Comput Syst Sci & Eng (2013) 1: 59–72
© 2013 CRL Publishing Ltd

International Journal of
Computer Systems
Science & Engineering

Adaptability driven by quality of execution in high level virtual machines for shared cloud environments

José Simão[1,3] **and Luís Veiga**[1,2]

[1] *INESC-ID Lisboa*
[2] *Instituto Superior Técnico (IST), Portugal*
[3] *Instituto Superior de Engenharia de Lisboa (ISEL), Portugal*
E-mail: jsimao@cc.isel.ipl.pt, luis.veiga@inesc-id.pt

Cloud infrastructures execute workloads from different tenants supported by a non-trivial virtualization stack, which includes high language virtual machines, operating system services and system-level virtual machines. As more and more applications target high level virtual machines (such as the Java VM), they are a relevant abstraction layer not properly explored to enhance resource usage, control, and effectiveness. We propose an economics-inspired model to balance relative resource savings (e.g., to prioritize tenants) and perceived performance degradation, resulting in a *yield* of applying a given management strategy. The model can be used to drive a resource scheduling algorithm aiming to determine where the reduction will be more economically effective, i.e., will contribute in lesser extent to performance degradation. We discuss how critical resources (heap size and CPU) can be allocated and transferred among high level virtual machines. Experimental evaluation shows that the application of our model, when choosing to take the appropriate resource allocation, results in a significant yield to the cloud provider while, in most cases, execution degradation is small.

1. INTRODUCTION

Workloads running on cluster-enabled infrastructures (e.g. Cloud Infrastructures) are supported by different levels of virtualization. In these environments, applications running on high-level language virtual machines (e.g. Java Virtual Machine, Common Language Runtime) make use of services provided by the operating system, which shares hardware resources through the underlying hypervisor (e.g. Xen [5], VMWare ESX Server [31]). 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.

Although in shared environments such as clouds [24], cycle sharing infrastrucures [25] or shared computing cluster in general, different types of applications can be running, resources are often delivered in an equal manner to each one, missing the opportunity to manage the available resources in a more efficient and application aware or driven way. 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, they are still far from being fully able to influence the behavior of a specific application, effectively (wide range and impact), efficiently (low overhead) and flexibly (with no or little intrusive coding). This is so because clients of system VM are not the applications but operating systems which make it more difficult (from the system VM point of view) to distinguish progress and influence specific application progress.

As more applications target managed runtimes, High Level Language Virtual Machines (HLL-VM) are a relevant abstraction layer that has not been properly explored to enhance resource usage, control, and effectiveness, with increased rich semantics and flexibility. Therefore, managed runtimes, executing the workloads of multiple tenants, must adapt themselves to the execution of applications, 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. Several metrics can be used to infer how applications are making progress given the resources they are using. 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 in a more fine-grained way from incremental indicators of application progress, such as amount of input processed, disk and network output generated, execution phase detection or memory pages updates. Still, for the time being, we will consider the application execution times.

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 tenants' 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 especially to the application's user.

Although existing public runtimes (Java, CLR) incorporate security models that can be used to sandbox the execution of some components, they simple allow or deny a certain operation to proceed according to the sandbox permission set. In the research community, the proposed extended runtimes are focused on accounting resource usage to avoid application's bad behavior, and do not support the reconfiguration of their inner mechanisms [15, 7]. Furthermore, existing work on adaptability in cluster-enabled runtimes does not properly support the proposed scenario. Meanwhile, others have recently shown the importance of adaptability at the level of HLL-VMs, either based on the application performance [17] or by changes in their environment [14]. 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 an economics-inspired model to drive adaptability in environments where resources are shared by several tenants. Our adaptability model is used 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. In this work we apply the yield-based model to measure the results of different strategies regarding a) the heap size, based on the relation between the ratio of live objects and the time spent in GC; b) CPU allocation in a per workload way.

Section 2 presents the rationale of our adaptation model, based on the yield obtained from applying different strategies to each tenant's workload. Section 3 discusses the overall architecture of QoE-JVM and how 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, describing how the adaptation model is used to manage critical resources such as memory and CPU. Section 5 shows the improvements in the QoE of several well-known applications. Section 6 relates our research to other systems in the literature, framing them with our contribution. Finally, 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, we resort to the Cobb-Douglas production function from Economics to motivate and to help characterize the QoE, as described next.

As said, we are partially inspired by the Cobb-Douglas [9] 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.

$$Y = A \cdot K^{\alpha} \cdot L^{\beta} \qquad (1)$$

In this equation, Y is the total production, or the revenue of all the goods produced in a given period, L represents the labour 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 the three extra elements 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, according to relative variations in resources (to assess *resource efficiency*), and their complementary variations in *production cost per unit*, PCU, as an approximation of the marginal cost (capital), in resources, to achieve the obtained progress (output). The term *unit* is a generic one because we want to apply this rationale to different kinds of resources, as described next.

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

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 (e.g. memory size, CPU cores, bandwidth) and give it to another application that can benefit from this transfer. Examples of transferable units are 50 *MiBytes* of heap size, 1 core and 2 *MiBytes* of bandwidth. This transfer may have a negative impact in the application that 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 measure the approximate savings in PCU, that is, the relation between employed resources and effective progress, as described next.

Variations in the PCU can be regarded as an opportunity for *yield* regarding a given resource r, and a *management strategy*. The term *strategy* generically identifies the currently in use and the available configuration options. Naturally, comparing strategy s_a and s_b only makes sense if they are of the same nature. For example, s_a and s_b can represent different kinds of garbage collection algorithms or different ratios to grow/shrink the heap size. So, the *yield* is a return or reward from applying a given strategy to some managed resource, during the time span ts, 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 $U_r(s_b)$ relate the *usage* of resource r, given two different management strategies or allocation configuration, s_a and s_b. For example, if r represents memory then U would be total bytes currently allocated. 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.

