (1) (2000) 15–48, http://dx.doi.org/10.1016/S0743-1066(99)00024-2, http://www.sciencedirect.com/science/article/pii/S0743106699000242.

[15] R. Morabito, J. Kjllman, M. Komu, Hypervisors vs. lightweight virtualization: A performance comparison, in: Proceedings of the 2015 IEEE International Conference on Cloud Engineering, in: IC2E '15, IEEE Computer Society, Washington, DC, USA, 2015, pp. 386–393, http://dx.doi.org/10.1109/IC2E.2015.74.

[16] M.A. Rodriguez, R. Buyya, Container-based cluster orchestration systems: A taxonomy and future directions, Softw: Pract. Exp. 0 (0) arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2660, http://dx.doi.org/10.1002/spe.2660, https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2660.

[17] A. Brogi, S. Forti, Qos-aware deployment of iot applications through the fog, IEEE Internet Things J. 4 (5) (2017) 1185–1192, http://dx.doi.org/10.1109/JIOT.2017.2701408.

[18] M. Maas, K. Asanović, J. Kubiatowicz, Return of the runtimes: Rethinking the language runtime system for the cloud 3.0 era, in: Proceedings of the 16th Workshop on Hot Topics in Operating Systems, in: HotOS '17, ACM, New York, NY, USA, 2017, pp. 138–143, http://dx.doi.org/10.1145/3102980.3103003, http://doi.acm.org/10.1145/3102980.3103003.

[19] T.-I. Salomie, G. Alonso, T. Roscoe, K. Elphinstone, Application level ballooning for efficient server consolidation, in: Proceedings of the 8th ACM European Conference on Computer Systems, in: EuroSys '13, ACM, New York, NY, USA, 2013, pp. 337–350.

[20] R. Bruno, P. Ferreira, A study on garbage collection algorithms for big data environments, ACM Comput. Surv. 51 (1) (2018) 20:1–20:35, http://dx.doi.org/10.1145/3156818, http://doi.acm.org/10.1145/3156818.

[21] L.M. Vaquero, L. Rodero-Merino, R. Buyya, Dynamically scaling applications in the cloud, SIGCOMM Comput. Commun. Rev. 41 (1) (2011) 45–52.

[22] M. Maas, K. Asanović, T. Harris, J. Kubiatowicz, 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, in: ASPLOS '16, ACM, New York, NY, USA, 2016, pp. 457–471.

[23] M. Hauswirth, P.F. Sweeney, A. Diwan, M. Hind, Vertical profiling: Understanding the behavior of object-priented applications, in: Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, in: OOPSLA '04, ACM, New York, NY, USA, 2004, pp. 251–269.

[24] S. Oaks, Java Performance: The Definitive Guide, first ed., O'Reilly Media, Inc., 2014.

[25] G. Pollock, D. Thompson, J. Sventek, P. Goldsack, The Asymptotic Configuration of Application Components in a Distributed System, Technical report, University of Glasgow, Glasgow, UK, 1998, http://eprints.gla.ac.uk/79048/.

[26] Y. Bu, V. Borkar, G. Xu, M.J. Carey, A bloat-aware design for big data applications, in: Proceedings of the 2013 International Symposium on Memory Management, in: ISMM '13, ACM, New York, NY, USA, 2013, pp. 119–130.

[27] D. Gront, A. Kolinski, Utility library for structural bioinformatics, Bioinformatics 24 (4) (2008) 584–585.

[28] A. Francisco, C. Vaz, P. Monteiro, J. Melo-Cristino, M. Ramirez, J. Carrico, Phyloviz: phylogenetic inference and data visualization for sequence based typing methods, BMC Bioinformatics 13 (1) (2012) 87.

[29] J. Simão, S. Esteves, L. Veiga, Smartgc: Online memory management prediction for paas cloud models, in: On the Move To Meaningful Internet Systems. OTM 2017 Conferences - Confederated International Conferences: CoopIS, C&TC, and ODBASE 2017, Rhodes, Greece, October 23-27, 2017, Proceedings, Part I, 2017, pp. 370–388, http://dx.doi.org/10.1007/978-3-319-69462-7_25.

