Chapter 1

# A Taxonomy of Adaptive Resource Management Mechanisms in Virtual Machines: Recent Progress and Challenges

José Simão and Luís Veiga

José Simão
INESC-ID Lisboa, Instituto Superior de Engenharia de Lisboa (ISEL/IPL),
e-mail: jsimao@gsd.inesc-id.pt

Luís Veiga
INESC-ID Lisboa, Universidade de Lisboa - Instituto Superior Técnico,
e-mail: luis.veiga@inesc-id.pt

# Contents

**1 A Taxonomy of Adaptive Resource Management Mechanisms in Virtual Machines: Recent Progress and Challenges** . . . 1
José Simão and Luís Veiga
1.1 Introduction . . . 1
1.2 From Virtual Machines Fundamentals to Recent Trends . . . 3
1.2.1 Computation as a resource . . . 4
1.2.2 Memory as a resource . . . 5
1.2.3 Input/Output as a resource . . . 7
1.2.4 Research Trends . . . 8
1.3 Adaptation techniques . . . 8
1.3.1 System Virtual Machine . . . 10
1.3.2 High-Level Language Virtual Machine . . . 11
1.3.3 Summary of techniques . . . 13
1.4 The RCI taxonomy . . . 15
1.4.1 Quantitative Criteria of the RCI taxonomy . . . 17
1.4.2 Classification of techniques . . . 20
1.4.3 Aggregation of quantities . . . 23
1.4.4 Critical analysis of the taxonomy . . . 24
1.5 VM systems and their classification . . . 24
1.5.1 System Virtual Machine . . . 25
1.5.2 High-Level Language Virtual Machines . . . 29
1.6 Summary and Open Research Issues . . . 34
References . . . 35

**Abstract** Cloud infrastructures make extensive use of hypervisors (e.g. Xen, ESX), containers (e.g. LXC) and high level virtual machines (e.g. CLR, Java), broadly known as *virtual machine* (VM) technologies, to achieve workload isolation and efficient resource management. Isolation is a static mechanism that relies on hardware or operating system support to be enforced. Resource management is dynamic and VMs must self-adapt or be instructed to adapt in order to fit their guest's needs. In this chapter we review the main approaches for adaptation and monitoring in virtual machines deployments, their tradeoffs, and their main mechanisms for resource management. We frame them into an *adaptation loop* where sensors are monitored (e.g. page utilization), decisions are made (e.g. if-else rule, proportional-integral-derivative controller) and actions are performed using actuators (e.g. share page, change heap size). As is common in systems research, improvement in one property is accomplished at the expense of some other property. So, we present a taxonomy that, when applied to different solutions that use or augment virtual machines, can help visually in determining their similarities and differences. We analyze adaptability in virtual machines using three seemingly orthogonal characteristics: responsiveness (R), comprehensiveness (C) and intricateness (I). The process of classification and comparing systems is detailed and several representative state of the art systems are evaluated.

## 1.1 Introduction

Cloud computing infrastructures makes extensive use of virtualization technologies, either at the system or programming language level, providing a flexible allocation of hardware resources and applying the necessary resource scheduling to run multi-tenant datacenters [97, 24, 108]. Both system-level VMs (Sys-VM) and High-level language VMs (HLL-VM) are designed to promote isolation [87]. All these features are essential to consolidate applications into a smaller amount of physical servers, saving operational costs and reducing the carbon footprint of datacenters [18, 95, 34].

Dynamic allocation of resources use different strategies, either aiming to maximize fairness in the distribution of resources or deliberately favor a given guest based on past resource consumption and prediction on future resource demand. Among all resources, CPU [110, 39, 44] and memory [101, 61, 9] are the two for which a larger body of work can be found. Nevertheless, other resources, such as the access to I/O operations, have also been analyzed [63, 53, 40].

Most HLL-VMs have only one guest at each time - the application. As a consequence, in most cases, some resources are monitored not to be partitioned but for the runtime to adapt its algorithms to the available environment. For example, a memory outage could force some of the already compiled methods to be unloaded, freeing memory to maintain more data resident. Several systems have been proposed to control system resources usage in HLL-VMs, most of them targeting the Java runtime (e.g. [31, 19, 84, 22]). They use different approaches: from making modifications to a standard VM, or even proposing a new implementation from scratch, to modifications in the byte codes and hybrid solutions.

In each work, different compromises are made, putting more emphasis either on the portability of the solution (i.e., not requiring changes to the VM) or on the portability of the guests (i.e., not requiring changes to the application source code). In order to do so, VMs, or middlewares augmenting their services, can be framed into the well-known adaptation loop [69], where systems monitor themselves and their context, analyse the incoming values and detect significant changes, decide how to react, and act to execute such decisions. In this chapter we group these steps in three distinct phases, similarly to the adaptability loop of other works in the context of autonomic systems [7, 58]: i) monitoring, ii) decision, iii) actuation. Monitoring determines which components of the system (e.g. hardware, VM, application) are observed. Control and decision take these observations and use them in some decision strategy to decide what has to be changed. Enforcement deals with applying the decision to a given component/mechanism of the VM.

However, existing surveys of virtualization technologies (e.g. [14, 55]) tend to focus on a wide variety of approaches which sometimes results only in an extensive catalog. One of the first published surveys of research in virtual machines was presented in 1974 [38]. Goldberg's work was focused on the principles, performance and practical issues regarding the design and development of system-level virtual machines that, at the time, were developed by IBM, the Massachusetts Institute of Technology (MIT), and few others. Arnold et al. [14] focus only on HLL-VMs and particularly on the techniques that are used to control the optimizations employed by the just-in-time (JIT) compiler, taking advantage of runtime profiling information.

This chapter surveys several techniques used by virtual machines, and systems that depend on them, to make an adaptive resource management, extending previous preliminary work [74, 77]. Here we fully describe the adaptation loop of virtual machines discussing their principles, algorithms, mechanisms and techniques. We then detail a way to qualitative classify each of those according to their responsiveness, i.e. how fast it can react to changes, their comprehensiveness, i.e. the scope of the mechanisms involved, and their intricateness, i.e. the complexity of the modifications to the code base or to the underlying systems. These metrics are used to classify the mechanisms and scheduling policies. The goal is not to find the best system, as this depends on the scenario where the system is going to be used, but instead it aims to identity the tradeoffs underpinning each system.

Section 1.2 presents the architecture of high-level and system-level VMs, depicting the building blocks that are used in research concerning resource usage. Section 1.3 presents several adaptation techniques found in the literature and frames them into the adaptation loop. In Section 1.4, the classification framework is presented. For each of the resource management components of VMs, and for each of the three steps of the adaptation loop, we propose the use of a quantitative classification regarding the impact of the mechanisms used by each system. We then use this framework to classify 18 state-of-the-art systems in Section 1.5, aiming to compare and better understand the benefits and limitations of each one.

## 1.2 From Virtual Machines Fundamentals to Recent Trends

Virtualization technologies have been used since the primordials of multi-user systems. The idea of having better isolation among different users in a multi-user system was first explored by IBM [11]. In these systems, each user was assigned a *virtual machine* which executed in the context of a so called *control program* (CP).

In the last two decades this idea was extended and further explored to support the execution of commodity operating systems in each virtual machine, without losing performance. Resource isolation was further enforced so that bad behaving virtual machines cannot disrupt the service of other instances [17]. This is due not only to the software but also to new hardware support that enhances the performance of VMs running on a multi-tenant server [2].

System-level virtual machines execute under the control of a virtual machine monitor (VMM) to control the access of the guest operating system running in each virtual machine to the physical resources, virtualizing processors, memory, and I/O. Recently, operating systems extended the process-level isolation mechanisms with further virtualization of the file system, name spaces and drivers (e.g. network)[5, 6]. Furthermore, the integration of resource consumption controls, made it possible to run workloads on a new kind of execution environment, called *container*, under the same OS.

High Level Language VMs, which are highly influenced by the Smalltalk virtual machine [33], also provide a machine abstraction to their guest, which is an end-user application. The just-in-time (JIT) compiler is responsible for this translation and is, in itself, a source of adaptation driven by the dynamics in the flow of execution (e.g. hot methods are compiled using more sophisticated optimizations) [14]. Memory management has a high impact on the use of memory and CPU. After more than three decades of research work focusing on tunning garbage collection algorithms [52], recent research work is made towards the selection of application-specific algorithms and parameters, in particular, heap size and the moment of triggering memory collection [60, 47, 90].

Figures 1.1 to 1.4 depicts four types of deployments. The first is a traditional configuration where an operating system (OS) regulates the access of native applications (i.e., the ones that use the services of the OS) to the

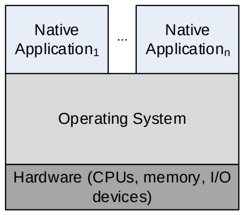

**Fig. 1.1** a. Non virtualized system

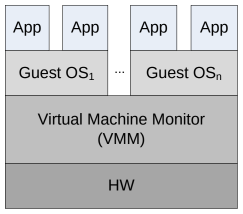

**Fig. 1.2** b. System-level VM

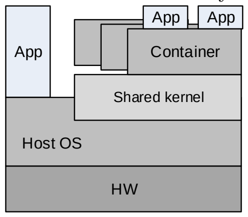

**Fig. 1.3** c. Container type VM

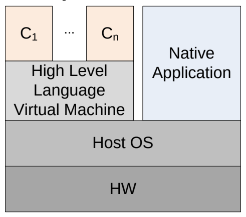

**Fig. 1.4** d. High-level language VM

hardware. The second, Figure 1.2, represents a configuration where an hypervisor, known as virtual machine monitor, takes control of the hardware, making it possible to host several system-level virtual machine on top of the same physical resources. Each virtual machine runs a possibly different operating systems instance. Figure 1.3 shows the position of containers. These execution environments share the kernel with the host OS and allow applications to run with an extra level of isolation from the remaining user-level processes. Finally, Figure 1.4 depicts the position of high-level language VMs. They are at the level of native applications but support the hosting of *managed* components which rely (almost exclusively) on the services provided by these VMs. This chapter focus on deployments 1.2 and 1.4.