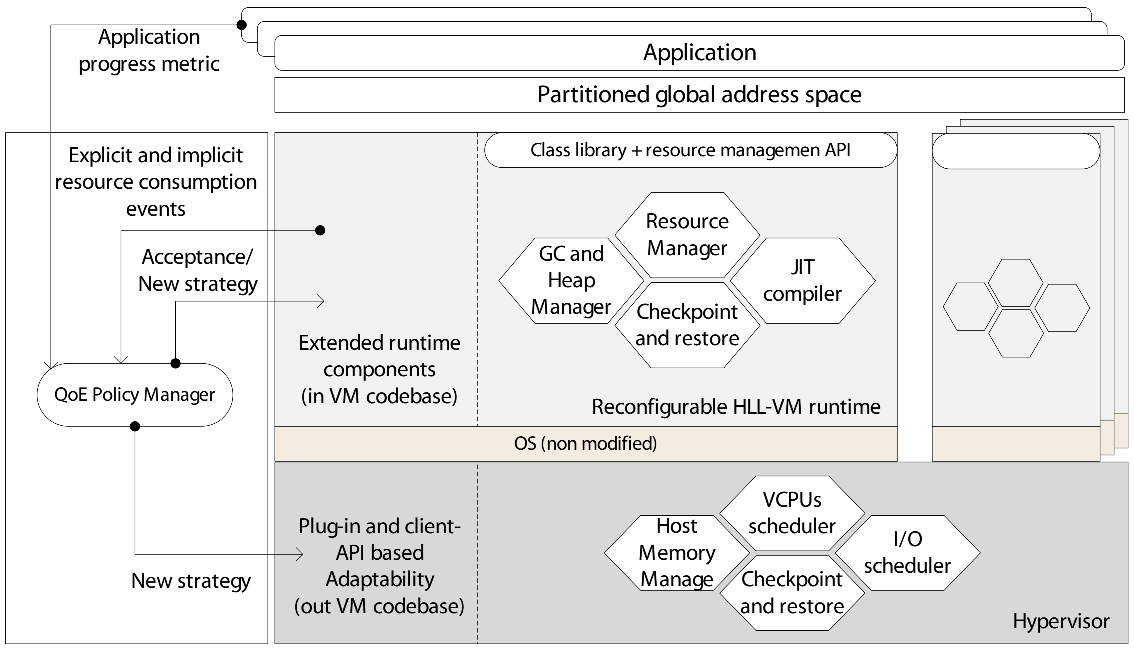

Application progress metric
Application
Partitioned global address space
Explicit and implicit resource consumption events
Acceptance/ New strategy
QoE Policy Manager
Extended runtime components (in VM codebase)
Class library + resource managemen API
Resource Manager
GC and Heap Manager
JIT compiler
Checkpoint and restore
Reconfigurable HLL-VM runtime
OS (non modified)
Plug-in and client-API based Adaptability (out VM codebase)
New strategy
VCPUs scheduler
Host Memory Manage
I/O scheduler
Checkpoint and restore
Hypervisor

Figure 1: Overall architecture

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 requirements regarding their QoE. Target applications have 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 or strong adaptations, as defined by Salehie et al. in [21]. Weak adaptation represents adaptations that change parameters (e.g. modify the heap growth ratio), while strong adaptation deals with replacing components. The former is usually a light weighted enforcement. The later can improve the quality of the overall system but has a potentially high cost of enforcement, as it is the case of changing the garbage collection algorithm itself or migrating the application to another node. QoE-JVM encompasses a distributed shared objects middleware and a reconfigurable high-level language virtual machine (HLL-VM). Regarding these layers, we have augmented existing middleware and implemented new functionalities inside the HLL-VM [26].

Each instance of an HLL-VM is enhanced with services that are not available in regular VMs. These services include a) the accounting of resource consumption; b) dynamic reconfiguration of internal parameters and/or mechanisms; c) 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 continues to run existing applications as they are. In this work we emphasis on weak/light adaptations while in [28] we have proposed a strong adaptation mechanism to checkpoint and migrate the execution state of a HLL-VM.

Although the architecture includes available adaptations mechanisms, both at the operating system (OS) and system level virtual machine (Sys-VM), we avoid extending new functionalities in these layers to maximize the portability of our solution and acceptability.

The hypervisor's internal components are depicted at the bottom of Figure 1. Some of them can be directly configured, as it is the case of parameters regarding the CPU scheduler and the host memory. In the case of the memory manager, an extra mechanism of enforcement must be available at each guest OS to effectively reclaim memory, i.e. the *balloon driver*[31]. 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).

The policy manager is responsible for loading and fulfilling the policies provided by administrators and possibly users regarding resource management. It achieves this by, globally, sending the necessary commands to the resource-aware VMs, in order for them to modify some runtime parameters, or the type of algorithm used. In this work, our policy is based on an economics-inspired one which was detailed in Section 2. The policy takes into account the resources allocated and the perceived progress of each tenant, so that when a reconfiguration is necessary, the selected tenant will be the one to be hurt the least.

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 extension points exist or need to be created inside the HLL-VM. The next two subsections will discuss these topics with further detail.

3.1 Resource types and usage

In the model presented at Section 2, $Savings_r$ refers to any computational resource (r) 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 and number of opened files. *Implicit* resources are consumed as the result of executing the application, but are not explicitly requested through a given library interface. Examples include the heap size, the number of cores or the 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 [15]. For well-behaved applications, 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 some types of *implicit* resources because of their potential to provide elasticity to resource management. 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 (e.g. x number of cores) or rates (e.g. y KiBytes/seconds). 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 *cap* (i.e. the maximum percentage of CPU a VM is able to use, even if there is available CPU time). 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 rationale can be made about resource pools (e.g. thread pools, connection pools), which are a common strategy to manage resources in applications handling multiple requests, such as web and database servers.

3.2 Progress monitoring

Our economics-inspired metric needs to take as input the *performance degradation* of the application. In practical terms, performance relates to the progress, slower or faster, the application can make with the allocated resources.

To compare different metrics to measure progress, 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 analyze graphs of objects have a visited/processed object set, which will encompass all objects when the algorithm terminates. If the rate of objects processed can be determined it will indicate how the application is making progress. Other examples would be applications to perform video encoding, where the number of frames processed is a measure of progress [18, 20].

There is a balance and trade-off in measuring progress, using a metric that is close to the application semantics, and the transparency of progress measuring. The number of requests processed, for example, is metric closely related to the application semantic, which gives an almost direct notion of progress. Nevertheless, 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. But relating this type of metrics to the application effective progress is a challenging task. 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;
- **Mutator execution time**. When mutators (i.e. execution flows of applications) have high execution percentages, in proportion to the time spent in garbage collector, this indicate that the application is making more progress than others where garbage collection is using a higher percentage of total execution.
- **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.

