QoE-JVM: An Adaptive and Resource-Aware Java Runtime for Cloud Computing

José Simão[1][3] and Luís Veiga[1][2]

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

Abstract. Cloud computing has been dominated by system-level virtual machines to enable the management of resources using a coarse grained approach, largely in a manner independent from the applications running on these infrastructures. However, in such environments, although different types of applications can be running, the resources are often delivered in a equal manner to each one, missing the opportunity to manage the available resources in a more efficient and application aware or driven way.
Our proposal is QoE-JVM supporting Java applications with a global and elastic distributed image of a high-level virtual machine (HLL-VM), where total resource consumption and allocation (within and across applications in the infrastructure) are driven by incremental gains in *quality-of-execution* (QoE), which relates the resources allocated to an application and the performance the application can extract from having those resources. In this paper, we discuss how critical resources (memory and CPU) can be allocated among HLL-VMs, so that Cloud providers can exchange resource slices among virtual machines, continually addressing where those resources are required, while being able to determine where the reduction will be more economically effective, i.e., will contribute in lesser extent to performance degradation.

1 Introduction

Workloads running on cluster-enabled infrastructures (e.g. Cloud computing) are supported by different levels of virtualization. In these environments, applications running on high-level language virtual machines (e.g. JVM, CLR) make use of services provided by the operating system, which shares hardware resources through the underlying hypervisor (e.g. Xen, VMWare Server). The complexity of this execution stack makes the allocation of resources fitting the application's needs (e.g. execution time, monetary cost) a challenging task.

System virtual machines provide tools and programmatic interfaces to determine the management policy of the fine-grained resources they control (e.g. memory reservation, CPU proportional share). Nevertheless, we are still far from being able to influence an application behavior, effectively (wide range and impact), efficiently (low overhead) and flexibly (with no or little intrusive coding).

As more applications target managed runtimes, high level language virtualization is a relevant abstraction layer that has not been properly explored to enhance resource usage, control, and effectiveness. Therefore, a managed environment must adapt itself to the execution of applications, being executed by multiple tenants, with different (and sometimes dynamically changing) requirements in regard to their *quality-of-execution* (QoE). QoE aims at capturing the adequacy and efficiency of the resources provided to an application according to its needs. It can be inferred coarsely from application execution time for medium running applications, or request execution times for more service driven ones such as those web-based, or from critical situations such as thrashing or starvation. Also, it can be derived with more fine-grain from incremental indicators of application progress, such as amount of input processed, disk and network output generated, execution phase detection or memory pages updates.

QoE can be used to drive a VM economics model, where the goal is to incrementally obtain gains in QoE for VMs running applications requiring more resources or for more privileged tenants. This, while balancing the relative resource savings drawn from other tentants' VMs with perceived performance degradation. To achieve this goal, certain applications will be positively discriminated, reconfiguring the mechanisms and algorithms that support their execution environment (or even engaging available alternatives to these mechanisms/algorithms). For other applications, resources must be restricted, imposing limits to their consumption, regardless of some performance penalties (that should also be mitigated). In any case, these changes should be transparent to the developer and specially to the application's user.

Existing work on adaptability in cluster-enabled runtimes does not properly support the proposed scenario. Existing public runtimes (Java, CLR) are not resource-aware, as they are mostly bounded by the underlying operative system. On the other hand, in the research community, the proposed runtimes are focused on accounting resource usage to avoid application's bad behavior, and do not support the desired reconfigurability of their inner mechanisms [12, 4]. There is no notion of *resource effectiveness* in the sense that when there are scarce resources, there is no attempt to determine where to take such resources from applications (i.e. either isolation domains or the whole VM) where they hurt performance the least. Others have recently shown the importance of adaptability at the level of HLL-VMs, either based on the application performance [14] or by changes in their environment [11]. Nevertheless, they are either dependent on a global optimization phase, limited to a given resource, or make the application dependent on a new programming interface.

This paper presents the QoE model which we use to determine from which tenants resource scarcity will hurt performance the least, putting resources where they can do the most good to applications and the cloud infrastructure provider. We describe the integration of this model into QoE-JVM, a distributed execution environment where nodes cooperate to make an efficient management of the available local and global resources. At a lower-level, the QoE-JVM is a cluster-enabled runtime with the ability to monitor base mechanisms (e.g. thread scheduling; garbage collection; CPU, memory or network consumptions) to assess application's performance and the ability to reconfigure these mechanisms at runtime. At a higher-level, it drives resource adaptation according to a VM economics model based on aiming overall quality-of-execution through resource efficiency.

Section 2 presents the rationale of our adaptations metrics, based on QoE and resource effectiveness. Section 3 discusses the overall architecture of QoE-JVM and how the application progress can be measured transparently along with the type of resources that are relevant to be adapted at runtime. Section 4 describes the current implementation effort regarding an adaptive managed runtime. Section 5 shows the improvements in the QoE of several well known applications, and the cost of the modifications made to the runtime. Section 6 relates our research to other systems in the literature, framing them with our contribution and Section 7 closes and makes the final remarks about our work.

2 QoE-JVM Economics

Our goal with QoE-JVM is to maximize the applications' *quality of execution* (QoE). We initially regard QoE as a best effort notion of *effectiveness* of the resources allocated to the application, based on the computational work actually carried out by the application (i.e., by employing those allocated resources). To that end the Cobb-Douglas [6] production function from Economics to motivate and to help characterize the QoE, as decribed next.