[30] R. Bruno, D. Patricio, J. Simão, L. Veiga, P. Ferreira, Runtime object lifetime profiler for latency sensitive big data applications, in: Proceedings of the Fourteenth EuroSys Conference 2019, in: EuroSys '19, ACM, New York, NY, USA, 2019, pp. 28:1–28:16, http://dx.doi.org/10.1145/3302424.3303988, http://doi.acm.org/10.1145/3302424.3303988.

[31] B. Dantas, C. Fleitas, A. Almeida, J.a. Forja, A.P. Francisco, J. Simão, C. Vaz, Ngspipes: Fostering reproducibility and scalability in biosciences, in: Proceedings of the 8th ACM International Conference on Bioinformatics, Computational Biology,and Health Informatics, in: ACM-BCB '17, ACM, New York, NY, USA, 2017, p. 603, http://dx.doi.org/10.1145/3107411.3108213, http://doi.acm.org/10.1145/3107411.3108213.

[32] A. Lèbre, J. Pastor, M. Bertier, F. Desprez, J. Rouzaud-Cornabas, C. Tedeschi, A.-C. Orgerie, F. Quesnel, G. Fedak, Beyond the clouds, how should next generation utility computing infrastructures be designed?, in: Z. Mahmood (Ed.), Cloud Computing: Challenges, Limitations and R&D Solutions, Springer, 2014, https://hal.inria.fr/hal-01067888.

[33] A.G. Garca, I. Espert, G. Hernndez, SLA-Driven dynamic cloud resource management, Future Gener. Comput. Syst. 31 (2014) 1–11, http://dx.doi.org/10.1016/j.future.2013.10.005.

[34] J. Simão, L. Veiga, A taxonomy of adaptive resource management mechanisms in virtual machines: Recent progress and challenges, in: Cloud Computing, Principles, Systems and Applications, Springer, 2017, pp. 59–98, Ch. 3.

[35] V.F. Rodrigues, R. da Rosa Righi, G. Rostirolla, J.L. Victória Barbosa, C. André da Costa, A.M. Alberti, V. Chang, Towards enabling live thresholding as utility to manage elastic master-slave applications in the cloud, J. Grid Comput. 15 (4) (2017) 535–556, http://dx.doi.org/10.1007/s10723-017-9405-3.

[36] B. Liu, P. Li, W. Lin, N. Shu, Y. Li, V. Chang, A new container scheduling algorithm based on multi-objective optimization, Soft Comput. (2018) http://dx.doi.org/10.1007/s00500-018-3403-7.