The next three sections will briefly describe how fundamental resources, CPU, memory, and I/O, are virtualized by the two types of VMs. The systems presented in Section 1.5 are based on the building blocks presented here, using them to implement different adaptive resource management strategies. We conclude with a section about recent trends on the mechanism available on these two types of VMs.

### *1.2.1 Computation as a resource*

The virtualization of the CPU concerns two distinct aspects: i) the translation of instructions and; ii) the scheduling of virtual CPUs to a physical CPU. In this chapter we focus on the scheduling problem. Although an efficient binary translation is of utmost importance, and several techniques are used [87], this is done in a way that is dependent on the execution requirement of a given tenant. In Sys-VMs, the VMM must decide the mapping between the real CPUs and each running VM [17, 26]. In the case of HLL-VMs, because they rely on the underlying OS to schedule their threads of execution. In spite of this portability aspect, the specification of HLL VMs is supported by a memory model [59] making it possible to reason about the program behavior.

The VMM scheduler, where each guest VM is assigned to one or more virtual CPUs (VCPU), has different requirements from the schedulers used in operating systems [93]. Typically, the OS uses a priority-based approach which is different from the family of schedulers used by the VMM. The VM monitor scheduling is ruled by a *Proportional Share* assigned to each VM of the system, based on its *share* (or *weight*) [91, 26].

Cherkasova et al. [26] further classifies schedulers as: i) work conservative or non-work conservative and; ii) preemptive or non-preemptive. Work conservative schedulers take the *share* as a minimum allocation of CPU to the VM. If there are available CPUs, VCPUs will be assigned to them, regardless the VM's share. In non-work conservative, even if there are available CPUs, VCPUs will not be assigned above a given previously defined value (known as *cap* or *cpu limit*). A preemptive scheduler can interrupt running VCPUs if a ready to run VCPU has a higher priority.

In Section 1.5 we present different systems that dynamically change the scheduler's parameters to give guest VMs the capacity that best fits their needs.

### 1.2.2 Memory as a resource

The design of memory management system is inherently complex, regardless of the target environment. Virtual machines (VMs) are no exception and they add an extra level to the system stack.

As pointed out by Smith et al. [87], the VMM extra level of indirection generalizes the virtual memory mechanisms of operating systems. To maintain isolation, the guest OS continues to see a real address (i.e., machine address) but this address can in fact change during the activation of the VM. So, the VMM must establish a *virtual* $\rightarrow$ *real* $\rightarrow$ *physical* mapping for each guest OS and VM.

When an OS kernel, running on an active VM, uses a *real* address to perform an operation (e.g., I/O), the VMM must intercept this address and change it to the correspondent physical one. On the other hand, user level applications use a *virtual* address to accomplish their operations. To avoid a two-fold conversion, the VMM keeps *shadow pages* for each process running on each VM, mapping *virtual* $\rightarrow$ *physical* addresses. Access to the page table pointer is virtualized by the VMM, trapping read or write attempts and returning the corresponding table pointer of the running VM. The translation look-aside buffer (TLB) continues to play its accelerating role because it will still keep in cache the *virtual* $\rightarrow$ *physical* addresses.

To effectively manage the allocation of physical memory, the VMM must re-assign pages between VMs. The decision about which specific pages are to be relinquished is actually made by the guest OS running on the VM that is selected by the VMM to give away memory. This is done by interacting with a kernel driver at the OS, known as the *balloon* driver [17, 101].

The balloon driver is controlled by memory management policies which will be introduced in Section 1.3. When the balloon is instructed to inflate it will make the guest OS swap memory to secondary storage. When the balloon is instructed to deflate, the guest OS can use more physical pages, reducing the need to swap memory. Another issue related to memory management in the VMM is the sharing of machine pages between different VMs. If these pages have code or read-only data they can be shared avoiding redundant copies.

The goal of memory's virtualization in high language VMs is to free the application from explicit dealing with memory deallocation, giving the perception of an unlimited address space. This avoids keeping track of references to data structures (i.e. objects), promoting easier extensibility of functionalities because the bookkeeping code that must be written in non-virtualized environment is no longer needed [107, 87].

Different strategies have been researched and used during the last decades. Simple *mark and sweep*, *compacting* or *copying collectors*, all identify live objects starting from a root set (i.e., the initial set of references from which live objects can be found, containing thread stacks and globals). All these approaches strive for a balance between the time the program needs to stop and the frequency the collecting process needs to execute. This is mostly influenced by the heap dimension and, in practice, some kind of nursery space is used to avoid searching all the heap.

As parallel hardware becomes ubiquitous and GC pause time reduction becomes essential, the stop-the-world approach has been questioned, resulting in the design of concurrent and incremental collectors [28, 96]. However, recent studies show that the base approach has no fundamental scalability problem [35] and that the GC impact can be diminished with parallel techniques, which still need to stop the program, but that explore the root set in parallel.

Researchers have analyzed garbage collection performance and found it to be application-dependent [89] and even input-dependent [60, 94]. Based on these observations, several adaptation strategies have been proposed [14], ranging from parameters adjustments (e.g. the current size of the managed heap [42, 84]) to changing the algorithm itself before the first execution [86] or at runtime [89].

### *1.2.3 Input/Output as a resource*

In both types of VMs, virtualization of input/output deals with the emulation, accounting and constraining of using available physical devices. In spite of these similar goals, virtualization occurs with different impacts. In a VMM, the access to device drivers can be para-virtualized or full virtualized. In the first scenario, a cooperative guest OS is expected to call a virtual API in the VMM [17]. In the second scenario (a full virtualized environment) the VMM can either intercept the I/O operation, at the device driver or system call level [87]. The typical option is to virtualize at the device driver level, installing virtual device drivers at each guest, which, from the guest operating system standpoint, are regular drivers.

The main challenge in I/O virtualization for fully virtualized systems, such as the ESX [101] or the KVM [56] hypervisors, is to avoid the extra context switches between the guest and the host to handle interrupts generated by I/O devices [8, 40]. The interrupts are, by nature, asynchronous and sent to the CPU to signal the completion of I/O operations. So, the overhead comes from the extra CPU cycles necessary to exit the guest, run the host interrupt handler and inject the virtual interrupt in the guest.

The performance of I/O-intensive applications in a virtualized environment is also affected by the CPU scheduling and memory sharing mechanisms [25, 68, 63, 26]. The CPU scheduling strategy of each physical core to the virtual cores has impact in the I/O performance of the applications running on top of virtual machines. A detailed analysis of the scheduler's impact on VM's performance is available in the literature [63, 26]. The main observations were related to the domain driver's preemption during the dispatch of multiple network events and the order of VMs in the run queue.

High-level language VMs rely on the operating system API to accomplish input/output operations as disk and network read and writes. Depending on the address space isolation supported by the VM, accounting and regulation have different levels of granularity. In a classic JVM implementation, accountability can be done globally at the VM or on a per-thread basis [92]. In HLL-VMs supporting the abstraction of different address spaces (e.g. isolates in Multi-task VM [31], application domains in the Common Language Runtime) accounting is made independently for each of these spaces.

In summary, although the interaction with I/O devices has a major role in the design of virtual machines, the sub-systems responsible for this task do not have to make regular scheduling or allocation decisions. So, this chapter will not focus on these works, but on adaptive techniques related to the virtualization of CPU and memory (which indirectly contribute to the performance of I/O-intensive applications).

### *1.2.4 Research Trends*

The ACM library [1] shows that articles with the terms “VM” and “Virtual Machine” continues to increase. Extrapolating the total number of publication up to 2016 to the end of the decade, the number will more than double the results of the previous decade, the 2000s. Because of their strategic role in cloud deployments they will certainly continue to be analysed and enhanced.

Regarding Sys-VMs, major research efforts continue to be done in memory virtualization techniques. For example, Amit et al. proposes VSwapper [12], which substitutes the classic balloon driver in the common case of uncooperative guests. Although this situation is known for its poor performance, VSwapper uses a combination of intricate techniques to overcome the problem, monitoring host disk blocks and establishing a relation to guest memory pages in order to detect page writes and reads that hinder performance.

When looking to HLL-VMs, research in resource management is currently driven by the need to incorporate further mechanisms to regulate memory usage when running manage runtime in clusters. Although this has been a topic of research for more than a decade now [29], new challenges were introduced by cloud deployments, namely, the execution on top of Sys-VMs and big-data applications.

Manage runtimes are the basis of modern processing and storage framework widely used by cloud-enabled application. However, because many times they execute on top of Sys-VMs, there is the need to externally instruct the HLL-VM to relinquish some memory so that the VMM can deliver it to other tenants [70, 49].

Considering a single node running instance, some improvements for big-data workloads are also being explored to avoid the problems introduced by object churn and very large heap sizes [37], including in NUMA-based architectures [36]. But because typically the workloads run on top of multiple physical nodes, researchers are looking for ways to coordinate resource management, in particular GC operations [75, 57].

## 1.3 Adaptation techniques

In a software system, adaptation is regulated by monitoring, analyzing, deciding and acting [69]. Monitoring is fed by sensors and actions are accomplished by actuators, forming a process known as the *adaptation loop*, as depicted in Figure 1.5. Virtual machines, regardless of their type, are no exception. The two intermediate phases, Analysis and Decision, are in many cases seen as one [58]. An example is the Observe, Decide and Act loop proposed by IBM for autonomic systems [7]. This chapter follows the same approach and resumes the adaptation loop to three major phases: monitor, analysis/decision, action.