We are partially inspired by the Cobb-Douglas [7] production function (henceforth referred as equation) from Economics to motivate and to help characterize the QoE. The Cobb-Douglas equation, presented in Equation 1, is used in Economics to represent the *production* of a certain good.

$$Y = A \cdot K^{\alpha} \cdot L^{\beta} \tag{1}$$

In this equation, P is the total production, or the revenue of all the goods produced in a given period, L represents the labor applied in the production and K is the capital invested.

It asserts the now common knowledge (not at the time it was initially proposed, ca. 1928) that *value* in a society (regarded simplistically as an economy) is created by the combined employment of human work (*labour*) and *capital* (the ability to grant resources for a given project instead of to a different one). The extra elements in the equation (A, α, β) are mostly mathematical fine-tuning artifacts that allow tailoring the equation to each set of *real-life* data (a frequent approach in social-economic science, where exact data may be hard to attain and to assess). They take into account technological and civilization multiplicative factors (embodied in A) and the relative weight (cost, value) of capital (α) and labour (β) incorporated in the production output (e.g., more capital intensive operations such as heavy industry, oil refining, or more labour intensive such as teaching and health care).

Alternatively, labour can be regarded, not as a variable representing a measure of human work employed, but as a result, representing the efficiency of the capital invested, given the production output achieved, i.e., labour as a multiplier of resources into production output. This is usually expressed by representing Equation 1 in terms of L, as in Equation 2. For simplicity, we have assumed all the three factors to be equal to one. First, the technological and civilization context does not apply, and since the data center economy is simpler, as there is a single kind of activity, computation, and not several, the relative weight of labour and capital is not relevant. Furthermore, we will be more interested in the variations (relative increments) of efficiency than on efficiency values themselves, hence the simplification does not introduce error.

$$L = \frac{Y}{K} \tag{2}$$

Now, we need to map these variables to relevant factors in a cloud computing site (a data center). *Production output* (Y) maps easily to application progress (the amount of computation that gets carried out), while *capital* (K), associated with money, maps easily to resources committed to the application (e.g., CPU, memory, or their pricing) that are usually charged to users deploying applications. Therefore, we can regard labour (considered as the *human factor*, the efficiency of the capital invested in a project, given a certain output achieved) as how effectively the resources were employed by an application to attain a certain progress. While resources can be measured easily by CPU shares and memory allocated, application progress is more difficult to characterize. We give details in Section 3 but we are mostly interested in *relative variations* in application progress (regardless of the way it is measured), as shown in Equation 3, and their complementary variations in *production cost per unit*, *PCU*.

$$\varDelta\ L \approx \frac{\varDelta\ Y}{\varDelta\ K}, \quad and \quad thus \quad \varDelta\ PCU \approx \frac{\varDelta\ K}{\varDelta\ Y} \tag{3}$$

We assume a scenario where, when applications are executed in a constrained (overcommitted) environment, the infrastructure may remove m units of a given resource, from a set of resources R, and give it to another application that can benefit from this transfer. This transfer may have a negative impact in the application who offers resources and it is expected to have a positive impact in the receiving application. To assess the effectiveness of the transfer, the infrastructure must be able to measure the impact on the giver and receiver applications, namely somehow to approximate savings in PCU as described next.

Variations in the PCU can be regarded as an opportunity for *yield* regarding a given resource r, and a management strategy or allocation configuration s_x, i.e., a return or reward from applying a given strategy to some managed resource, as presented in Equation 4.

$$Yield_r(ts, s_a, s_b) = \frac{Savings_r(s_a, s_b)}{Degradation(s_a, s_b)} \tag{4}$$

Because QoE-JVM is continuously monitoring the application progress, it is possible to incrementally measure the yield. Each partial $Yield_r$, obtained in a given time span ts, contributes to the total one obtained. This can be evaluated either over each time slice or globally when applications, batches or workloads complete. For a given execution or evaluation period, the total yield is the result of summing all significant partial yields, as presented in Equation 5.

$$TotalYield_r(s_a, s_b) = \sum_{ts=0}^{n} Yield_r(ts, s_a, s_b) \tag{5}$$

The definition of $Savings_r$ represents the savings of a given resource r when two allocation or management strategies are compared, s_a and s_b, as presented in Equation 6. The functions $U_r(s_a)$ and $U_r(s_b)$ relates the *usage* of resource r, given two allocation configurations, s_a and s_b. We allow only those reconfigurations which offer savings in resource usage to be considered in order to calculate yields.

$$Savings_r(s_a, s_b) = \frac{U_r(s_a) - U_r(s_b)}{U_r(s_a)} \tag{6}$$

Regarding *performance degradation*, it represents the impact of the savings, given a specific performance metric, as presented in Equation 7. Considering the time taken to execute an application (or part of it), the performance degradation relates the execution time of the original configuration, $P(s_a)$, and the execution time after the resource allocation strategy has been modified, $P(s_b)$.

$$Degradation(s_a, s_b) = \frac{P(s_b) - P(s_a)}{P(s_a)} \tag{7}$$

Each instance of the QoE-JVM continuously monitors the application progress, measuring the *yield* of the applied strategies. As a consequence of this process, QoE, for a given set of resources, can be enforced observing the *yield* of the applied strategy, and then keeping or changing it as a result of having a good or a bad impact. To accomplish the desired reconfiguration, the underlying resource-aware VM must be able to change strategies during execution, guided by the global QoE manager. The next section will present the architecture of QoE-JVM detailing how progress can be measured and which resources are relevant. Section 4 shows how existing high-level language virtual machines can be extended to accommodate the desired adaptability.