[37] C.C. Lin, P. Liu, J.J. Wu, Energy-aware virtual machine dynamic provision and scheduling for cloud computing, in: Proceedings - 2011 IEEE 4th International Conference on Cloud Computing, CLOUD 2011, 2011, pp. 736–737, http://dx.doi.org/10.1109/CLOUD.2011.94.
[38] W. Lin, H. Wang, Y. Zhang, D. Qi, J.Z. Wang, V. Chang, A cloud server energy consumption measurement system for heterogeneous cloud environments, Inform. Sci. 468 (2018) 47–62, http://dx.doi.org/10.1016/j.ins.2018.08.032, http://www.sciencedirect.com/science/article/pii/S0020025518306364.
[39] L. Sharifi, L. Cerdà-Alabern, F. Freitag, L. Veiga, Energy efficient cloud service provisioning: Keeping data center granularity in perspective, J. Grid Comput. 14 (2) (2016) 299–325, http://dx.doi.org/10.1007/s10723-015-9358-3.
[40] ISO/IEC, Information technology Data centres Key performance indicators Part 2: Power usage effectiveness (PUE), 2016, https://www.iso.org/standard/63451.html.
[41] A. Musaddiq, Y.B. Zikria, O. Hahm, H. Yu, A.K. Bashir, S.W. Kim, A survey on resource management in iot operating systems, IEEE Access 6 (2018) 8459–8482, http://dx.doi.org/10.1109/ACCESS.2018.2808324.
[42] O. Agmon Ben-Yehuda, E. Posener, M. Ben-Yehuda, A. Schuster, A. Mu'alem, Ginseng: Market-driven memory allocation, in: Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, in: VEE '14, ACM, New York, NY, USA, 2014, pp. 41–52.
[43] G. Sun, Y. Li, Y. Li, D. Liao, V. Chang, Low-latency orchestration for workflow-oriented service function chain in edge computing, Future Gener. Comput. Syst. 85 (2018) 116–128, http://dx.doi.org/10.1016/j.future.2018.03.018, http://www.sciencedirect.com/science/article/pii/S0167739X18300153.
[44] P. Kathiravelu, P. Van Roy, L. Veiga, Composing network service chains at the edge: A resilient and adaptive software-defined approach, Transactions on Emerging Telecommunications Technologies 29 (11) (2018) e3489, http://dx.doi.org/10.1002/ett.3489, https://onlinelibrary.wiley.com/doi/abs/10.1002/ett.3489, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/ett.3489, e3489 ett.3489.
[45] J. Sun, S. Sun, K. Li, D. Liao, A.K. Sangaiah, V. Chang, Efficient algorithm for traffic engineering in cloud-of-things and edge computing, Comput. Electr. Eng. 69 (2018) 610–627, http://dx.doi.org/10.1016/j.compeleceng.2018.02.016, http://www.sciencedirect.com/science/article/pii/S0045790617333876.
[46] N. Bobroff, P. Westerink, L. Fong, Active control of memory for Java virtual machines and applications, in: 11th International Conference on Autonomic Computing (ICAC 14), USENIX Association, Philadelphia, PA, 2014, pp. 97–103.
[47] C. Cameron, J. Singer, 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.
[48] S.M. Blackburn, P. Cheng, K.S. McKinley, Myths and realities: the performance impact of garbage collection, in: Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems, 2004, pp. 25–36, http://dx.doi.org/10.1145/1005686.1005693.
[49] S.M. Blackburn, K.S. McKinley, 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, in: PLDI '08, ACM, New York, NY, USA, 2008, pp. 22–32.
[50] R. Shahriyar, S.M. Blackburn, X. Yang, K.S. McKinley, Taking off the gloves with reference counting immix, in: A.L. Hosking, P.T. Eugster, C.V. Lopes (Eds.), OOPSLA, ACM, 2013, pp. 93–110.
[51] S. Soman, C. Krintz, Application-specific garbage collection, J. Syst. Softw. 80 (2007) 1037–1056.
[52] L. Chen, G. Serazzi, D. Ansaloni, E. Smirni, W. Binder, What to expect when you are consolidating: effective prediction models of application performance on multicores, Cluster Comput. 17 (1) (2014) 19–37.
[53] D.R. White, J. Singer, J.M. Aitken, R.E. Jones, Control theory for principled heap sizing, in: Proceedings of the 2013 International Symposium on Memory Management, in: ISMM '13, ACM, New York, NY, USA, 2013, pp. 27–38.
[54] J. Singer, G. Kovoor, G. Brown, M. Luján, Garbage collection auto-tuning for java mapreduce on multi-cores, in: Proceedings of the International Symposium on Memory Management, 2011, pp. 109–118.
[55] K. Nguyen, K. Wang, Y. Bu, L. Fang, J. Hu, G.H. Xu, FACADE: a compiler and runtime for (almost) object-bounded big data applications, in: ASPLOS, ACM, 2015, pp. 675–690.