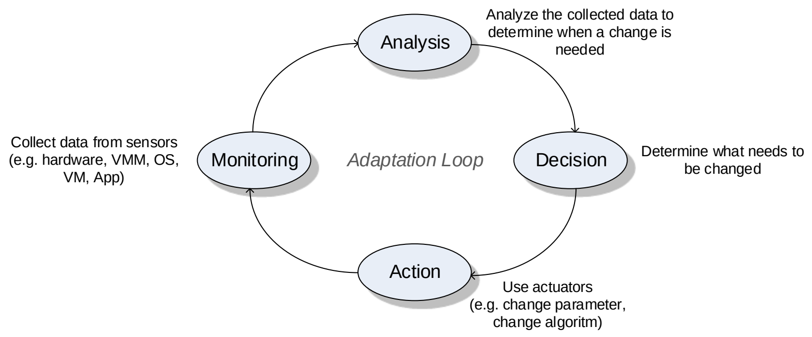
Analysis
Analyze the collected data to determine when a change is needed
Collect data from sensors (e.g. hardware, VMM, OS, VM, App)
Monitoring
Adaptation Loop
Decision
Determine what needs to be changed
Action
Use actuators (e.g. change parameter, change algoritm)

**Fig. 1.5** Adaptation loop

In a broad sense, virtual machines have an important property of autonomic systems which is self-optimization [7]. An example are the adaptive JIT compilation techniques of HLL-VMs [14] or GC algorithms that use feedback-directed online techniques to avoid page faults [41]. Furthermore, virtual machines export adaptability mechanisms that are used by outside decision systems to reconfigure VM's parameters or algorithms.

There is a broad range of strategies regarding the analysis and decision processes. Many solutions that augment system VMs use control theory elements, such as the proportional-integral-derivative controller, and Additive-Increase/Multiplicative-Decrease (AIMD) rules, to regulate one or more VM's parameters. Typically, when the analysis and decision are done in the critical execution path (e.g. scheduling, JIT, GC), the choice must be done as fast as possible, and so, a simpler logic is used.

In our previous work, we have addressed adaptation with strategies based on economic models and awareness of the workloads. Regarding System VMs, we have addressed adaptation of VM allocation [73] and resizing mechanisms [80, 82]. Regarding High-Level Language VMs (Java VM), we have studied the economics of enforcing resource (CPU and memory) throttling [79], taking into account application performance [81], and the tradeoffs between resource savings and performance degradation/improvement, when aggressively transferring resources among applications [78]. At the middleware level, federating several VMs, adaptation concerns memory management in object caching/replication aggressiveness [98], driven by declarative policies [99], and adapting the number of VMs/nodes dynamically allocated to multi-threaded Java applications, running on top of multi-tenant clustered runtimes [76].

Next we will present and discuss the state of the art regarding the three major steps of the adaptation loop for each type of VM and their internal resource management mechanisms.

### *1.3.1 System Virtual Machine*

The VMM has built-in parameters to regulate how resources are shared by their different guests. These parameters regulate the allocation of resources to each VM and can be adapted at runtime to improve the behavior of applications given a specific workload. The adaptation process can be internal, driven by profiling made exclusively inside of the VMM, or external, which depends on application's events such as the number of pending requests. In this section, the two major VMM subsystems, CPU scheduling and Memory Manager, will be framed into the adaptation processes - monitoring, decision, and acting.

#### CPU Management

CPU management relates to activities that can be done exclusively inside the hypervisor or both inside and outside. An example of an exclusively inside activity is the CPU scheduling algorithm. To enforce the weight assigned to each VM, the hypervisor has to monitor the time of CPU assigned to each VCPUs of a VM, decide which VCPU will run next, and assign it to a CPU [71, 26]. An example of an inside and outside management strategy is the one employed by systems that monitor events outside the hypervisor (e.g. operating systems load queue, application level events) but then use its internal actuators to adjust parameters. For example, monitoring the waiting time inside the spin lock synchronization primitive (in the kernel of the guest operating system) may be necessary to inform the hypervisor's scheduler about the best co-scheduling [65] decisions of VCPUs [104].

Decision strategies can be simple, like the proportional share-based that enforces predefined *shares* defined by high level policies in a multi-tenant environment. More complex decisions, made outside the hypervisor, include: i) control theory using a PID controller [110, 67], ii) linear optimization [66], iii) signal processing and statistical learning algorithms [39].

The actions taken by the CPU scheduler inside the hypervisor include: i) number of VCPUs [71], ii) co-scheduling [100, 104, 105], iii) VCPU migration [100], iv) number of threads and sleep time [110]. Systems where decisions are made outside the hypervisor use the available actuators, namely: i) VCPUs share, ii) VCPUs *cap* [39, 66, 46].

#### Memory Management

In this step of the control loop, the VMM needs to determine how pages (or parts of it) are being used by each VM. To do so, it must collect information regarding: i) page utilization [101, 103, 62], ii) page (and sub-page) contents equality or similarity [101, 43]. Some systems also propose to monitor application performance, either by instrumentation or external monitoring, in order to collect information closer to the application's semantics [49, 70].

The VMM supports overcommit, that is, the total memory configured to the overall VMs can be higher than the one that is physically available. When pages of memory need to be transfered between VMs (and their guest OS) different types of decisions are made based on: i) shares [101] ii) feedback control [46], iii) LRU reference histogram [103], iv) linear programming [49].

To change the system state regarding its memory use there three main approaches: i) page sharing, ii) page transfer between VMs, iii) compress page contents. While page sharing and transfer relies on intrinsic mechanisms of the VMM, as presented in Section 1.2.2, page compression is an extension to these base mechanisms.

### *1.3.2 High-Level Language Virtual Machine*

In this section, the three major language VM subsystems, JIT compiler, GC and Resource manager, will be framed into the adaptation processes. HLL-VMs monitor events inside their runtime services or in the underlying platform. As always, there is a trade off between deciding fast but poorly, or deciding well (or even optimally), but spending too much resources and time in the process of doing so. Most systems base their decision on an heuristic, that is, some kind of adjustment function or criterion that, although it cannot be fully formally reasoned about, it still gives good results when properly used. Nevertheless, some do have a mathematical model guiding their behavior [94]. Next we will analyze the most common strategies.

#### 1.3.2.1 Just in time compilation

The JIT is mostly self-contained in the sense that the monitoring process (also known as profiling in this context) collects data only inside the VM. Modern JIT compilers are consumers of a significant amount of data collected during the compilation and execution of code.[1] Hot methods information is acquired using: i) sampling, ii) instrumentation. In the first case, the execution stacks are periodically observed to determine hot methods. In the second case, method code is instrumented so that its execution will fill the appropriate runtime profiling structures. Sampling is known to be more efficient [14] despite its partial view of events.

To determine which methods should be compiled or further optimized, there are two distinct group of techniques: i) counter-based, ii) model-based.

[1] The adaptive optimization system (AOS) in Jikes RVM [10] produces a log with approximately $700 Kbytes$ of information regarding call graphs, edge counters and compilation advices when running and JIT compiling 'bloat', one of DaCapo's benchmarks [20]

Counter-based systems look at different counters (e.g. method entry, loop execution) to determine if a method should be further optimized. The threshold values are typically found by experimenting with different programs [14]. In a model-driven system, optimization decisions are made based on a mathematical model which can be reasoned about. Examples include a cost-benefit model where the recompilation cost is weighted against further execution with the current optimization level [10, 54].

Adaptability techniques in the JIT compiler are used to produce native optimized code while minimizing impact in application's execution time overhead. Because native takes more memory than intermediate representations, some early VMs discarded native code compilations when memory became scarce. With the growth of hardware capacity this technique is less used. Thus, the actions that can complete the adaptation loop are: *i)* partial or total method recompilation, *ii)* inlining, or *iii)* deoptimization.

#### 1.3.2.2 Garbage collection

Although the way garbage collection is made usually does not change during program's execution, managed runtimes incorporate some form of memory adaptation strategy [14]. In the literature, several sensors are used to guide the decision process, both from the managed runtime and operating system, including: i) memory structures dimensions (e.g. heap in used) [85, 86], ii) GC statistics (e.g. GC load, GC frequency) [89], iii) relevant events in the operating systems (e.g. page faults, allocation stalls) [41, 48], iv) working set size [109].

Improvements to overall system performance is made reducing time spent in GC operations. Heap-related structures are adapted both before and during program execution. Adjusting before program execution is made after a previous analysis of several executions, varying relevant parameters. While there are some mathematical models of objects' lifetimes, they are essentially used to explain the GC behavior and not to drive a decision process [16]. The techniques used in the decision phase range from heuristics to more formal processes: i) simple heuristics, ii) machine learning, iii) PID controller, iv) microeconomic theories such as the elasticity of demand curves.

Actions regarding GC adaptability range from simply triggering the GC in a specific situation to the hot-swap of the algorithm itself (e.g. to avoid memory exhaustion [89]), as described next: i) GC parameters [86], ii) heap size [85], iii) GC algorithm [89].

#### 1.3.2.3 Resource management

Monitoring resources, that is, collecting usage or consumption information about different kinds of resources at runtime (e.g. state of threads, loaded classes) can be done through: i) a service exposed by the runtime [15, 31], ii) byte code instrumentation [51]. In the former, it is possible to collect more information, both from a quantitative and qualitative perspective. A well-known example is the Java Management Extensions (JMX) [64]. Because HLL-VMs do not necessarily expose this kind of service, instrumentation allows some accounting in a portable way. Accounted resources usually include CPU usage, allocated memory and relevant system objects such as threads or files.

This subsystem has to decide whether a given action (e.g. consumption) over a resource can be done or not. This is accomplished with a policy, which can be classified as: *i)* internal or *ii)* external. In an internal policy, the reasoning is hard-coded in the runtime, possibly only giving the chance to vary a parameter (e.g. number of allowed opened files). An external policy is defined outside the scope of the runtime, and thus, it can change for each execution or even during execution.