3 Architecture

Figure 1 presents the overall architecture of our distributed JVM *platform as a service* for Cloud environments. Our vision is that QoE-JVM will execute applications with different requisites regarding their QoE. Target applications have

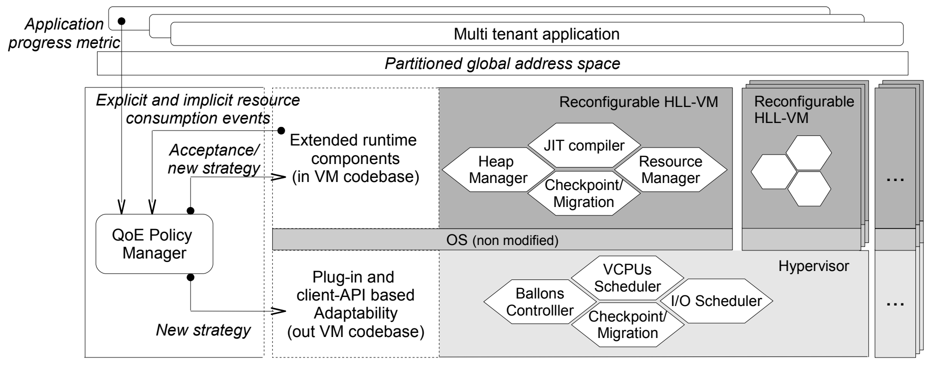


Fig. 1: Overall architecture

typically a long execution time and can spawn several execution flows to parallelize their work. This is common in the field of science supported by informatics like economics and statistics, computational biology and network protocols simulation.

QoE-JVM is supported by several runtime instances, eventually distributed by several computational nodes, each one cooperating to the sharing of resources. For an effective resource sharing, a global mechanism must be in place to make weak (e.g. change parameters) or strong (e.g. change GC algorithm, migrate running application) adaptations [16]. QoE-JVM encompasses a distributed shared objects middleware, a reconfigurable high-level language virtual machine (HLL-VM), and, at the bottom, available reconfigurable mechanisms of system level virtual machine (Sys-VM). In this architecture, the operating system (OS) services are only used, not extended.

In this article, we emphasize the extensions made to a HLL-VM, in order to support monitoring and control of resources along with the adaptive mechanisms. Regarding the other two lower levels of the *platform stack* (OS and hypervisor) our work focus is on using existing APIs to maximize portability of current solutions and acceptability, in particular using the management API of current hypervisors. The hypervisor's internal components are depicted at the bottom of Figure 1. Some of then can be directly configured, as it is the case of parameters regarding the CPU scheduler or the ballooning mechanism to reclaim guest's memory. Guest operating systems provide several internal tools that typically do not have a remote API and are mostly used for profiling proposes, not to influence the application behavior. A well known exception is the priority parameter of OS processes (e.g. the `nice` parameter of a linux process).

Each instance of an HLL-VM is enhanced with services that are not available in regular VMs. These services include: i) the accounting of resource consumption, ii) dynamic reconfiguration of internal parameters and/or mechanisms, and iii) mechanisms for checkpointing, restore and migration of the whole application. These services should and must be made available at a lower-level, inside an extended HLL-VM, for reasons of control, interception and efficiency. They are transparent to the application, and so, the extended HLL-VM continue to run existent applications as they are. To effectively apply the economic model presented in Section 2 it is necessary to quantify the application progress metric, what resources are relevant and which extensions points exist or need to be created inside the HLL-VM.

3.1 Progress monitoring

We classify applications as request driven (or interactive) and continuous process (or batch). Request driven applications process work in response to an outside event (e.g. http request, new work item in the processing queue). Continuous processing applications have a target goal that drives their calculations (e.g. align DNA sequences). For most non-interactive applications, measuring progress is directly related to the work done and the work that is still pending. For example, some algorithms to process graphs of objects have a completed objects set, which will encompass all objects when the algorithm terminates. Other examples would be applications to transform video encoding, where the number of frames processed is a measure of progress.

There is a balance between application semantics and transparency of progress measuring. While the number of requests in a request driven application gives the best notion of progress, it will not always be possible to acquire that information. On the other hand, low level activity, such as I/O or memory pages access, is always possible to acquire inside the VM or the OS. The following are relevant examples of metrics that can be used to monitor the progress of an application, presented in a decreasing order of application semantics, but with an increasing order regarding transparency.

- **Number of requests processed**: This metric is typically associated with interactive applications, like Bag-of-tasks environments or Web applications;
- **Completion time**: For short and medium time living applications, where it is not possible to change the source code or no information is available to lead an instrumentation process, this metric will be the more effective one. This metric only requires the QoE-JVM to measure *wall clock time* when starting and ending the application;
- **Code: instrumented or annotated**: If information is available about the application high level structure, instrumentation can be used to dynamically insert probes at load time, so that the QoE-JVM can measure progress using a metric that is semantically more relevant to the application;
- **I/O: storage and network**: For application dependent on I/O operations, changes in the quantity of data saved or read from files or in the information sent and received from the network, can contribute to determine if the application reached a bottleneck or is making progress;
- **Memory page activity**: Allocation of new memory pages is a low level indicator (collected from the OS or the VMM) that the application is making effective progress. A similar indication will be given when the application is writing in new or previous memory pages.

	CPU	*Mem*	*Net*	*Disk*	*Pools*
Counted	number of cores	size	-	-	size (min, max)
Rate	cap percentage	growth/ shrink rate	I/O rate	I/O rate	-

Table 1: Implicit resources and their throttling properties

Although QoE-JVM can measure low level indicators as I/O storage and network activity or memory page activity, Section 5 uses the metric completion time because the applications used to demonstrate the benefits of our system have a short execution time.