[56] E. Andreasson, F. Hoffmann, O. Lindholm, To collect or not to collect? machine learning for memory management, in: Proceedings of the 2nd Java Virtual Machine Research and Technology Symposium, USENIX Association, Berkeley, CA, USA, 2002, pp. 27–39.
[57] J. Singer, G. Brown, I. Watson, J. Cavazos, Intelligent selection of application-specific garbage collectors, in: Proceedings of the 6th International Symposium on Memory Management, in: ISMM '07, ACM, New York, NY, USA, 2007, pp. 91–102.
[58] M. Hauswirth, P.F. Sweeney, A. Diwan, Temporal vertical profiling, Softw. Pract. Exp. 40 (8) (2010) 627–654.
[59] J. Rao, C.-Z. Xu, Online Capacity identification of multitier websites using hardware performance counters, IEEE Trans. Parallel Distrib. Syst. 22 (3) (2011) 426–438.
[60] F.T. Schneider, M. Payer, T.R. Gross, Online optimizations driven by hardware performance monitoring, in: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation, 2007, pp. 373–382.
[61] A.-R. Adl-Tabatabai, R.L. Hudson, M.J. Serrano, S. Subramoney, Prefetch injection based on hardware monitoring and object metadata, in: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation, in: PLDI '04, ACM, New York, NY, USA, 2004, pp. 267–276, http://dx.doi.org/10.1145/996841.996873.
[62] R. Jones, A. Hosking, E. Moss, The Garbage Collection Handbook: The Art of Automatic Memory Management, first ed., Chapman & Hall/CRC, 2011.
[63] N. Sachindran, J.E.B. Moss, E.D. Berger, Mc2: High-performance garbage collection for memory-constrained environments, in: Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, in: OOPSLA '04, ACM, New York, NY, USA, 2004, pp. 81–98, http://dx.doi.org/10.1145/1028976.1028984, http://doi.acm.org/10.1145/1028976.1028984.
[64] D. Detlefs, C. Flood, S. Heller, T. Printezis, Garbage-first garbage collection, in: Proceedings of the 4th International Symposium on Memory Management, in: ISMM '04, ACM, New York, NY, USA, 2004, pp. 37–48, http://dx.doi.org/10.1145/1029873.1029879, http://doi.acm.org/10.1145/1029873.1029879.
[65] J. Singer, R.E. Jones, G. Brown, M. Luján, The economics of garbage collection, in: Proceedings of the 2010 International Symposium on Memory Management, in: ISMM '10, ACM, New York, NY, USA, 2010, pp. 103–112.
[66] D. Pavlovic, P. Pepper, D.R. Smith, Formal derivation of concurrent garbage collectors, in: C. Bolduc, J. Desharnais, B. Ktari (Eds.), Mathematics of Program Construction, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 353–376.
[67] S.M. Blackburn, R. Garner, C. Hoffmann, A.M. Khang, K.S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S.Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J.E.B. Moss, B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, B. Wiedermann, The daCapo benchmarks: Java benchmarking development and analysis, in: OOPSLA '06: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, ACM, New York, NY, USA, 2006, pp. 169–190.
[68] S. Chidamber, C. Kemerer, A metrics suite for object oriented design, IEEE Trans. Softw. Eng. 20 (6) (1994) 476–493.
[69] N.P. Ricci, S.Z. Guyer, J.E.B. Moss, Elephant tracks: Portable production of complete and precise gc traces, in: Proceedings of the 2013 International Symposium on Memory Management, in: ISMM '13, ACM, New York, NY, USA, 2013, pp. 109–118.
[70] G.J. Janakiraman, J.R. Santos, Y. Turner, Justrunit: Experiment-based management of virtualized data centers, in: G.M. Voelker, A. Wolman (Eds.), 2009 USENIX Annual Technical Conference, San Diego, CA, USA, June 14-19, 2009, USENIX Association, 2009, https://www.usenix.org/conference/usenix-09/justrunit-experiment-based-management-virtualized-data-centers.
[71] L. Sharifi, N. Rameshan, F. Freitag, L. Veiga, Energy efficiency dilemma: P2p-cloud vs. datacenter, in: IEEE 6th International Conference on Cloud Computing Technology and Science, CloudCom 2014, Singapore, December 15-18, 2014, IEEE, 2014, pp. 611–619, http://dx.doi.org/10.1109/CloudCom.2014.137.
[72] J. Platt, Fast training of support vector machines using sequential minimal optimization, in: B. Schoelkopf, C. Burges, A. Smola (Eds.), Advances in Kernel Methods - Support Vector Learning, MIT Press, 1998.
[73] L. Breiman, Random forests, Mach. Learn. 45 (1) (2001) 5–32, http://dx.doi.org/10.1023/A:1010933404324.