This subsystem is particularly important in VMs that support several independent processes running in a single instance of the runtime. Research and commercial systems apply resource management actions to: i) limit resource usage, ii) perform resource reservation. Limiting resource usage aims to avoid denial of service, or to ensure that the (possibly payed) resource quota is not overused. The last scenario is less explored in the literature [31]. Resource reservation ensures that, when multiple processes are running in the same runtime, it is possible to ensure a minimum amount of resources to a given process.

### *1.3.3 Summary of techniques*

In this section we summarize several techniques identified in the literature. Figure 1.6 presents the techniques used in the adaptation loop of Sys-VMs. They are grouped by the two major adaptation targets, CPU and memory, and then into the three major phases of the adaptation loop. The CPU management sub-tree is the one that has more elements (i.e., more adaptation techniques). This reflects the emphasis given by researchers to this component of Sys-VMs. Regarding memory, early work of Waldspurger [101] and Barham et al. in [17] laid solid techniques for virtualizing and managing this resource. Recent work shows that, to improve perform of workloads regarding their use of memory, it is crucial to have more application-level information [103, 49].

Figure 1.7 presents the techniques used in the adaptation loop of systems using HLL-VMs. They are grouped into the three major adaptation targets: i) JIT compiler, ii) garbage collection, iii) resource management. Each adaptation target is then divided into the three phases of the adaptation loop. The garbage collection sub-tree has a higher number of elements when compared

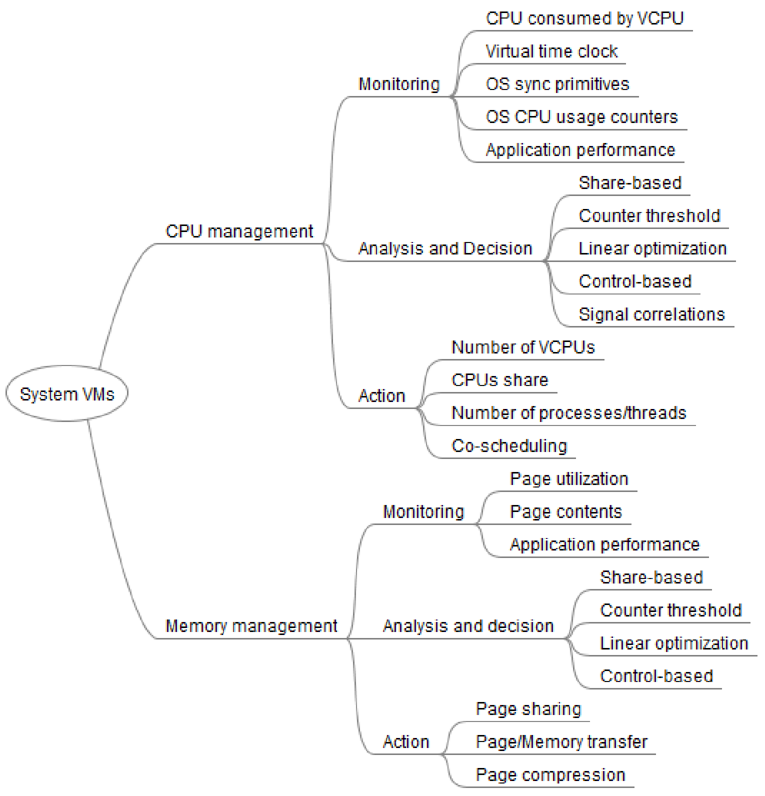

**Fig. 1.6** Techniques used by System VMs in the monitoring, decision and action phases

with any of the other two. This reflects different research paths, but also a higher dependency of the garbage collection process on the workloads and on the context of execution (i.e., shared environment, limited memory, etc.).

The techniques used in the monitoring and action phase are domain-specific. For example, there are sensors related to the utilization of memory pages or actuators that change a parameter in the garbage collection algorithm. On the contrary, the strategies used in the decision phase can be found in other adaptability works and, in general, in autonomic computing systems [7, 58].

Maggio et al. [58] have focused attention on the characterization of decision techniques. They divide them into three broad categories: heuristics, control-based, and machine learning. In fact, we can also see these categories when we look to the techniques identified in this section. Figure 1.6 and Figure 1.7 show that the decision strategies are either heuristic (e.g. microeconomics, share-based), control-based (e.g. PID controller), based on signal processing techniques (e.g. correlation of different windows of samples), and machine learning (e.g. reinforcement learning). Regarding strategies that use linear

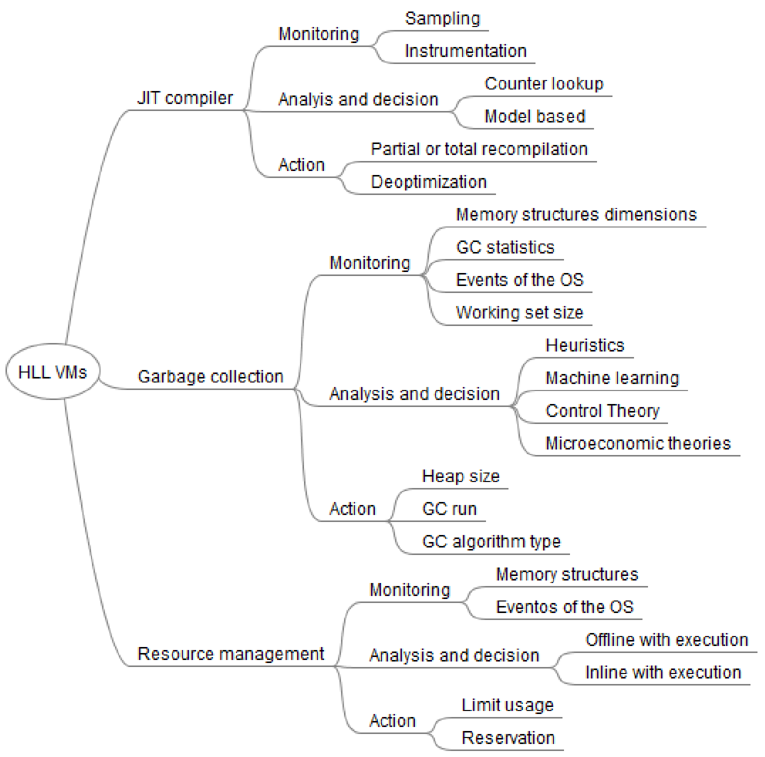

**Fig. 1.7** Techniques used by HLL-VMs in the monitoring, decision and action phases

programming, they are used only to make a general model of the scheduling variables. In practice, these approaches use integer linear programming which is known to be NP-hard. Thus, they use some kind of greedy approach to solve it.

Based on the survey of these different techniques, the next section will present a classification framework that aims to compare complete adaptive systems.

## 1.4 The RCI taxonomy

To understand and compare different adaptation processes we now introduce a framework for classification of VM adaptation techniques. The classification is based on the different techniques described earlier and depicted in Figure 1.6 and Figure 1.7. The analysis and classification of the techniques and the way they are used in each of the adaptation loops revolves around three fundamental criteria: ***R****esponsiveness*, ***C****omprehensiveness* and ***I****ntricateness*.

We call it RCI taxonomy. Our goal is to put each system in perspective and compare them regarding three criteria. The final RCI values of a given system depend on the techniques the system uses for monitoring, decision and acting.

These aspects were chosen, not only because they encompass many of the relevant goals and challenges in VM adaptability research, but also because they seem to embody a fundamental underlying tension: *to achieve improvements in two of these aspects the system must do so at the expense of the other*. System design in always a trade-off between different choices. A well known example is the CAP theorem [23], showing the tension existing in the general design of distributed systems. In the particular case of peer-to-peer systems, high availability, scalability, and support for dynamic populations are other kind of tensions [21].

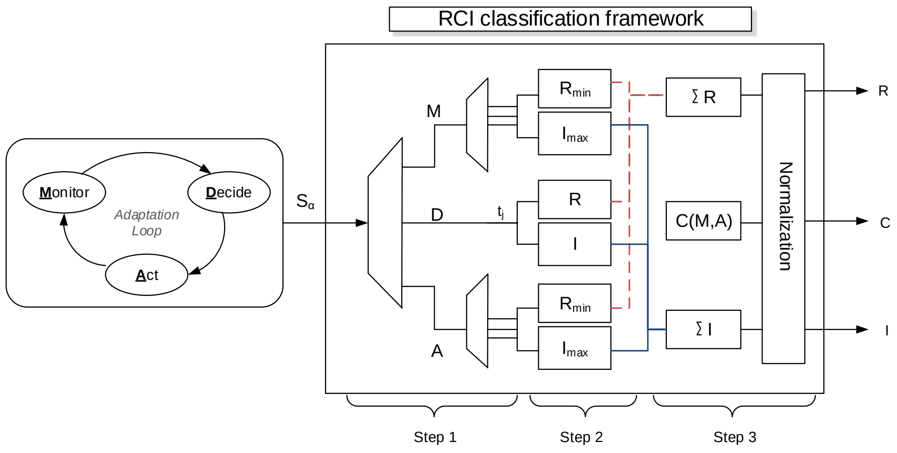

**Fig. 1.8** A step-by-step classification process

The framework starts by taking the input system and decomposes it into the adaptation techniques used in the monitoring, decision, and acting phase. This is represented in step 1 of Figure 1.8. Then, for each technique, a value for R, and I is determined (step 2). The metric C is determined in step 3 by taking into account the order of magnitude of the number of sensors and actuators. Also in step 3, the previous values are aggregated and normalized, determining the final RCI tuple for the system.

Decomposing the system into the previously mentioned parts (step 1) is simply done by analyzing the reported techniques, both in their nature and cardinality. To proceed with the classification process, the framework must determine:

i) Which quantitative value is assigned to each technique in the monitoring, decision or acting phase;

ii) How these values are aggregated to reach a final RCI tuple.