3.2 Resource types and usage

We consider resources to be any measurable computational asset which applications consume to make progress. Resources can be classified as either *explicit* or *implicit*, regarding the way they are consumed. *Explicit* resources are the ones that applications request during execution, such as, number of allocated objects, number of network connections, number of opened files. *Implicit* resources are consumed as the result of execution the application, but are not explicitly requested through a given library interface. Examples include, the heap size, the number of cores, network transfer rate.

Both types of resource are relevant to be monitored and regulated. *Explicit* and *implicit* resources might be constrained as a protection mechanism against ill behaved or misusing application [12]. For well behaved application, restraining these resources further below the application contractual levels will lead to an execution failure. On the other hand, the regulation of *implicit* resources determines how the application will progress. For example, allocating more memory will potentially have a positive impact, while restraining memory will have a negative effect. Nevertheless, giving too much of memory space is not a guarantee that the application will benefit from that allocation, while restraining memory space will still allow the application to make some progress. In this work we focus on controlling *implicit* resources because of their elastic characteristic. QoE-JVM can control the admission of these resources, that is, it can throttle resource usage. It gives more to the applications that will progress faster if more resources are allocated. Because resources are finite, they will be taken from (or not given to) other applications. Even so, the QoE-JVM will strive to choose the applications where progress degradation is comparatively smaller.

Table 1 presents *implicit* resources and the throttling properties associated to each one. These proprieties can be either counted values or rates. To regulate CPU and memory both types of properties are applicable. For example, CPU can be throttled either by controlling the number of cores or the percentual cap. Memory usage can be regulated either through a fixed limit or by using a factor to shrink or grow this limit. Although the heap size cannot be smaller than the working set, the size of the over committed memory influences the application progress. A similar rational can be made about resource pools.

4 The Adaptive Managed Runtime

Controlling resource usage inside a HLL-VM can be carried out by either *i)* intercepting calls to classes in the classpath that virtualize access to system resources or *ii)* changing parameters or algorithms of internal components. Examples of the first hook type are classes such as `java.util.Socket` (where the number of bytes sent and received per unit of time can be controlled) and `java.util.concurrent.ThreadPoolExecutor` class (where the parameters minimum and maximum control the number of threads). An example of the second hook type is the memory management subsystem (garbage collection strategy, heap size).

We have implemented the JSR 284 - Java Management API, delegating resource consumption decisions to a callback, handled by a new VM component, the Resource Management Layer (RML), either allowing, denying it (e.g. by throwing an exception) or delaying it which allows not breaking application semantics. In a previous work [19] we describe the details to enforce the JSR-284 semantics and basic middleware support to spawn threads in remote nodes (i.e., across the cluster). In this paper we show how the yield-driven adaptation process, discussed in Section 2, can be used to govern global application and thread scheduling and placement, regarding other two fundamental resources - heap size and CPU usage.

4.1 Memory: yield-driven heap management

The process of garbage collection relates to execution time but also to allocated memory. On the CPU side, a GC algorithm must strive to minimize the pause times (more so if stop-the-world type). On the memory side, because memory management is virtualized, the allocated memory of a managed language runtime is typically bigger than the actual working set. With many runtimes executing in a shared environment, it is important to keep them using the least amount of memory to accomplish their work.

Independently of the GC algorithm, runtimes can manage the maximum heap size, mainly determining the maximum used memory. The memory management system of the research runtime Jikes RVM [1] uses a built-in matrix to determine how the heap will grow, shrink or remain unchanged, after a memory collection. The growing factor is determined by the ratio of live objects and the time spent in the last GC operation. The default rates determine that the heap shrinks about 10% when the time spent in GC is low (less than 7%) when compared to regular program execution, and the ratio of live objects is also low (less than 50%). This allows for savings in memory used. On the other hand, the heap will grow for about 50%, as the time spent in GC also grows and the number of live objects remains high. This growth in heap size will lead to an increase in memory used by the runtime, aiming to use less CPU time because the GC will run less frequently.

Considering this heap management strategy, the *heapsize* is the resource which contributes to the *yield* measurement. To determine how the workloads executed by each tenant react to different heap management strategies, such as, more heap expander or more heap saving, we apply the Equation 6. The memory saving ($Savings_{hsize}$) is then found comparing the results of applying two different allocation matrices, M_α and M_β, as presented in Equation 8. In this equation, U_{hsize} represents the maximum number of bytes assigned to the heap.

$$Savings_{hsize} = \frac{U_{hsize}(M_\alpha) - U_{hsize}(M_\beta)}{U_{hsize}(M_\alpha)} \tag{8}$$

4.2 CPU: yield-driven CPU ballooning

A similar approach can be extended to CPU management employing a strategy akin to *ballooning*[4]. In our case, the ballooning is carried out by taking CPU from an application by assigning a single core and adjusting the CPU priority of its encompassing JVM instance. This makes the physical CPUs available for other VMs. This is achieved by engaging the Resource Management Layer of our modified JVM that, ultimately, either interfaces with the OS, or with a system VM, such as Xen [3], lowering CPU caps.

Thus, regarding CPU as the resource, the savings in computational capability (that can be transferred to other tenants) can be measured in FLOPS or against a known benchmark as Linpack. The savings are found by comparing two different CPU shares or priorities, CPU_α and CPU_β, as presented in Equation 9. In this case U_{flops} give us the total CPU saved, e.g., relative to the number of FLOPS or Linpack benchmarks that can be run with the CPU 'saved'.