Table 1: Implicit resources and their throttling properties

	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	-

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** to measure performance degradation, as defined in Section 2. This is so because the applications used to demonstrate the benefits of our system are benchmarks that are representative of different types of workloads and have a short execution time. However, the metric **mutator execution time** also has an important role because the strategies to manage the heap, described in Section 4, take into account the dual of this metric - the ratio of time spent in garbage collection. In [27] we show how annotated and instrumented code can also be used to measure the application progress and drive runtime adaptability.

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 (i.e. garbage collection algorithm, 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 it, denying it (i.e. by throwing an exception), or delaying it, which allows not breaking application semantics. This extension was built into the research virtual machine Jikes RVM [3] and the GNU classpath. In a previous work [26] 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 two other fundamental resources - heap size and CPU usage.

4.1 Memory: yield-driven heap management

The process of garbage collection (GC) 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 of 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. In the memory management system of the research runtime Jikes RVM [3], after a full heap collection, the runtime considers whether the heap should change size. The algorithm to change the heap size takes into account the percentage of live objects and the ratio of time spent in GC. Live objects ratio measures the relation between the total memory reserved (which includes large, immortal and non-movable objects spaces) and the space reserved for regular objects allocation, as presented in Equation 8.

$$liveObjectsRatio = \frac{reservedMemory}{currentHeapSize} \tag{8}$$

On the other hand, the ratio of time spent in GC measures the relation between the accumulated time spent in GC related activities and total application time, as presented in Equation 9.

$$gcTimeRatio = \frac{\sum_{collection=0}^{n} GCDuration_{collection}}{totalAppTime} \tag{9}$$

This heap size change policy is represented by a function that takes as input the *liveObjectsRatio* and the *gcTimeRatio*, returning the heap size growth/shrink percentage. In this paper we refer to this function as a *matrix* because it maps two parameters to one. The default policy 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%, when the execution environment spends more time in GC activities, and the number of live objects still 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 targeted at heap expansion or more at heap saving, we apply Equation 6. The memory savings ($Savings_{hsize}$) are then found by comparing the results of applying two different allocation policies, that is, two different allocation matrices, M_α and M_β, as presented in Equation 10. In this equation, U_{hsize} represents the maximum number of bytes assigned to the heap.

$$Savings_{hsize} = \frac{U_{hsize}(M_\alpha) - U_{hsize}(M_\beta)}{U_{hsize}(M_\alpha)} \tag{10}$$

4.2 CPU: yield-driven CPU ballooning

A similar approach can be extended to CPU management employing a strategy akin to *ballooning*.[1] 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 available confinement mechanisms at the OS layer such as *containers* [30, 6], akin approaches specifically designed for CPU scheduling [13], or with the hypervisor CPU scheduler [5, 1], 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 11. 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'.

$$Savings_{\mathit{flops}} = \frac{U_{\mathit{flops}}(CPU_\alpha) - U_{\mathit{flops}}(CPU_\beta)}{U_{\mathit{flops}}(CPU_\alpha)} \qquad (11)$$

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.

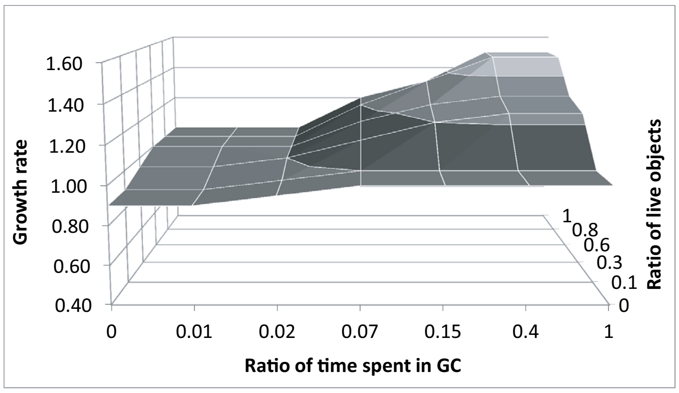
5. EVALUATION

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 different types of workloads. We evaluated our work using Intel(R) Core(TM) i7 (with four cores), 8MiBytes of cache and 12GiBytes of RAM, running Linux Ubuntu 12.04 LTS. Jikes RVM code base is version 3.1.2 and the *production* configuration was used to build the source code.

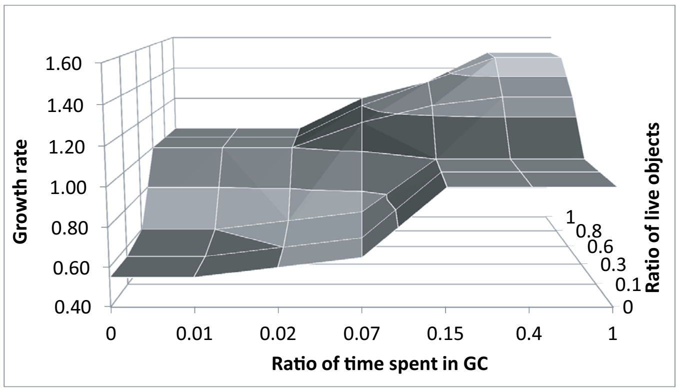
5.1 QoE Yield applied to memory management

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 designed and set up three new heap size changing matrices. The distinctive *factors* are the growth and decrease rates determined by each matrix.

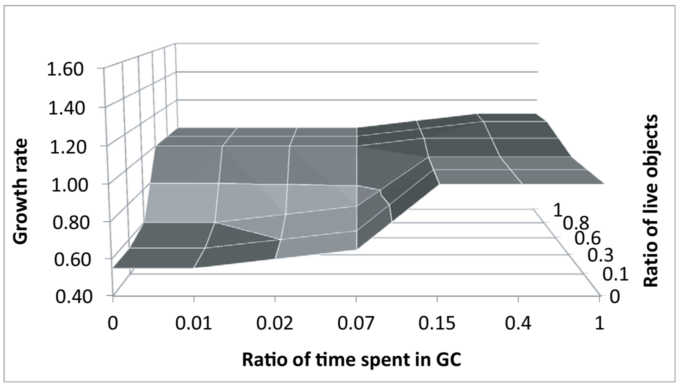
Matrices M_1 and M_2, presented in Figure 2.b and 2.c, impose a strong reduction on the heap size when memory usage and management activity is low (i.e. few live objects and short time spent on GC). Nevertheless they provide very different growth rates, with M_1 having a faster rate when heap space is scarce.