These two steps are detailed in the following sections. First, Section 1.4.1 discusses a quantitative criteria, where design options, representing groups/classes of techniques, are assigned a single value. Next, Section 1.4.2 maps the set of specific techniques presented in Section 1.3 to these classes, so that each technique is assigned a unique value of R and I. This completes step 2 of the classification process. Finally, Section 1.4.3 explains the rationale of step 3, showing how the previous values are aggregated with the C metric to determine a system's RCI.

## *1.4.1 Quantitative Criteria of the RCI taxonomy*

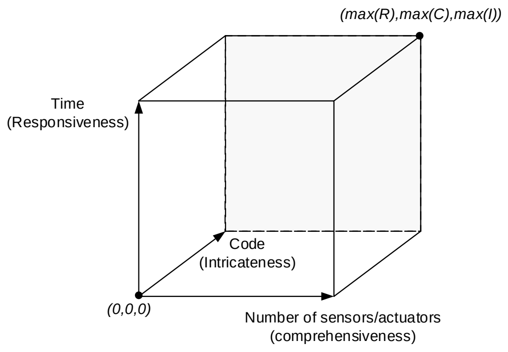

**Fig. 1.9** Systems design interval

We think the three metrics are able to capture a *design interval* as presented in Figure 1.9. They are a proxy for time, space and complexity-related characteristics. Our conjecture is that we will see systems that are away from the minimum and the maximum of the cube, that is, neither too simple (e.g., near the base of the coordinates) nor excelling in the three metrics (e.g., near or coincident with the maximum point in the design space). The following list points the exact meaning of the three criteria, regarding each of the adaptation phases. Next, we will detail how they are mapped to a numeric scale, in each phase, which will be used to determine the RCI of systems.

- **Responsiveness**. It captures time-related characteristics of the techniques. Regarding the monitoring phase, it depends on the latency of reading a value. Higher values are assigned to sensors immediately available on the VM code base, where higher values represent external sensors (operating system or application-specific). For the decision phase, responsiveness is lower in those techniques that take longer to reach a given adaptation target. Regarding the action phase, high values indicate that the effect is (almost) immediate, while a low value represents actuators that will take some time to produce effects.

- **Comprehensiveness**. It captures quantity-related characteristics of the techniques. Regarding the monitoring and deciding phases, it gets higher as the quantity of the monitored sensors increases. Regarding the acting phase, the comprehensiveness value grows with the quantity of actuators that the system can engage.
- **Intricateness**. It captures the inherent complexity of the techniques. Regarding the monitoring and acting phase, higher values are reserved for sensors/actuators that had to be added to the base code of the virtual machine, operating system, or application layer. Low values represent sensors/actuators that are already available and can be easily used. In the deciding phase, intricateness represents the inherent complexity of the deciding strategy. For example, an if-then-else rule has low intricateness but advanced control theory has higher values.

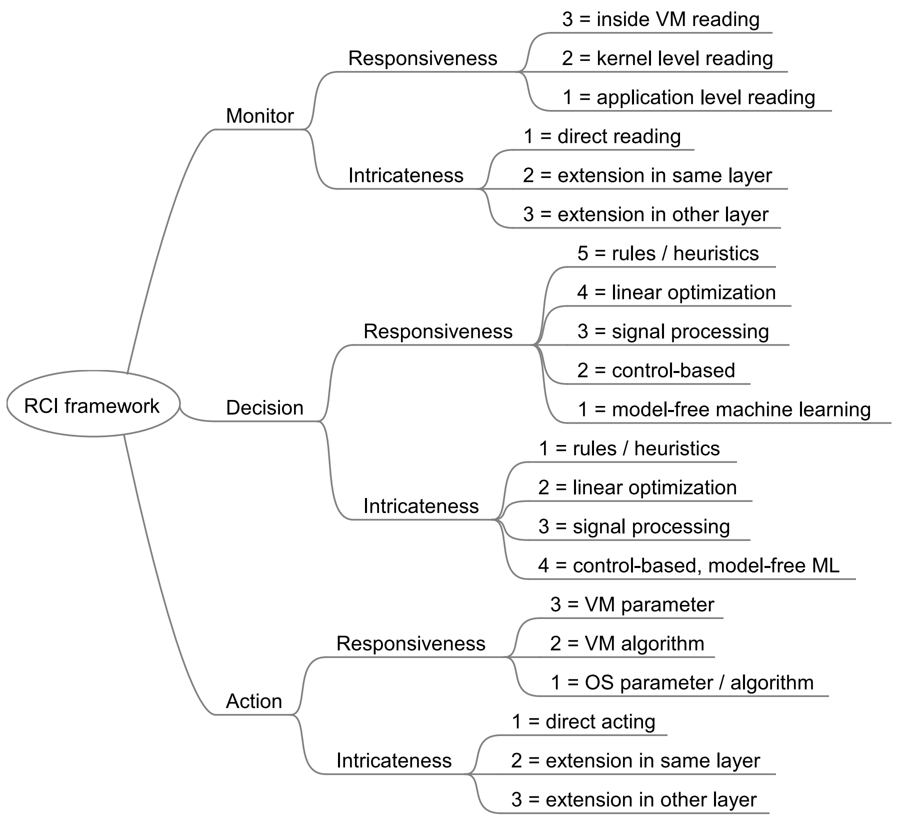

**Fig. 1.10** Quantitative values for the design options of the RCI framework

Figure 1.10 represents each of these criteria (R, C, I) for the three adaptation phases (M, D, A). For each criteria, in each adaptation phase, the figure shows several options there are used during the classification of a given technique, used in step 2 of the classification process. It does so by showing the mapping between a design option (e.g. use a sensor that is an extension inside the VM) and a quantitative value. These values establish an order among different options.

It is important so stress that these design options do not represent a specific technique but a class of techniques. For example, "direct reading" in the criterion I of the phase M, is to be selected when the sensor is available in the original code base or in another level of the system stack, without the necessity of building further extensions. This indirection makes the classification system generic because the number of techniques, sensors and actuators can grow in the future while being accommodated by the taxonomy in one of the existing classes. Even so, we think these classes are expressive and distinctive enough to characterize different levels of responsiveness, comprehensiveness, and intricateness.

The mapping between classes and specific techniques will be presented next, in Section 1.4.2. Note also that the scale of the values is not important (they typically represent different orders of magnitude) as long as the values are positive and monotonically increasing or decreasing, in accordance with the corresponding criteria.

Across all the adaptation phases, comprehensiveness is directly represented by the number of sensors or actuators, as explained previously. This is represented by $n$, which is a positive quantity (between 1 and 3) corresponding to the number of sensors or actuators that are used. This means that the comprehensiveness increases as this number grows. The other two criteria have more distinctive characterizations in each of the adaptation phases, which we elaborate next:

- **Monitoring**. The responsiveness of the monitoring phase depends on the cost of reading. The cost of reading relates to the time spent in reading a single value, that is, how fast can a single value be collected. This depends on the layer where the sensor is in relation to where the decision is made. For example, some systems use application-level monitors which require inter-process communication to read them (e.g. number of completed SQL transactions [49]). Others depend only on values collected inside the virtual machine monitor or the HLL-VM context. A middle ground approach is that of systems that depend on sensors from other layers, such as the OS, but, reading them has a low cost (e.g. the `/proc` virtual file system).
  The intricateness of the monitoring phase is a measurement of how complex is the code for reading sensors. Value 1 is assigned to systems that use preexisting sensors of the virtual machine or in the execution environment, which have a direct access. Value 2 is for extensions made inside the virtual machine and value 3 is assigned when extensions were made

in the underlying system and/or hardware (e.g. operating system, in the case of HLL-VMs).

- **Deciding**. The responsiveness and intricateness of the deciding phase is in a large part inspired by the study of Maggio et al. [58]. They discuss how feedback control mechanisms compare to each other in the context of a benchmark suite composed of multi-threaded programs, instrumented with the Application Heartbeat framework [50]. Taking into account the analyzed techniques, our classification framework is based on five decision types i) rules/heuristics; ii) linear optimization; iii) control-based solutions; iv) signal processing techniques; v) model-free machine learning solutions.
  We have classified these five types of decision strategies as decreasingly responsive, because they take an increasing amount of time to reach a certain target point. They are increasingly intricate with the exception of control-based solutions which we consider more intricate than signal processing. This is so because of the panoply of parameters that usually have to be tunned. A model-free solution has also the highest intricateness value because the tunning of assigning credits to each possible action and the balance between exploitation versus exploration (i.e. balancing between making the best decision given current information or explore more system states) [58].
- **Acting**. In this phase, responsiveness reflects the capacity of the actuator to produce an observable and measurable consequence. Any throttle to the processing capacity will have almost immediate effect and so a value of 1 is assigned to this type of actuators. Regarding memory, tweaking the set of pages assigned to a VM will have a quicker impact than simply changing its memory share. Changing heap parameters is, in comparison with the other techniques, the least responsive one, and so it gets a value of 3. Intricateness has, in this phase, a similar characterization to the one made in the monitoring phase.

In the following section, we map the previous analyzed techniques to this tree of design options.

### *1.4.2 Classification of techniques*

Based on the quantitative values of the taxonomy described in the previous section, we now focus on mapping of the techniques described in Section 1.3 to a value, so that a final RCI of each system can be determined, and different systems can be compared.

Tables 1.1-1.3 refer to system-level virtual machines and map a specific sensor, actuator or decision technique to a particular value. For each line, the first column identifies a technique (as presented in Figures 1.6 and 1.7) while the second and third columns contain a design class and the corresponding value, for responsiveness (second column) and intricateness (third column). Tables 1.4-1.6 are the ones corresponding to the high-level language virtual machines and follow the same logic.