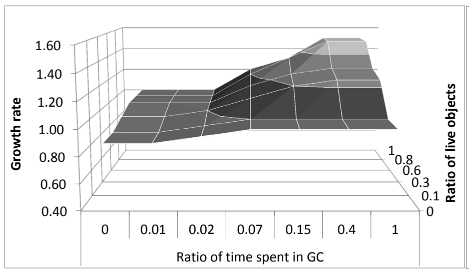
$$Savings_{flops} = \frac{U_{flops}(CPU_\alpha) - U_{flops}(CPU_\beta)}{U_{flops}(CPU_\alpha)} \tag{9}$$

The next section presents the impact of these modifications in the runtime and the results of applying our resource management strategy related to heap management and CPU share.

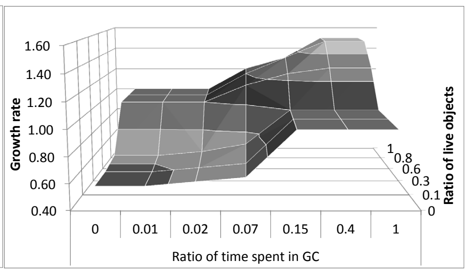
5 Evaluation

In this section we discuss how the resource management economics, presented in Section 2, were applied to manage the heap size and CPU usage regarding

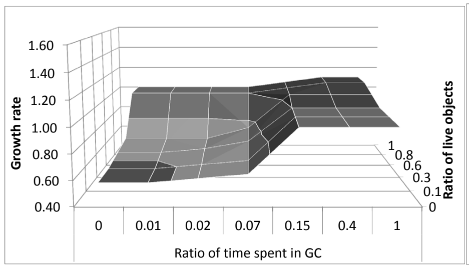
[4] Employed by virtual machine monitors in system VMs, prior to migration, by having a kernel driver allocating memory pages excessively, in order to drive the guest OS to swapping and reduce the amount of useful pages in guest physical memory. This allows the core working set of the guest VM to be obtained with a grey-box approach.



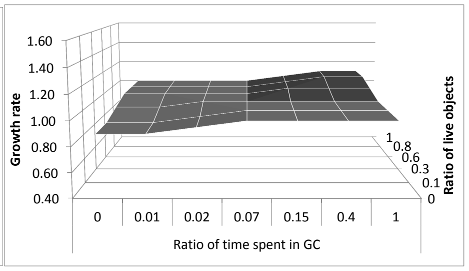
(a) M_0



(b) M_1



(c) M_2



(d) M_3

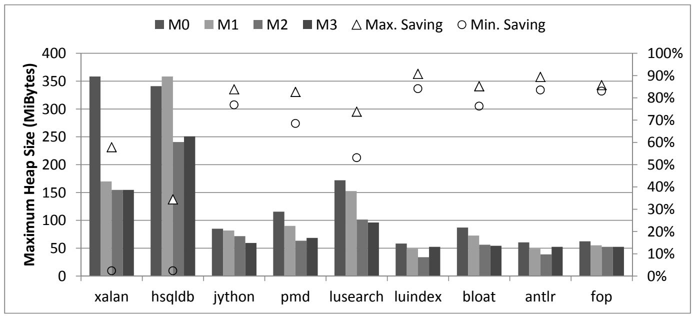
Fig. 2: Default (M_0) and alternative matrices to control the heap growth.

different types of workloads. We evaluated our work using Intel(R) Core(TM)2 Quad processors (with four cores) and 8GB of RAM, running Linux Ubuntu 9.04. Jikes RVM code base is version 3.1.1 and the *production* configuration was used to build the source code.

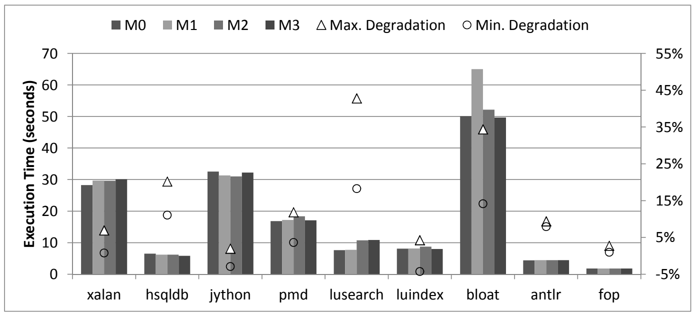
5.1 Heap Size

The default heap growing matrix (hereafter known as M_0) is presented in Figure 2.a. In this, and in the remaining matrices, 1.0 is the neutral value, representing a situation where the heap will neither grow nor shrink. Other values represent a factor of growth or shrink, depending if the value is greater or smaller than 1, respectively. To assess the benefits of our resource management economics, we have setup three new heap size changing matrices. The distinctive *factors* are the growth and decrease rates determined by each matrix.

Matrices M_1 and M_2, presented in Figure 2.b and 2.c, impose a strong reduction on the heap size when memory usage and management activity is low (i.e. few live objects and short time spent on GC). Nevertheless they provide very different growth rates, with M_1 having a faster rate when heap space is scarce. Finally, matrix M_3 makes the heap grow and shrink very slowly, enforcing a more rigid and conservative heap size until program dynamics reach a high activity point (i.e. high rate of live objects and longer time spent on GC) or decrease activity sharply.



(a) Maximum heap size and average savings percentage



(b) Execution time and average performance degradation percentage

Fig. 3: Results of using each of the matrices ($M_{0..3}$), including savings and degradation when compared to a fixed heap size.

Each tenant using the Cloud provider infrastructure can potentially be running different programs. Each of these programs will have a different *production*, i.e. execution time, based on the *capital* applied, i.e. growth rate of the heap. To represent this diversity, we used the well known DaCapo benchmarks [5], a set of programs which explore different ways of organizing programs in the Java language.