J. Simão is a Professor Adjunto at the Engineering School of the Polytechnic Institute of Lisbon (ISEL), where he lectures about programming methodologies, virtualization technologies and is regent of the computer security and cloud computing courses. He is part of the coordination board for bachelor and master courses in Computer Science and Engineering courses at ISEL. He is an Associate Researcher at INESC-ID Lisboa. Holds a Ph.D. in Computer Engineering, since 2015, from University of Lisbon (IST) with the thesis "Economics inspired Adaptive Resource Allocation and Scheduling for Cloud Environments". He has a MSc also from ULisboa, in 2008, with a dissertation about privacy enhanced location services. His main research interests include resource management in clusters and cloud infrastructures,

with a special focus on scheduling algorithms and virtualization technologies, both at language and system level. He also has an interest in computer security, having participated in a project for the National Security Office to develop new cryptographic services using the Portuguese Public Key Infrastructure, as well as short-term training and specialized consulting for the software industry.

L. Veiga is an Associate Professor (tenured), and Executive Director for Faculty Affairs in the Computer Science and Engineering Department at Instituto Superior Técnico, Universidade de Lisboa. He teaches courses on Middleware for Distributed Internet Applications, Virtual Execution Environments and Cloud Computing and Virtualization. He is a Senior Researcher at INESC-ID, Group Manager for Distributed Systems Group, and Vice-Coordinator for Computing Systems and Computer Networks. He coordinates locally at INESC-ID FP7 project CloudForEurope, and National project ContexTWA, on context-aware pervasive sensing Portugal Telecom. He has coordinated 4 other concluded National projects since 2010, on resource management and data processing in decentralized P2P systems, scheduling for cloud & virtualization, multi-core programming and replicated data consistency. He led Tasks in FP7 project Timbus on digital preservation and virtualization, and leads Tasks in H2020 TRACE project on privacy-enabled mobile and sensor tracking services, and evaluated EU FP7 and third-country project proposals (Belgium). He has over 100 peer-reviewed scientific publications in journals, conferences, book chapters, workshops (Best Paper Award at Middleware 2007, Best-Paper Award Runner Up at IEEE CloudCom 2013, Best-Paper Candidate at CloudCom 2014). He was General Chair for Middleware 2011, and belonged to Middleware Steering and Program Committee. He was Virtualization track co-Chair for IEEE CloudCom 2013, Local Chair for Euro-Par 2014 Track on Distributed Systems and Algorithms, and Program Chair for IEEE CSE 2015. He is Associate Editor, Lead of the Emerging Middleware Thematic Series, in Springer Journal of Internet Services and Applications (JISA). He is Workshops Chair for ACM EuroSys 2016. He was an "Excellence in Teaching in IST" mention recipient (2008, 2012, 2014), and awarded Best Researcher Overall at INESC-ID Prize (2014, nominated 2013, 2015) and Best Young Researcher at INESC-ID Prize (2012, nominated 2010, 2011). He is a member of the Scientific Board of the MSc in Computer Engineering and Telecommunications, of Erasmus Mundus European Master and Joint Doctorate in Distributed Computing. He is Member of ACM, and Senior Member, Chair of IEEE Computer Society Chapter, IEEE Section Portugal for 2014–2016. He has served terms on CS Department Council, CS-Ph.D. Management Board, and University Assembly. He belongs to the Permanent Committee of the IST School Assembly.