**Table 1.1** System VMs: Sensors monitored

| Sensor | ***R*** option | Value | ***I*** option | Value |
|---|---|---|---|---|
| page utilization | inside VM | 3 | direct reading | 1 |
| page contents | inside VM | 3 | extension same layer | 2 |
| page faults | kernel | 2 | direct reading | 1 |
| memory demand | kernel | 2 | direct reading | 1 |
| application's performance | outside | 1 | direct reading | 1 |
| Virtual time clock | inside VM | 3 | direct reading | 1 |
| CPU consumed by each VCPU | inside VM | 3 | direct reading | 1 |
| Xen CPU/Mem consumed | kernel reading | 2 | direct reading | 1 |
| OS sync primitives | kernel | 2 | extension other layer | 3 |
| OS CPU usage counter | kernel | 2 | direct reading | 1 |

**Table 1.2** System VMs: Decision techniques

| Control technique | ***R*** option | Value | ***I*** option | Value |
|---|---|---|---|---|
| share based | rule / heuristic | 5 | rule / heuristic | 1 |
| counter threshold | rule / heuristic | 5 | rule / heuristic | 1 |
| integer linear programming | linear optimization | 4 | linear optimization | 2 |
| PID controller | Control-based | 2 | Control-based | 4 |
| resource usage samples correlation | signal processing | 3 | signal processing | 3 |
| LRU histogram | rule / heuristic | 5 | rule / heuristic | 1 |

**Table 1.3** System VMs: Actuators used in the action phase

| Actuator | ***R*** option | Value | ***I*** option | Value |
|---|---|---|---|---|
| page sharing | VM parameter | 3 | extension in same | 2 |
| page compression | VM algorithm | 2 | extension in same | 2 |
| page/memory transfer | VM parameter | 3 | direct acting | 1 |
| co-scheduling | VM parameter | 3 | extension in same | 2 |
| number of VCPUs assigned to CPU | VM parameter | 3 | direct acting | 1 |
| change shares or caps | VM parameter | 3 | direct acting | 1 |
| number of processes/threads | VM parameter | 3 | direct acting | 1 |

Looking at the techniques used in the monitor phase, Tables 1.1 and 1.4 show us that only two techniques have the minimum responsiveness. This is so because most of the sensors are *near* the VM execution space (either in a sub-system of the VM or in the operating system). Low intricateness also is dominant as most sensors are already available.

Regarding the decision phase, analyzed in Tables 1.2 and 1.5, a majority of techniques have high responsiveness values. As a consequence, they are less intricate. In HLL-VMs, techniques are usually either very simple or have maximum complexity.

Finally, regarding the action phase, we note that all actuators are either already available in the VM code base or are extensions to the VM code base. Contrary to sensors, no new actuators are proposed for other layers of the execution stack. This leads to not having, in practice, actuators with the maximum intricateness.

**Table 1.4** HLL VMs: Sensors monitored

| Sensor | ***R*** option | Value | ***I*** option | Value |
|---|---|---|---|---|
| Memory structures dimensions | inside | 3 | direct | 1 |
| Events of the operative system | kernel | 2 | direct | 1 |
| Working set size | kernel | 2 | extension other layer | 3 |
| GC load | inside | 3 | direct | 1 |
| Frequency of GC | inside | 3 | direct | 1 |
| Memory usage patterns | app | 3 | extension same layer | 2 |

**Table 1.5** HLL VMs: Decision techniques

| Control technique | ***R*** option | Value | ***I*** option | Value |
|---|---|---|---|---|
| if-then-rule | rule / heuristic | 5 | rule / heuristic | 1 |
| Generic condition | rule / heuristic | 5 | rule / heuristic | 1 |
| Reinforcement learning | Model-free ML | 1 | Model-free ML | 4 |
| PID controller | Control-based | 2 | Control-based | 4 |
| Elasticity (micro-economy) | rule / heuristic | 5 | rule / heuristic | 1 |

**Table 1.6** HLL VMs: Actuators used in the action phase

| Actuator | ***R*** option | Value | ***I*** option | Value |
|---|---|---|---|---|
| Heap size | VM parameter | 3 | direct | 1 |
| Run GC | VM parameter | 3 | direct | 1 |
| Change GC algorithm | VM algorithm | 2 | extension same layer | 2 |
| Limit usage | VM algorithm | 2 | extension same layer | 2 |
| Reservation | VM algorithm | 2 | extension same layer | 2 |

### *1.4.3 Aggregation of quantities*

In this section, we give the details about the implementation of the final stage of step 2 and how step 3 operates, as depicted in Figure 1.8.

Regarding the final stage of step 2, because a given system may use more than one sensor, in the monitoring phase, and more than one actuator, in the acting phase, the framework must determine how a single R and I value is determine for these two phases (i.e. $R_M$, $R_A$, $I_M$, $I_A$). Regarding responsiveness, we consider the technique with the lowest responsiveness, as presented in Equation 1.1. This was so because the monitor or the action phase will be as responsive as the least responsive technique the system uses. Regarding the intricateness metric, we use the technique with the highest value as a representative of the phase's intricateness. Finally, note that this is not an issue for the decision phase because specific systems only use one strategy.

$$R_\pi = minimum\ of\ techniques'\ responsiveness,\ where\ \pi\ \in\ \{M, A\} \quad (1.1)$$

For a given system, $S_\alpha$, the three metrics of the framework, responsiveness, comprehensiveness and intricateness, are represented by $R(S_\alpha)$, $C(S_\alpha)$, $I(S_\alpha)$, respectively. Each of these metrics depends on the specific values of the techniques used by the system. So, to determine $R(S_\alpha)$, the framework adds the responsiveness of each phase of the adaptation loop (**M**onitor, **D**ecision, **A**ction), as presented in Equation 1.2. A similar operation is done to determine the intricateness metric.

$$R(S_\alpha) = \sum_{\pi\ \in\ \{M,D,A\}} R_\pi(S_\alpha) \quad (1.2)$$

To determine comprehensiveness, $C(S_\alpha)$, the framework takes into account the number of sensors used in the monitoring phase, the number of actuators used in the acting phase, and adds them to reach a single value. This is the operation identified as $C(M, A)$ in step 3 of Figure 1.8.

As an example, consider system $S_\alpha$, which uses several hypothetical techniques for each phase of the adaptation loop. Step 1 of the framework determines that the techniques must be identified (e.g. $T_{a..f}$). Then, for each technique, a quantitative value is assigned regarding its responsiveness and intricateness for the three phases of the adaptation loop.

The last line of Table 1.7 shows the result of the *aggregation* operations used to determine, for each of the three phases, the **R** and **I** values. The aggregate function *minimum* is used for responsiveness, while the aggregate function *maximum* is used for intricateness.

Table 1.8 completes the example, showing the *arithmetic* operations necessary to determine the overall **R**, **C**, **I** values of the hypothetical system $S_\alpha$.

**Table 1.7** Example of the aggregations made in step 2 for system $S_\alpha$

| System | Monitor | R | I | Decision | R | I | Action | R | I |
|---|---|---|---|---|---|---|---|---|---|
| $S_\alpha$ | $T_a$ | 2 | 3 | $T_d$ | 2 | 3 | $T_e$ | 1 | 2 |
| | $T_b$ | 3 | 2 | | | | $T_f$ | 2 | 1 |
| | $T_c$ | 1 | 2 | | | | | | |
| | | **1** | **3** | | **2** | **3** | | **1** | **2** |

The values from the last line of Table 1.7 are the ones used to determine **R** and **I** in Table 1.8, following the equation 1.2.

**Table 1.8** Example of the arithmetic operations in step 2 for system $S_\alpha$

| System | R | C | I |
|---|---|---|---|
| Sa | 1+2+1 | #sensors+#actuators | 3+3+2 |

### *1.4.4 Critical analysis of the taxonomy*

The RCI taxonomy aims to show trade-offs in the design of adaptive systems in the context of virtual machines. Its critical point is the design options tree, presented in Figure 1.10, and the corresponding quantitative values. It can be the case that either the design options do not represent the entire design space or that the quantitative values are not correctly assigned. We tried to minimize this by designing the taxonomy after examining several systems to better understand the scope of the design space. However, we are still missing to collect the opinion of other researchers in the area while using the taxonomy, and possibly improve it based on theirs feedback.

In the next section, relevant works are analyzed regarding monitoring and adaptability in virtual machines, both at system as well as managed languages level. The RCI taxonomy is used to compare different systems and better understand how virtual machine researchers have explored the tension between responsiveness, comprehensiveness, and intricateness.

## 1.5 VM systems and their classification

In this section we start by surveying several state of the art systems, regarding system-level VMs, Section 1.5.1, and high-level language VMs, Section 1.5.2. In each case we frame the analyzed systems into the classification framework presented in Section 1.4, describing each of the techniques used, resulting in the classification and comparison of complete systems.

### *1.5.1 System Virtual Machine*

The following are succinct descriptions of system-level VMs and systems that extend them. We start by presenting a well known open-source hypervisor. A list of systems that extend this or other similar hypervisors follows. Most of them are centered either on CPU or memory. At the end of the section, Table 1.9 summarizes the techniques used in each system. This process was identified as step 1 in Figure 1.8. This is the base for determining each system's RCI.

Friendly Virtual Machines (FVM).

This VMM aims to neither overused or underused resources. The responsible for adjusting the demand of the underlying resources is delegated to each guest, resulting in a distributed adaptation system [110]. The decision phase is regulated by feedback control rules such as Additive-Increase/Multiplicative-Decrease (AIMD), typically used in network congestion avoidance [27]. A VM runs inside a hosted virtual machine, the User Mode Linux. The FVM's daemon installed at each guest controls the number of processes and threads that are effectively running at each VM. When only a single thread of execution exists, FVM will adapt the rate of execution forcing the VM to periodically sleep.

ASMan.