To measure the *yield* of each matrix we have setup an *identity matrix* (all 1's), that is, a matrix that never changes the heap size. Figures 3.a shows the maximum heap size (left axis) after running the DaCapo benchmarks with configuration `large` and a maximum heap size of 350 MiBytes, using all the matrices presented in Figure 2. In the right axis we present the maximum and minimum of *resource savings*, as defined in Equation 6. These values were obtained for each of the matrices when compared to the *identity matrix* with heap size fixed at 350 MiBytes. The *resource savings* are above 40% for the majority of the workloads, as can be seen in more detail in Table 2.

In Figure 3.b we present the evaluation time of the benchmarks (left axis) and the average *performance degradation* (right axis), as defined in Equation 7,

Matrix	M0			M1			M2			M3		
	Sav	Deg	**Yield**	Sav	Deg	**Yield**	Sav	Deg	**Yield**	Sav	Deg	**Yield**
xalan	2.3%	0.8%	*3.1*	53.7%	5.5%	*9.7*	57.9%	5.4%	***10.7***	57.9%	6.9%	*8.4*
hsqldb	7.1%	20.2%	*0.4*	2.3%	16.4%	*0.1*	34.4%	16.2%	*2.1*	31.6%	11.1%	***2.9***
jython	76.8%	1.9%	*39.9*	77.7%	-1.9%	*-40.4*	80.5%	-2.9%	*-27.5*	83.8%	0.8%	***104.6***
pmd	68.5%	3.6%	***18.9***	75.4%	5.9%	*12.8*	82.7%	11.8%	*7.0*	81.3%	5.1%	*16.1*
lusearch	53.1%	18.3%	*2.9*	58.4%	19.3%	***3.0***	72.4%	42.0%	*1.7*	73.8%	42.8%	*1.7*
luindex	84.1%	-2.8%	*-30.4*	86.6%	-2.5%	***-34.6***	90.8%	4.3%	*21.3*	85.7%	-4.3%	*-20.1*
bloat	76.3%	14.8%	*5.2*	80.2%	34.4%	*2.3*	84.7%	18.2%	***4.6***	85.2%	14.2%	*6.0*
antlr	83.5%	7.9%	***10.5***	86.6%	8.9%	*9.7*	89.4%	9.4%	*9.5*	85.7%	8.5%	*10.0*
fop	83.0%	2.7%	*30.7*	84.9%	1.0%	***89.0***	85.7%	2.7%	*31.7*	85.7%	1.8%	*48.1*

Table 2: The *yield* of the matrices presented in Figure 2

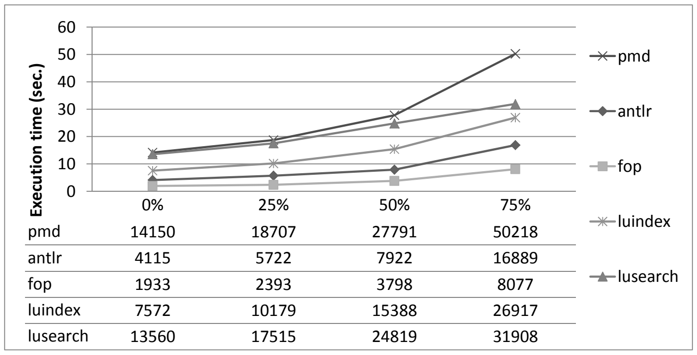
regarding the use of each of the ratio matrices. Degradation of execution time reaches a maximum of 35% for lusearch, Apache's fast text search engine library, but stays below 25% for the rest of the benchmarks. Table 2, summarizes the *yield*, as defined in Equation 4, when using different matrices to manage the heap size.

Two aspects are worth nothing. First, under the same resource allocation strategy, *resource savings* and *performance degradation* vary between applications. This demonstrates the usefulness of applying different strategies to specific applications. If the cloud provider uses M_2 for a tenant running `lusearch` type workload it will have a yield of 1.7. If it uses this aggressive saving matrix in `xalan` type workloads (Apache's XML transformation processor) it will yield 10.7, because it saves more memory but the execution time suffers a smaller degradation. Second, a negative value represents a strategy that actually saves execution time. Not only memory is saved but execution time is also lower. These scenarios are a minority though as they may simply reveal that the 350 $MiBytes$ of fixed heap size is causing to much page faults for that workload.

5.2 CPU

Our system also takes advantage of CPU restriction in a coarse-grained approach. Figure 4 shows how five different Java workloads (taken from the DaCapo benchmarks) react to the deprivation of CPU (in steps of 25%), regarding their total execution time. Figure 5 shows the relative performance slowdown, which represents the *yield* of allocating 75%, 50% and 25%, comparing with 100% of CPU allocation. Note that, comparing with previous graphics, some applications have longer execution times with 0% CPU taken because they are multithreaded and we used only 1 core for this test.

As expected, the execution time grows when more CPU is taken. This enables priority applications (e.g. paying users, priority calculus applications) to run efficiently over our runtime, having the CPU usage transparently restricted and given to others (a capability in itself currently unavailable in HLL-VMs). Finally,



	0%	25%	50%	75%
pmd	14150	18707	27791	50218
antlr	4115	5722	7922	16889
fop	1933	2393	3798	8077
luindex	7572	10179	15388	26917
lusearch	13560	17515	24819	31908

Fig. 4: Effects of restraining CPU by 25%, 50% and 75%

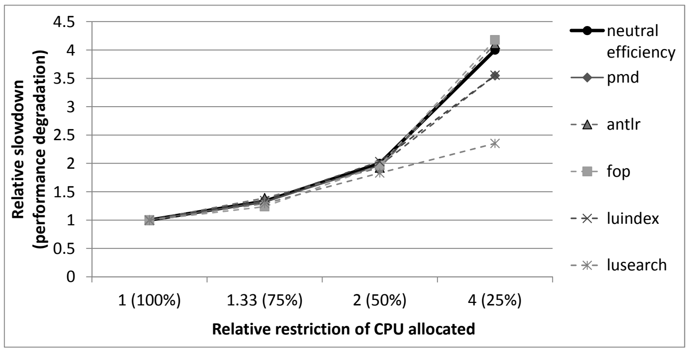


Fig. 5: Relative slowdown

we note that 3 applications (`pmd`, `luindex` and `lusearch`) have yields greater than 1 when CPU restriction is equal or above 50%, as they stay below the neutral efficiency line in Figure 5, due to memory or I/O contention.