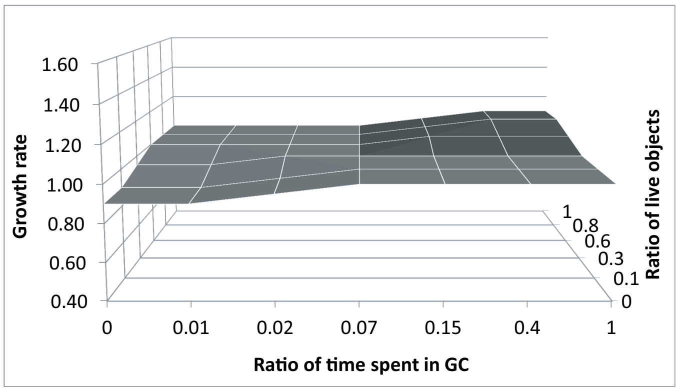
(a) M_0



(b) M_1



(c) M_2



(d) M_3

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

[1]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.

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.

The overall behavior of the matrices can be visualized, in alternative, as directives for percentage increases and reductions in heap size, being depicted with colored arrows in Figure 3-a to -d.

Furthermore, to compare the matrices from a quantitative point of view we define two norms, the *growth norm*, presented in Equation 12 and the *shrink norm*, presented in Equation 13. These capture the aggressiveness of impact (when expanding or shrinking), that the different matrices have in the heap size, and its skew/bias towards expansion or shrinkage (more or less expander, or shrinking-driven). In particular, Equation 12 calculates a sum, across the two dimensions of the matrix (inspired by an integral over a plane, but in a discrete domain), aggregating only the net contributions for heap expansion found in the matrix (only the part greater than 1 in each element). Conversely, Equation 13 calculates a sum, across the two dimensions of the matrix (also inspired by an integral over a plane), this time aggregating only the net contributions for heap reduction in size.

$$\|M\|_{growth} = \sum_{i=1}^{n}\sum_{j=1}^{m} g(a_{ij}),$$

$$\text{where } g(x) = \begin{cases} x-1 & \text{if } x > 1, \\ 0 & \text{if } x <= 1 \end{cases} \tag{12}$$

$$\|M\|_{shrink} = \sum_{i=1}^{n}\sum_{j=1}^{m} s(a_{ij}),$$

$$\text{where } s(x) = \begin{cases} 1-x & \text{if } x < 1, \\ 0 & \text{if } x >= 1 \end{cases} \tag{13}$$

Table 2 summarizes the norms of the four matrices (M_0 from the base implementation of Jikes RVM, and M_1, M_2 and M_3 proposed alternatives). This helps us classifying a matrix and to quickly infer its expected behavior. Matrix M_0 is clearly biased towards expansion as its growth norm is greater than its shrink norm. This bias is assessed by the 8.08 ratio meaning an eight-fold potential more impact towards expansion than towards shrinkage. For instance, matrix M_2 has an opposite bias, as its norm ratio is 0.14, roughly 7 times more potential impact towards shrinkage than towards growth. Both of them have similar aggressiveness regarding aggregated impact (5.45 and 5.40).

On the other hand, matrices M_1 and M_3 have much reduced and almost no bias, as their growth and shrinkage norms are almost equivalent, hence the ratio around one (1.08 and 0.85). However they are very different relating to aggressiveness or potential impact, top for M_1 and very small for M_3.

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. the growth rate behavior allowed for the the heap. To represent this diversity, we used benchmarks from DaCapo [8] and SPECjvm2008 [2], which correspond to different ways of organizing programs in the Java language.

Figure 4 shows how the previously discussed matrices influence the use of different heap grow/shrink ratios when executing the benchmarks. Each figure plots in the y-axis the frequency, in a normalized form, with which a given growth (x-value greater than 1) or shrink (x-value lower than 1) ratio is used, after a decision to change the heap size. From Figure 4.b, we can confirm that matrix M_1 is the one with a potential for larger impact (aggressiveness) because it causes heap change percentages with values from a wider interval, when compared to the remainder matrices. On the other hand, Figure 4.d corroborates that matrix M_3 is the one with the smallest aggressiveness as it uses the smallest interval of values.

Table 2: Growth and shrink norms and their relation

	M_0	M_1	M_2	M_3	interpretation
$\|\cdot\|_{growth}$	4.85	4.05	0.65	0.65	
$\|\cdot\|_{shrink}$	0.60	4.75	4.75	0.60	
$\frac{\|\cdot\|_{growth}}{\|\cdot\|_{shrink}}$	8.08	0.85	0.14	1.08	(bias/skew)
$\|\cdot\|_{growth} + \|\cdot\|_{shrink}$	5.45	8.08	5.40	1.35	(aggressiveness/ potential)

Figure 5.a shows the maximum heap size (left axis) after running the DaCapo benchmarks with configuration `large` and a subset of SPECjvm2008[2] using all the matrices presented in Figure 2. The *yield* of each matrix is compared with a scenario where the memory management system uses a *neutral matrix* (all 1's) with a heap size fixed to 350 MiBytes. When using the four matrices, the heap size was configured to change between a minimum of 50MiBytes and a maximum of 350MiBytes. In the right axis we present the average *resource savings*, as defined in Equation 6. The *resource savings* are above 40% for the majority of the workloads, as can be seen in more detail in Table 3.

In Figure 5.b we present the evaluation time of the benchmarks (left axis) and the average *performance degradation* (right axis), as defined in Equation 7, regarding the use of each of the matrices. Degradation of execution time reaches a fourfold value for lusearch, Apache's fast text search engine library, but stays below 25% for the fast majority of the benchmarks. In particular regarding the SPECjvm2008 benchmarks, most of them have negative *performance degradation*, that is, they run faster with the growth/shrink matrices controlling the heap than with a fixed size. Table 3, summarizes the *yield*, as defined in Equation 4.

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 too many page faults for that workload.