The Adaptive Scheduling Manager (ASMan) [104] is an extension to Xen's scheduler. It adds the capacity to co-schedule virtual CPUs (VCPU) of VMs where there are threads holding a blocking synchronization mechanisms, such as spin locks. In non-virtualized systems, threads holding spin locks are not preempted. In a virtualized system, the VCPU continues to be hold by the thread but, because the hypervisor sees the VCPU as being idle, the VCPU is taken from execution and placed on the waiting queue. Using the concept of VCPU related degree (VCRD), the ASMan system determines the degree of relationship between the VCPUs in a VM. The system dynamically determines this metric by monitoring, in each guest OS, the time spent in spin locks. The VM is then classified with a low or high VCRD if it is bellow or above a certain threshold. When the VCRD is high, the VCPUs of that VM are co-scheduled.

HPC computing.

Shao et al. [71] adapt the VCPU mapping of Xen [17] for high performance computing applications, based on runtime information collected by a monitor that must be running inside each guest's operating system. They adjust the number of VCPUs 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. The adaptation process uses an addictive increase and subtractive decrease (AISD) strategy. Shao et al. focus their work on benchmarks used to represent the common operations of high performance computing applications. It acts on number of VCPUs assigned to each VM.

Auto Control.

The Auto Control system [66] uses a control theory model to regulate resource allocation, based on multiple inputs and driving multiple outputs. Inputs include CPU and I/O usage, together with application specific metrics. It acts on the allocation of caps for CPU and disk I/O. For each application, there is an application controller which collects the application's performance metrics (e.g. application throughput or average response time) and, based on the application's performance target, determines the new requested allocation. The model is adjusted automatically and so it can adapt to different operating points and workloads.

PRESS.

PRedictive Elastic ReSource Scaling for cloud systems (PRESS) [39] tries to allocate just enough resources to avoid service level violations while minimizing resource waste. To handle both cyclic and non-cyclic workloads, PRESS tracks resource usage and predicts how resource demands will evolve in the near future. The decision phase (which includes the analysis of oberved values) uses signal processing techniques (i.e., Fast Fourier Transform and the Pearson correlation). PRESS tries to look for a similar pattern (i.e., a signature) in the resource usage history. If this fails, PRESS uses a discrete-time Markov chain. The prediction scheme is used to regulate the CPU cap of the target VM.

Overbooking and Consolidation.

Heo et al. [46] focus on monitoring memory usage (including page faults) and application performance. They show that allocating memory in such an overcommitted environment without taking also into account the CPU, results in significant service level violations. The system uses a PID controller to dynamically change the allocating of memory (using the ballon driver) and the CPU cap.

Difference engine.

Differently from other system, Gupta et al. [43] share page content at the sub-page level, using a technique named page patching, which is made by observing the difference relative to a reference page. Based on a not recently used policy, Difference Engine also uses memory compression for pages that are not significantly similar to other pages in memory. Both techniques extends the more traditional mechanisms of copy-on-write full page sharing, already present at the Xen VMM.

VMMB.

In [62], Min et al. presents VMMB, a Virtual Machine Memory Balancer for Unmodified Operating Systems. VMMB monitors the memory demand (i.e. nested page faults and to guest swapping) re-allocates memory based on the QoS requirement of each VM. It uses the LRU histogram as input for their decision algorithm that determines the memory allocation size of each VM while globally minimizing the page miss ratio. Similarly to other works, they use balloon driver to enforce each VM's new memory size. When this is not enough, a VMM-level swapping is used to select a set of victim pages and immediately allocate memory to a selected VM.

#### 1.5.1.1 Overall systems analysis

Table 1.9 summarizes the systems analyzed in this section. After the system name, the second column identifies the dominant resource, that is, the resource over which the system is monitoring but also acting. From the third to the fourth column, we present the techniques used in each of the adaptation phases. The last column allows us to quickly determine if the system proposes extensions to the code base of the VM or not.

Figure 1.11 depicts the overall RCI of each system that uses or augments a system-level VM. It presents a visual, quantitative and comparative analysis, which completes Table 1.9. Overall, systems tend to favor responsiveness design options (as this metric prevails in every system).

When looking for memory-dominant systems (Difference engine, VMMB, Oberbooking, Ginko) we see that Overbooking is less responsive because it tries to embrace a large number of sensors and actuators. In the CPU-

**Table 1.9** Sys-VM Systems

| System | Dominant Resource | Monitor | Decision | Action | Modified VMM/VM |
|---|---|---|---|---|---|
| FVM | CPU | VTC | PID Controller AIMD | Number of threads, periodic sleep | Yes |
| AutoControl | CPU, I/O | CPU, I/O usage, Average response time | Model Predictive, Quadratic solver | cap, disk share | No |
| Press | CPU | CPU, Mem, I/O usage | Pearson correlation | CPU cap | No |
| HPC | CPU | VCPU utilization rate, System Parallel level | Rules with AISD | Number of VCPUs | No |
| ASMan | CPU | Spin locks utilization and waiting time | Thresholding rules | co-scheduling | Yes |
| Ginko | Mem | Average time per URL request, #SQL transactions, response time | Linear programming | Balloon | No |
| Overbooking | CPU, Mem | CPU, Mem, Average time per URL request | PID Controller | CPU Cap, Balloon | No |
| VMMB | Mem | Page faults, swap operations | LRU histogram | Balloon, VMM swapping | Yes |
| Difference Engine | Mem | (sub-)Page contents | Not Recently Used | Page sharing, Patching, Compression | Yes |

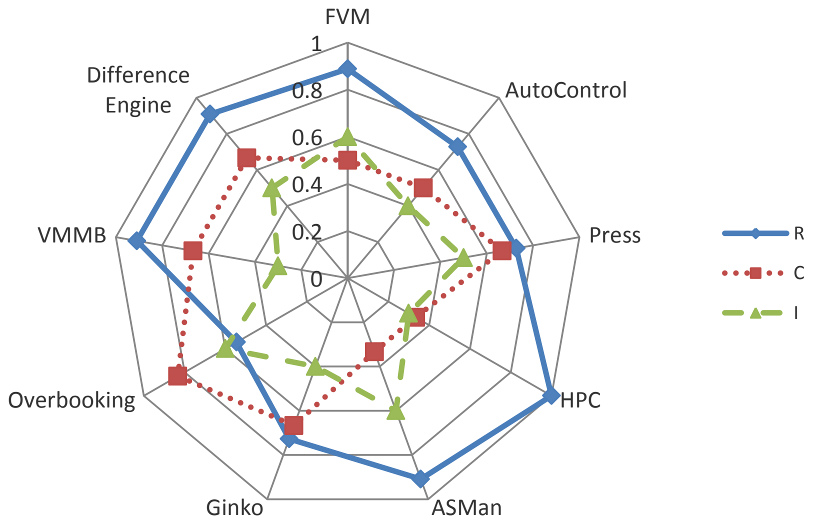

**Fig. 1.11** RCI of Sys-VMs

dominated systems, HPC is the one classified as the most responsiveness but uses simpler techniques (low intricateness) and a minimum number of sensors and actuators. ASMan is more intricate, basically because it needs extensions for the monitoring and action phase, but it had to give up on some responsiveness.

### *1.5.2 High-Level Language Virtual Machines*

This section will present and discuss different systems that monitor resource usage, resulting in either imposing limitations or changing the policies of the JIT, GC, or resource manager sub-systems. Adaptation in high-language virtual machines is made by changing their building block parameters (e.g. JIT level of optimization, GC heap size) or the actual algorithm used to perform certain operations. This section starts by presenting classic work on Java Virtual Machines (JVMs) whose goal was to incorporate resource usage constraints on regular VMs. It then surveys more recent systems where the focus was to diminish the impact of GC in program execution. At the end of the section, Table 1.10 summarizes the techniques used in each system. As in the case of system-level VMs, this process is the implementation of step 1 in Figure 1.8, which is the base for determining each system RCI.

KaffeOS.

Built on top of Kaffe virtual machine [15], KaffeOS [15] provides the ability to run Java applications isolated from each other and also to limit their resource consumption. KaffeOS, adds a process model to Java that allows a JVM to run multiple untrusted programs safely. The runtime system is able to account for and control all of the CPU and memory resources consumed on behalf of any process. Consumption of individual processes can be separately accounted for, because the allocation and garbage collection activities of different processes are separated. To account for memory, KaffeOS uses a hierarchical structure where each process is assigned a hard and a soft limit. Hard limits relate to reserved memory. Soft limits acts as guard limit not assuring that the process can effectively use that memory. Children tasks can have, globally, a soft limit bigger than their parent but only some of them will be able to reach that limit.

JRES.

The work of Czajkowski et al. [30] uses native code, library rewriting, and byte code transformations to account and control resource usage. JRES was the first work to specify an interface to account for heap memory, CPU time, and network consumed by individual threads or groups of threads. The proposed interface allows for the registration of callbacks, used when resource consumption exceeds some limits and when new threads are created.

Multitask Virtual Machine (MVM).

The MVM [31] extends the Sun Hotspot JVM to support *isolates* and resource management. *Isolates* are similar to processes in KaffeOS. The distinguishing difference of MVM is in its generic Resource Management (RM) API, which uses three abstractions: resource attributes, resource domain, and dispenser. Each resource is characterized by a set of attributes (e.g. memory granularity of consumption, reservable, disposable). In [31] the MVM is able to manage the number of open sockets, the amount of data sent over the network, the CPU usage and heap memory size. When the code running on an isolate wants to consume a resource, it will use a library (e.g. send data to the network) or runtime service (e.g. memory allocation). In these places, the resource domain to which the isolate is bound will be retrieved. Then, a call to the dispenser of the resource is made, which will interrogate all registered user-defined policies to know if the operation can continue. A dispenser controls the quantity of a resource available to resource domains.

Isla Vista.

Grzegorczyk et al. [41] takes into account a phenomenon known as *allocation stalls*, which happens during memory allocation when the system has only a few free pages. If this is so, one or more resident pages must be evicted to disk before any new page can be assigned to the requesting process. *Isla vista* implements an algorithm inspired by the exponential backoff model for TCP congestion control to avoid the stall, where transmission rate relates to heap size, and packet loss relates to page faults. Doing so, the heap size increases linearly when there are no allocation stalls. Otherwise, the heap shrinks and the growth factor for successive heap growth decisions is reduced. This is an heuristic to balance between inevitable paging operations and time spent in GC operations.