6 Related work

Adaptability is a vertical activity in current systems stack. System-wide VMs, high level language VMs and special propose middleware can all make adaptations of their internal mechanisms to improve the system performance in a certain metric.

These three levels of virtualization have different distances to the machine-level resources, with a increasing distance from system-wide VMs to special purpose middleware. The dual of this relation is the transparency of the adaptation process from the application perspective. A system-wide VM aims to distribute resources with fairness, regardless of the application patterns or workload. On the other end is the middleware approach where applications use a special purpose programming interface to specify their consumption restrictions. As more applications target high level language VMs, including the ones running on cloud data centers, this virtualization level, which our work encompasses, has the potential to influence high impact resources (akin to system-wide VMs), using application's metrics (akin to the middleware approach) but still with full transparency. This section presents work related to these three virtualization levels, focusing on adaptations whose goal is to improve the application performance by adapting the infrastructure mechanisms.

In [18], Sharma et al. present a way to dynamically provision virtual servers in a cloud provider, based on pricing models. They target application owners (i.e. suppliers of services to end users) who want to select the best configuration to support their peak workloads (i.e. maximum number of requests per second successfully handled), minimizing the cost for the application's owner. Sharma's work uses different mechanisms to guarantee the provisioning of resources, which include: readjusting CPU, memory caps and migration. To make good decisions they need to know, for each application, what is the peak supported by each provider's configuration, which is dependent on real workloads. Furthermore, because changes to virtual servers configuration is driven by the number of requests per second, it can miss the effective computation power needed by each request.

Shao et al. [17] adapts the VCPU mapping of Xen [3] based on runtime information collect at each guest's operative system. The numbers of VCPUs is adjusted to meet the real needs of each guest. Decisions are made based on two metrics: the average VCPU utilization rate and the parallel level. The parallel level mainly depends on the length of each VCPU's run queue. Shao's work on specific native applications. We believe our approach has the potencial to influence a growing number of applications that run on high-level language virtual machines and whose performance is also heavily dependent on memory management. PRESS [13] tries to allocate just enough resources to avoid service level violations while minimizing resource waste. It tracks resource usage and predicts how resource demands will evolve in the near future. To that end, it employs a signal processing or a statistical method to detect pattern, adjusting resource caps (with a tolerance factor) based on this analysis.

Ginko [14] is an application-driven memory overcommitment framework which allows cloud providers to run more system VMs with the same memory. For each VM, Ginkgo collects samples of the application performance, memory usage, and submitted load. Then, in production phase, instead of assigning the same amount of memory for each VM, Ginko takes the previously built model and, using a linear program, determines the VM ideal amount of memory to avoid violations of service level agreements. Our approach does not need a global optimization step each time we need to transfer resources among VMs.

High level languages virtual machines are subject to different types of adaptation regarding the just in time (JIT) compiler and memory management [2]. Nevertheless, most of them are hard coded and cannot be influenced without building a new version of the VM. In [9], Czajkowski et al. propose to enhance the resource management API of the Multitask Virtual Machine (MVM) [10], forming a cluster of this VMs where there are local and global resources that can be monitored and constrained. However, Czajkowski's work lacks the capacity to determine the effectiveness of resource allocation, relying on predefined allocations. In [15], a reconfigurable monitoring system is presented. This system uses the concept of Adaptable Aspect-Oriented Programming (AAOP) in which monitored aspects can be activated and deactivated based on a management strategy. The management strategy is in practice a policy which determines the resource management constraints that must be activated or removed during the application lifetime.

In [20] the GC is auto-tuned in order to improve the performance of a MapReduce Java implementation for multi-core hardware. For each relevant benchmark, machine learning techniques are used to find the best execution time for each combination of input size, heap size and number of threads in relation to a given GC algorithm (i.e. serial, parallel or concurrent). Their goal is to make a good decision about a GC policy when a new MapReduce application arrives. The decision is made locally to an instance of the JVM. The experiments we presented are also related to memory management, but our definition of QoE (as presented in Section 2) can go beyond this resource.

At the middleware level, Coulson et al. [8] present OpenCom, a component model oriented to the design and implementation of reconfigurable low-level systems software. OpenCom's architecture is divided between the kernel and the extensions layers. While the kernel is a static layer, capable of performing basic operations (i.e. component loading and binding), the extensions layer is a dynamic set of components tailored to the target environment. These extensions can be reconfigured at runtime to, for example, adapt the execution environment to the application's resource usage requisites. Our work handles mechanisms at a lower level of abstraction.

Duran et al. [11] uses a thin high-level language virtual machine to virtualize CPU and network bandwidth. Their goal is to provide an environment for resource management, that is, resource allocation or adaptation. Applications targeting this framework use a special purpose programming interface to specify reservations and adaptation strategies. When compared to more heavyweight approaches like system VMs, this lightweight framework can adapt more efficiently for I/O intensive applications. The approach taken in Duran's work bounds the application to a given resource adaptation interface. Although in our system the application (or the libraries they use) can also impose their own restrictions, the adaptation process is mainly driven by the underlying virtual machine without direct intervention of the applications.