[2]Given to incompatibilities with the GNU classpath not all SPECjvm2008 can be successfully executed in Jikes RVM

Ratioof time spent in GC	Ratio of live objects					
	0%	10%	30%	60%	80%	100%
0%	▼ -10%	▼ -10%	▼ -5%	▬ 0%	▬ 0%	▬ 0%
1%	▼ -10%	▼ -10%	▼ -5%	▬ 0%	▬ 0%	▬ 0%
2%	▼ -5%	▼ -5%	▬ 0%	▬ 0%	▬ 0%	▬ 0%
7%	▬ 0%	▬ 0%	▲ 10%	▲ 15%	▲ 20%	▲ 20%
15%	▬ 0%	▬ 0%	▲ 20%	▲ 25%	▲ 35%	▲ 30%
40%	▬ 0%	▬ 0%	▲ 25%	▲ 30%	▲ 50%	▲ 50%
100%	▬ 0%	▬ 0%	▲ 25%	▲ 30%	▲ 50%	▲ 50%

(a) M_0 percentage increments

Ratioof time spent in GC	Ratio of live objects					
	0%	0%	0%	0%	0%	0%
0%	▼ -45%	▼ -45%	▼ -40%	▬ 0%	▬ 0%	▬ 0%
1%	▼ -45%	▼ -45%	▼ -40%	▬ 0%	▬ 0%	▬ 0%
2%	▼ -40%	▼ -40%	▼ -35%	▬ 0%	▬ 0%	▬ 0%
7%	▼ -35%	▼ -35%	▼ -30%	▲ 15%	▲ 20%	▲ 20%
15%	▬ 0%	▬ 0%	▬ 0%	▲ 25%	▲ 35%	▲ 30%
40%	▬ 0%	▬ 0%	▬ 0%	▲ 30%	▲ 50%	▲ 50%
100%	▬ 0%	▬ 0%	▬ 0%	▲ 30%	▲ 50%	▲ 50%

(b) M_1 percentage increments

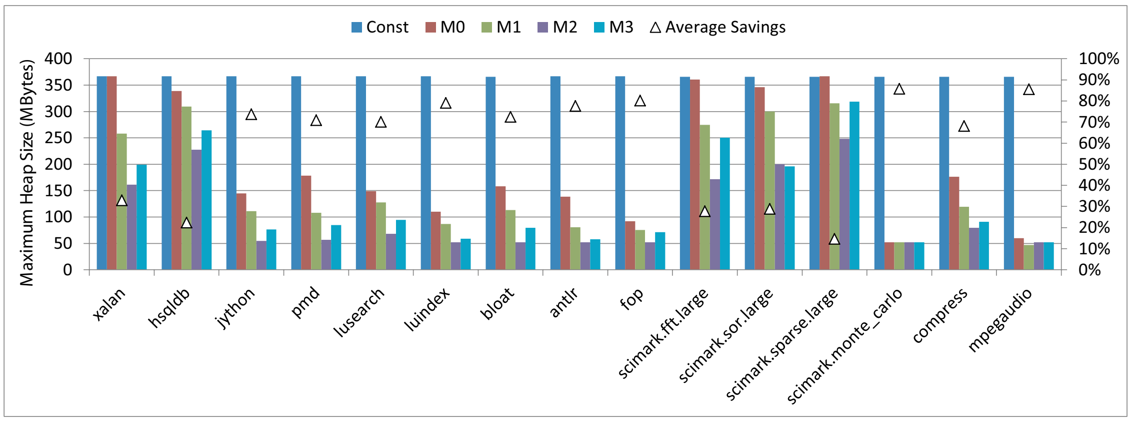
Ratioof time spent in GC	Ratio of live objects					
	0%	0%	0%	0%	0%	0%
0%	▼ -45%	▼ -45%	▼ -40%	▬ 0%	▬ 0%	▬ 0%
1%	▼ -45%	▼ -45%	▼ -40%	▬ 0%	▬ 0%	▬ 0%
2%	▼ -40%	▼ -40%	▼ -35%	▬ 0%	▬ 0%	▬ 0%
7%	▼ -35%	▼ -35%	▼ -30%	▬ 0%	▬ 0%	▬ 0%
15%	▬ 0%	▬ 0%	▬ 0%	▲ 5%	▲ 5%	▲ 5%
40%	▬ 0%	▬ 0%	▬ 0%	▲ 5%	▲ 10%	▲ 10%
100%	▬ 0%	▬ 0%	▬ 0%	▲ 5%	▲ 10%	▲ 10%

(c) M_2 percentage increments

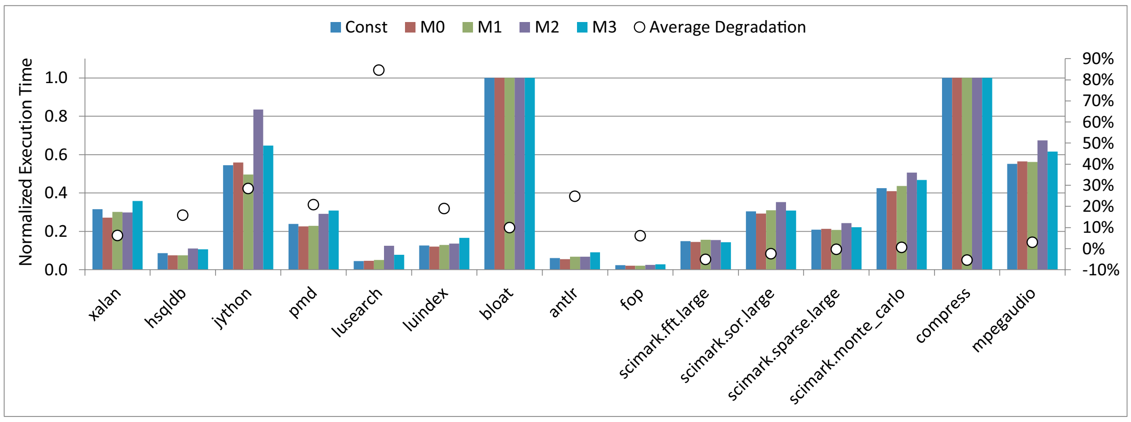
Ratioof time spent in GC	Ratio of live objects					
	0%	0%	0%	0%	0%	0%
0%	▼ -10%	▼ -10%	▼ -5%	▬ 0%	▬ 0%	▬ 0%
1%	▼ -10%	▼ -10%	▼ -5%	▬ 0%	▬ 0%	▬ 0%
2%	▼ -5%	▼ -5%	▬ 0%	▬ 0%	▬ 0%	▬ 0%
7%	▬ 0%	▬ 0%	▬ 0%	▬ 0%	▬ 0%	▬ 0%
15%	▬ 0%	▬ 0%	▬ 0%	▲ 5%	▲ 5%	▲ 5%
40%	▬ 0%	▬ 0%	▬ 0%	▲ 5%	▲ 10%	▲ 10%
100%	▬ 0%	▬ 0%	▬ 0%	▲ 5%	▲ 10%	▲ 10%

(d) M_3 percentage increments

Figure 3: Growth and shrink percentage for each matrix



(a) Maximum heap size and average savings percentage



(b) Execution time and average performance degradation percentage

Figure 5: Results of using each of the matrices ($M_{0..3}$), including savings and degradation when compared to a fixed heap size.

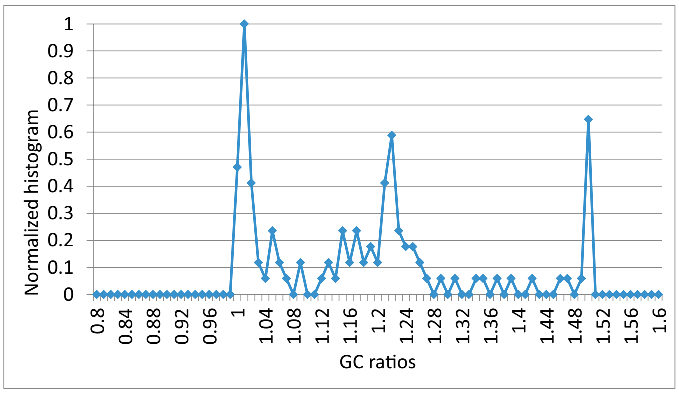
Table 3: Heap Size Savings, Execution Degradation and Yield

	xalan	hsqldb	jython	pmd	lusearch	luindex	bloat	antlr	fop	scimark.fft.large	scimark.sor.large	scimark.sparse.large	scimark.monte_carlo	compress	mpegaudio
								Savings							
M0	0.0%	7.7%	60.6%	51.4%	59.4%	70.0%	56.7%	62.3%	74.9%	1.4%	5.4%	-0.3%	85.7%	51.9%	83.7%
M1	29.7%	15.7%	69.7%	70.6%	65.1%	76.3%	69.1%	78.0%	79.4%	24.9%	18.1%	13.8%	85.7%	67.3%	87.1%
M2	56.0%	38.0%	85.1%	84.6%	81.4%	85.7%	85.7%	85.7%	85.7%	53.0%	45.3%	32.1%	85.7%	78.2%	85.7%
M3	45.7%	28.0%	79.1%	76.9%	74.3%	84.0%	78.2%	84.3%	80.6%	31.5%	46.4%	12.9%	85.7%	75.1%	85.7%
								Degradation							
M0	-1.5%	-1.2%	17.6%	8.6%	20.3%	9.1%	14.5%	3.9%	-2.1%	1.2%	-0.1%	6.4%	0.3%	4.2%	6.7%
M1	7.4%	-3.5%	2.5%	7.7%	26.1%	14.7%	12.4%	24.8%	-3.2%	2.7%	-0.1%	-2.6%	0.8%	-1.9%	-0.1%
M2	11.2%	50.9%	80.5%	43.7%	225.5%	26.9%	17.7%	31.0%	18.7%	-13.0%	-2.9%	-2.7%	-0.2%	-16.2%	2.4%
M3	7.6%	17.0%	13.1%	23.2%	66.2%	25.0%	-4.9%	39.4%	10.8%	-11.3%	-6.7%	-2.4%	1.5%	-7.8%	3.0%
								Yield							
M0	0.0	-6.6	3.4	6.0	2.9	7.7	3.9	15.8	-36.3	1.2	-89.6	0.0	280.8	12.5	12.4
M1	4.0	-4.5	28.4	9.2	2.5	5.2	5.6	3.1	-24.7	9.4	-297.1	-5.3	106.3	-36.4	-649.2
M2	5.0	0.7	1.1	1.9	0.4	3.2	4.8	2.8	4.6	-4.1	-15.9	-11.9	-393.2	-4.8	35.2
M3	6.0	1.7	6.1	3.3	1.1	3.4	-15.8	2.1	7.5	-2.8	-6.9	-5.4	57.8	-9.6	28.7

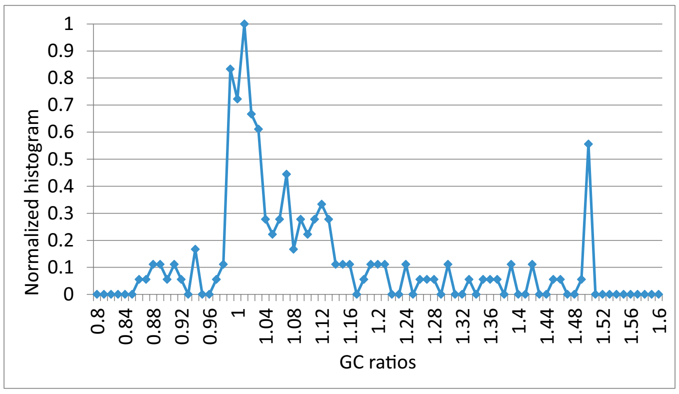
5.2 QoE Yield applied to CPU usage

Our system also takes advantage of CPU restriction in a coarse-grained approach. Figure 6 shows how five different Java workloads (taken from the DaCapo benchmarks) react to the deprivation of CPU (in slices of 25%), regarding their total execution time. Figure 7 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.