7 Conclusions

In this paper, we described the ongoing research to design QoE-JVM. It aims at offering a distributed execution environment, each executing an extended resource-aware runtime for a managed language, Java, and where resources are allocated based on their effectiveness for a given workload. QoE-JVM has the ability to monitor base mechanisms (e.g. CPU, memory or network consumptions) in order to assess application's performance and reconfigure these mechanisms in runtime.

Resource allocation and adaptation obeys to a VM economics model, based on aiming overall quality-of-execution (QoE) through resource efficiency. Essentially, QoE-JVM puts resources where they can do the most good to applications and the cloud infrastructure provider, while taking them from where they can do the least harm to applications.

We presented the details of our adaptation mechanisms in each VM (for memory, CPU) and their metrics. We experimentally evaluated their benefits, showing resources can be reverted among applications, from where they hurt performance the least (higher yields in our metrics), to more higher priority or requirements applications. The overall goal is to improve flexibility, control and efficiency of infrastructures running long applications in clusters.

Acknowlegments This work was partially supported by national funds through FCT Fundação para a Ciência e a Tecnologia, under projects PTDC/EIA-EIA/102250/2008, PTDC/EIA-EIA/108963/2008, PTDC/EIA-EIA/113613/2009, and PEst-OE/EEI/LA0021/2011

References

1. B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, and V. Sarkar. The Jikes research virtual machine project: building an open-source research community. *IBM Syst. J.*, 44:399–417, January 2005.
2. Matthew Arnold, Stephen J. Fink, David Grove, Michael Hind, and Peter F. Sweeney. A survey of adaptive optimization in virtual machines. In *Proceedings of the IEEE, 93(2), 2005. Special Issue on Program Generation, Optimization, ans Adaptation*, 2005.
3. Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37:164–177, October 2003.
4. Walter Binder, Jarle Hulaas, Philippe Moret, and Alex Villazón. Platform-independent profiling in a virtual execution environment. *Softw. Pract. Exper.*, 39:47–79, January 2009.
5. Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben 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*, pages 169–190, New York, NY, USA, 2006. ACM.
6. Charles Cobb and Paul Douglas. A theory of production. *American Economic Association*, 1:139–165, 1928.
7. C.W. Cobb and P.H. Douglas. A theory of production. *The American Economic Review*, 18(1):139–165, 1928.

8. Geoff Coulson, Gordon Blair, Paul Grace, Francois Taiani, Ackbar Joolia, Kevin Lee, Jo Ueyama, and Thirunavukkarasu Sivaharan. A generic component model for building systems software. *ACM Trans. Comput. Syst.*, 26:1:1–1:42, March 2008.
9. G. Czajkowski, M. Wegiel, L. Daynes, K. Palacz, M. Jordan, G. Skinner, and C. Bryce. Resource management for clusters of virtual machines. In *Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid*, CCGRID '05, pages 382–389, Washington, DC, USA, 2005. IEEE Computer Society.
10. Grzegorz Czajkowski, Stephen Hahn, Glenn Skinner, Pete Soper, and Ciarán Bryce. A resource management interface for the Java platform. *Softw. Pract. Exper.*, 35:123–157, February 2005.
11. H.A. Duran-Limon, M. Siller, G.S. Blair, A. Lopez, and J.F. Lombera-Landa. Using lightweight virtual machines to achieve resource adaptation in middleware. *IET Software*, 5(2):229–237, 2011.
12. N. Geoffray, G. Thomas, G. Muller, P. Parrend, S. Frenot, and B. Folliot. I-JVM: a Java Virtual Machine for component isolation in OSGi. In *IEEE/IFIP International Conference on Dependable Systems & Networks*, 2009.
13. Zhenhuan Gong, Xiaohui Gu, and J. Wilkes. Press: Predictive elastic resource scaling for cloud systems. In *Network and Service Management (CNSM), 2010 International Conference on*, pages 9 –16, oct. 2010.
14. Michael Hines, Abel Gordon, Marcio Silva, Dilma Da Silva, Kyung Dong Ryu, and Muli Ben-Yehuda. Applications know best: Performance-driven memory overcommit with ginkgo. In *CloudCom '11: 3rd IEEE International Conference on Cloud Computing Technology and Science*, 2011.
15. Arkadiusz Janik and Krzysztof Zielinski. AAOP-based dynamically reconfigurable monitoring system. *Information & Software Technology*, 52(4r):380–396, 2010.
16. Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4:14:1–14:42, May 2009.
17. Zhiyuan Shao, Hai Jin, and Yong Li. Virtual machine resource management for high performance computing applications. *Parallel and Distributed Processing with Applications, International Symposium on*, 0:137–144, 2009.
18. Upendra Sharma, Prashant Shenoy, Sambit Sahu, and Anees Shaikh. A cost-aware elasticity provisioning system for the cloud. In *Proceedings of the 2011 31st International Conference on Distributed Computing Systems*, ICDCS '11, pages 559–570, Washington, DC, USA, 2011. IEEE Computer Society.
19. José Simão, João Lemos, and Luís Veiga. A^2-VM a cooperative Java VM with support for resource-awareness and cluster-wide thread scheduling. In *Proceedings of the Confederated international conference on On the move to meaningful internet systems*, OTM'11, pages 302–320. Springer-Verlag, 2011.
20. Jeremy Singer, George Kovoor, Gavin Brown, and Mikel Luján. Garbage collection auto-tuning for java mapreduce on multi-cores. In *Proceedings of the International Symposium on Memory Management*, ISMM'11, pages 109–118, New York, NY, USA, 2011. ACM.