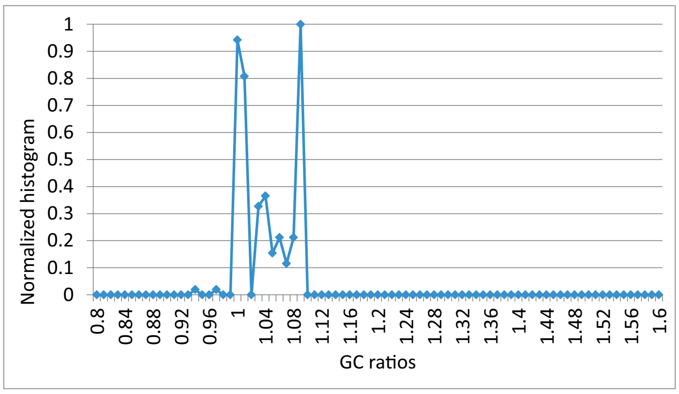
As expected, the execution time grows when more CPU is taken. This enables priority applications (e.g. paying users, pri-



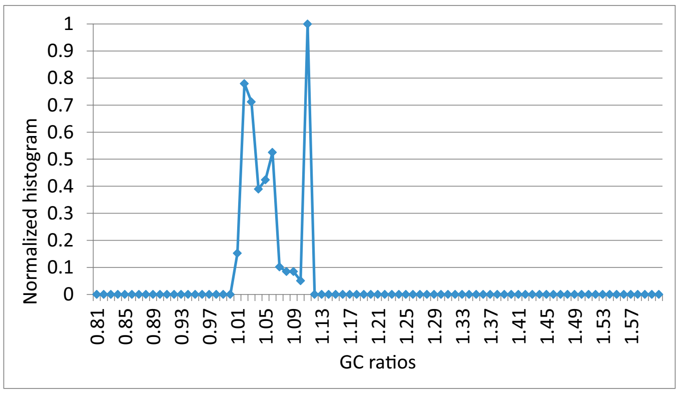
(a) M_0



(b) M_1



(c) M_2



(d) M_3

Figure 4: Histogram of GC ratios for each benchmark

ority 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, we note that the following applications (`hsqldb`, `fop` and `antlr`) have yields greater than 1 when CPU restriction is equal or above 50%, as they stay below the neutral efficiency line in Figure 7, due to memory or I/O contention. This is more clearly visible in Figure 8, where we can see those applications having relative efficiency gains when resources are partially restricted (and to what extent), as well as those that loose relative efficiency, but always to a smaller extent.

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 an 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 [23], 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 the configuration of virtual servers is driven by the number of requests per second, it can miss the effective computation power needed by each request.

Shao et al. [22] adapts the VCPU mapping of Xen [5] 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 potential 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 [16] tries to allocate just enough resources to avoid service level violations while minimizing resource waste. It tracks

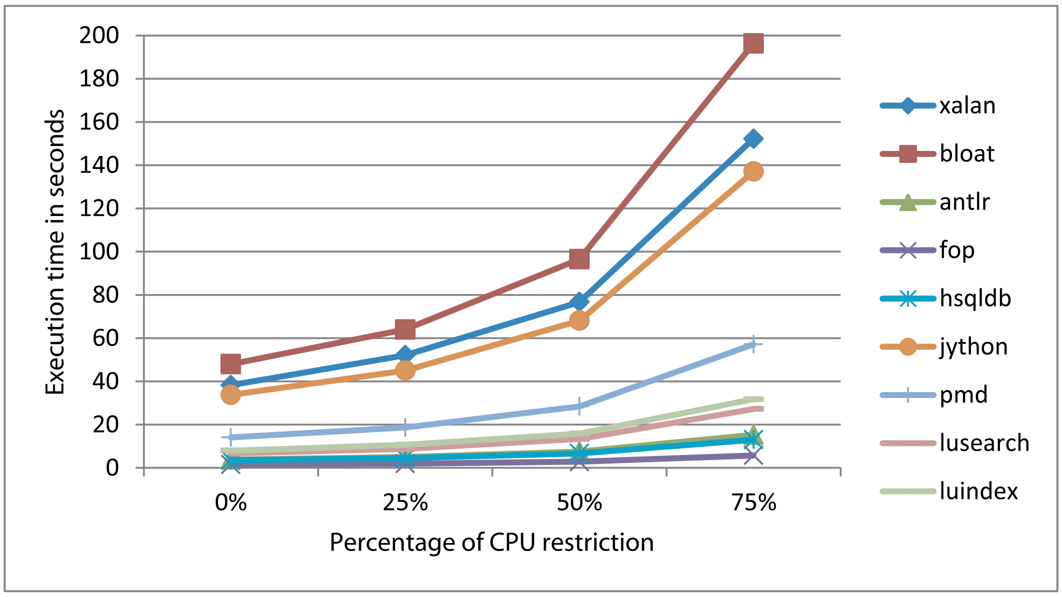


Figure 6: Effects of restraining CPU by 25%, 50% and 75%

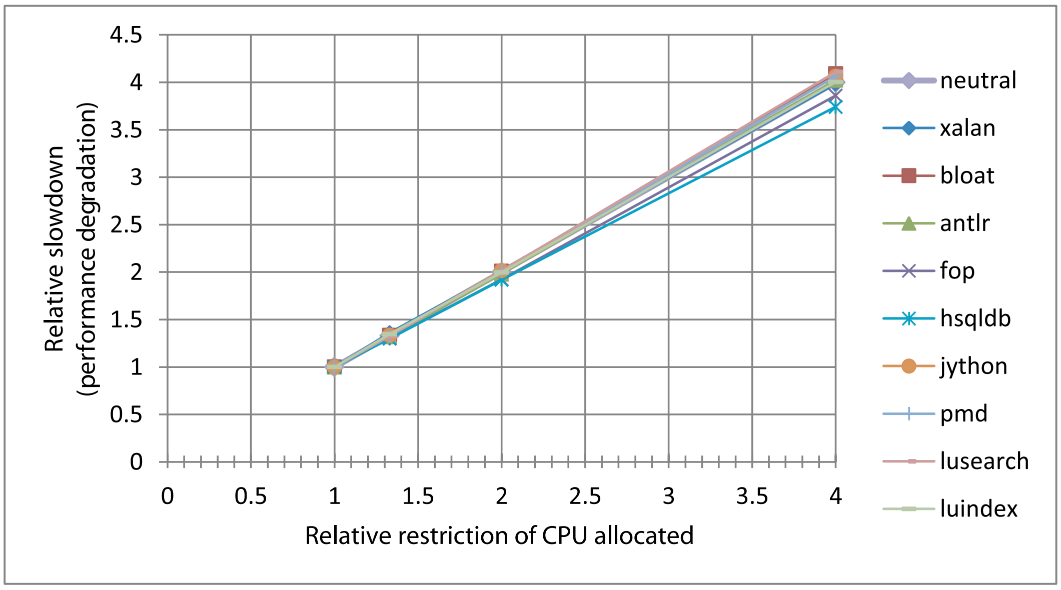


Figure 7: Relative slowdown

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 patterns, adjusting resource caps (with a tolerance factor) based on this analysis.

Ginko [17] 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 have 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 [4]. Nevertheless, most of them are hard coded and cannot be influenced without building a new version of the VM.

In [11], Czajkowski et al. propose to enhance the resource management API of the Multitask Virtual Machine (MVM) [12], 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 [19], 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.

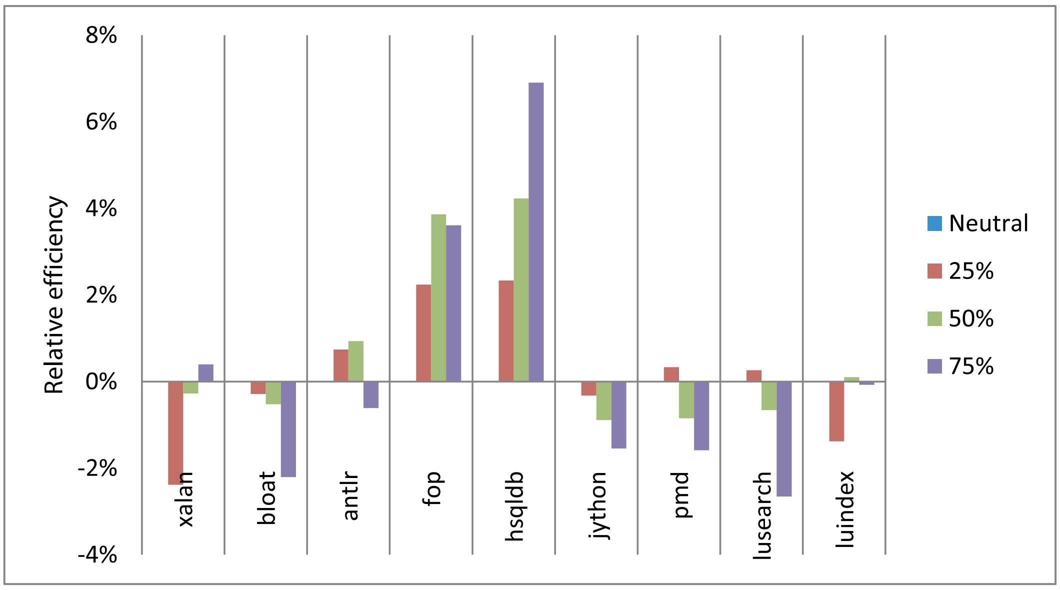


Figure 8: Relative efficiency

In [29] 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. [10] 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. [14] 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 interface for resource adaptation. 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 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. We presented the architecture of QoE-JVM which 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. Our adaptation model, based on the yield obtained from applying different strategies to each tenant's workload, aims at putting 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 heap size, and CPU allocation) 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 tenants with higher priority or to applications with more requirements. The overall goal is to improve flexibility, control and efficiency of infrastructures running long applications in clusters.

Acknowledgments. 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, PEst-OE/EEI/LA0021/2013 and the PROTEC program of the Polytechnic Institute of Lisbon (IPL).

REFERENCES

1. http://wiki.xensource.com/xenwiki/creditscheduler, visited at 31-03-2011.
2. http://www.spec.org/jvm2008/, visited 17-11-2012.
3. 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.
4. 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.
5. 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.
6. Sukadev Bhattiprolu, Eric W. Biederman, Serge Hallyn, and Daniel Lezcano. Virtual servers and checkpoint/restart in mainstream linux. *SIGOPS Oper. Syst. Rev.*, 42(5):104–113, July 2008.
7. 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.
8. 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.
9. C.W. Cobb and P.H. Douglas. A theory of production. *The American Economic Review*, 18(1):139–165, 1928.
10. 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.
11. 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.
12. 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.
13. Valéria Quadros dos Reis and Renato Cerqueira. Controlling processing usage at user level: a way to make resource sharing more flexible. *Concurrency and Computation: Practice and Experience*, 22(3):278–294, 2010.
14. 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.
15. 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.
16. 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.
17. 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.
18. Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin C. Rinard. Dynamic knobs for responsive power-aware computing. In Rajiv Gupta and Todd C. Mowry, editors, *ASPLOS*, pages 199–212. ACM, 2011.
19. Arkadiusz Janik and Krzysztof Zielinski. AAOP-based dynamically reconfigurable monitoring system. *Information & Software Technology*, 52(4r):380–396, 2010.
20. João Morais, João Nuno Silva, Paulo Ferreira, and Luís Veiga. Transparent adaptation of e-science applications for parallel and cycle-sharing infrastructures. In *Distributed Applications and Interoperable Systems*, volume 6723 of *Lecture Notes in Computer Science*, pages 292–300. Springer Berlin Heidelberg, 2011.
21. Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4:14:1–14:42, May 2009.
22. 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.
23. 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.
24. João Nuno Silva, Luís Veiga, and Paulo Ferreira. A^2HA - automatic and adaptive host allocation in utility computing for bag-of-tasks. *Journal of Internet Services and Applications*, 2(2):171–185, 2011.
25. João Nuno Silva, Luís Veiga, and Paulo Ferreira. nuboinc: Boinc extensions for community cycle sharing. In *SASO Workshops*, pages 248–253. IEEE Computer Society, 2008.
26. 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.
27. José Simão and Luís Veiga. A progress and profile-driven cloud-vm for resource-efficiency and fairness in e-science environments. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 357–362, New York, NY, USA, 2013. ACM.
28. José Simão, Tiago Garrochinho, and Luís Veiga. A checkpointing-enabled and resource-aware java virtual machine for efficient and robust e-science applications in grid environments. *Concurrency and Computation: Practice and Experience*, 24(13):1421–1442, 2012.
29. 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.
30. Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. *SIGOPS Oper. Syst. Rev.*, 41(3):275–287, March 2007.
31. Carl A. Waldspurger. Memory resource management in vmware esx server. *SIGOPS Oper. Syst. Rev.*, 36:181–194, December 2002.