GC switch.

Soman et al. [89] is adds to the memory management system the capacity of changing the GC algorithm during program execution. The system monitors application behavior (i.e. GC load versus the time used by application's threads), and resource availability, in order to decide when to dynamically switch the GC strategy. Their decision in based on heuristics so that when the GC load is *high*, they switch from a Semi-Space (which performs better when more memory is available) to a Generational Mark-Sweep collector (which performs better when memory is more constrained).

Paging-aware GC.

Hertz et al. [48] developed a GC triggering system that takes into account the overall state of the system where the VM is running and not its single process. Two approaches where considered. VMs use a whiteboard area to know if a GC is taking place in the system. If so, they deffer their collection to avoid clustering the environment with simultaneous collection. The other is called *selfish* and the VM only takes in consideration the heap size and page faults. Based on simple heuristics like the difference in sizes of the resident set and the evolution of page faults, the GC is triggered.

CRAMM [109], on the other hand, dynamically builds the working set size (WSS) as the application progresses, monitoring minor and major page faults. It then acts on the heap size to improve application performance. The system extends the virtual memory manager of the operating system so that the WSS is dynamically built as the application progresses, monitoring minor and major page faults. After each heap collection, the system requests a WSS estimative to the virtual memory manager. It then considers this value to resize the heap. After each GC run, the histogram is also reset since the new heap size will produce a new reference histogram pattern.

GC economics.

In [84], Singer et al. relates the heap size and number of garbage collections with the price and demand law of micro-economics - with bigger heaps there will be less collections. Their decision strategy is an heuristic based on the concepts of *memory elasticity* to find a tradeoff between heap size and execution time, driving by a user-supplied elasticity target. Actions are made over the heap size, to shrink or keep.

Control Theory.

Heap sizing was also researched as a control theory problem[106]. In White's et al. work, a PID controller is used where the control variable is the heap resize ratio and the measurement variable is the GC overhead. To determine the new heap size, the controller, after each collection cycle, measures the error between the current GC overhead and the target GC overhead, specified by the user. The goal is to achieve and maintain the user-defined target GC overhead. The controller's parameters, such as the gain and the oscillatory period, were manually fine-tuned for a set of benchmarks. They have only tested their system under a full-heap collector.

Machine Learning for Memory Management.

Machine learning techniques have been used to dynamically learn which is the best moment to garbage collect [13] and to choose, a-priori, the best GC configuration (algorithm, serial, parallel) [83, 85] given an profile run of the application. In the first case, a reinforcement learning algorithm is used. A binary action is to be taken in each step leading to the decision to run the GC or not. The reinforcement learning algorithm accumulates penalties based on its decisions and, as time passes, it *learns* which are the best situations to run the GC. In the second group of papers, an offline machine learning algorithm, based on decision trees, is used to generate a classifier that, given a profile run of a *new* program (i.e., not used to build the model), can predict a GC algorithm that minimizes the execution time.

#### 1.5.2.1 Overall systems analysis

Table 1.10 summarizes the system analyzed in this section. The majority of them are focused on the management of the heap size and use simple heuristics to guide this process. Exception are the ones using a PID controller [106] and a machine learning algorithm. However, these two systems either have to be fine-tuned manually or impose limitations on the type of garbage collector. Only one work takes into account the collocation of VMs and the need to transfer memory between them [48]. Even so, it is focused on the individual performance of each instance and not the distribution of memory based on the progress of each workload.

Figure 1.12 depicts the overall RCI of each system that augments a high-level language VM, complementing the analysis of Table 1.10. As in the case of system-level VMs, systems have design options that favor responsiveness. The system taking into account the elasticity curve of micro-economics has the highest level of responsiveness perhaps because of its low overall intricateness of sensors, decision process, and actuators. We also see that the extra intricateness of the decision phase in "Control" and "Learning" had a cost. In the first case, it was the overall responsiveness, while in the second the system had to be designed with a smaller number of sensors, reducing comprehensiveness. Further research is needed to determine if other unexplored techniques in these two fields can bring more advantage.

**Table 1.10** HLL-VM Systems

| System | Dominant Resource | Monitor | Decision | Action | Modifications |
|---|---|---|---|---|---|
| JRES | Mix | CPU, Heap, I/O | Rules | Limitation (CPU, Heap, I/O) | VM |
| Isla Vista | Mem | Allocation stalls in OS | Rules | Heap rezise | VM |
| Resource-driven | Mem | Page faults, Resident set size | 3 types of rules | Whole heap collection | VM |
| Control | Mem | GC overhead | PID Controller | Heap resize | Yes |
| PAMM | Mem | Heap Size, Page Faults | Threshold | Run GC | Program |
| CRAMM | Mem | WSS via Virtual Memory Manager, Heap Utilization | Fixed rule | Heap resize | VM/OS |
| Elasticity Curve | Mem | Number of GCs, Heap size | Elasticity threshold | Heap resize | VM |
| Switch | Mem | Heap Size, GC load, GC frequency | Threshold rule | GC algorithm | VM |
| Learning | Mem | Available Memory (current and variation between observations) | Reinforcement learning | Run GC | VM |

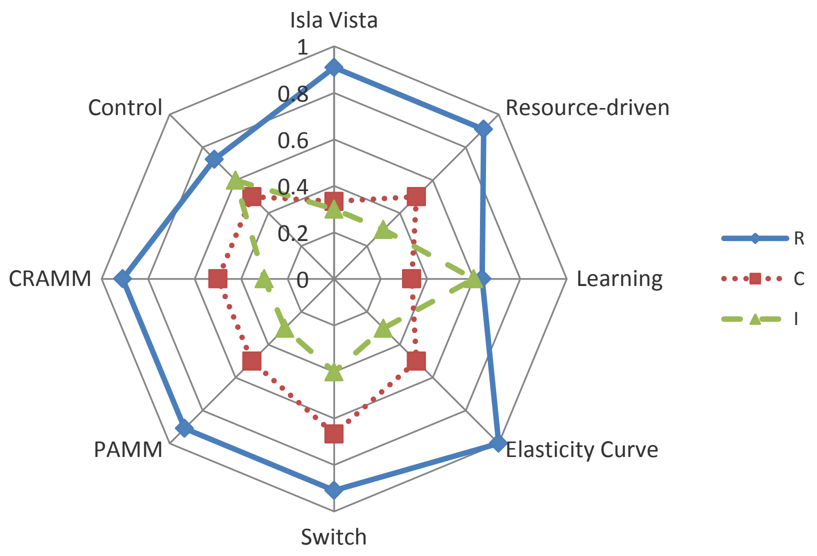

**Fig. 1.12** RCI of HLL-VMs

## 1.6 Summary and Open Research Issues

In this chapter, we reviewed the main approaches for adaptation and monitoring in virtual machines, their tradeoffs, and their main mechanisms for resource management. We framed them into the adaptation loop model (monitoring, decision, and actuation). Furthermore, we proposed a novel taxonomy and classification framework that, when applied to a group of systems, can help visually in determining their similarities and differences. Framed by this, we presented a comprehensive survey and analysis of relevant techniques and systems in the context of virtual machine monitoring and adaptability.

This taxonomy was inspired by two conjectures that arise from the analysis of existing relevant work in monitoring and adaptability of virtual machines. We presented the RCI conjecture on monitoring and adaptability in systems, identifying the fundamental tension among Responsiveness, Comprehensiveness, and Intricateness, and how a given adaptation technique aiming at achieving improvements on two of these aspects, can only do so at the cost of the remaining one.

In last years, the widespread use of management system for containerized applications, like Docker [4] and kubernetes [3], resurrected the interest of container-based operating system (COS) [88]. Sys-VMs allow for each guest so have a complete stack of the operating system and applications running in isolation from other guests. In contrast to this, containers loose some of the CPU and I/O isolation and share the same kernel OS, promising close to bare metal performance [72]. A container-based approach can give high-performance computing applications an easy and light way to transport jobs and a comprehensive resource scheduling environment [102].

An approach that is also becoming popular is the use of containers inside Sys-VMs. Doing so, datacenter providers can reuse their current virtualization infrastructure while going towards the need of more users. Also from a desktop environment point of view, having containers inside Sys-VMs allows the for non-Linux users to enjoy this technology and benefit from an extra degree of isolation when running their sensitive workloads [102, 32].

With managed runtimes dominating the landscape of systems to process big-data, research will continue to reduce the impact of platforms in workload's performance, especially regarding automatic memory management and the interface between HLL-VMs and the underlying execution stack. Regarding memory management, the generational principle is well suited for most general applications but in big-data deployments, either related to storage or stream processing, this assumption is not always beneficial and new segmenting options have to be considered [37]. Regarding the deployments of managed runtimes, further efforts are necessary to explore how to reduce the cost of interfacing with operating services (especially I/O) as this is also a cause of performance bottleneck. A research opportunity are hybrid runtimes that run in kernel mode and take direct advantage of the available hardware [45].

**Acknowledgements**

This work was supported by national funds through Fundação para a Ciência e a Tecnologia with reference PTDC/EEI-SCR/6945/2014, and by the ERDF through COMPETE 2020 Programme, within project POCI-01-0145-FEDER-016883, the Engineering School of the Polytechnic Institute of Lisbon (ISEL/IPL).

## References

1. Acm digital library, visited november 2016
2. Enabling intel virtualization technology features and benefits. visited november 2016.
3. http://kubernetes.io, visited november 2016
4. https://www.docker.com/, visited november 2016
5. Lxc. https://linuxcontainers.org/, visited november 2016.
6. Windows server containers. https://msdn.microsoft.com/en-us/virtualization/windowscontainers/about/index, visited november 2016.
7. An architectural blueprint for autonomic computing. Tech. rep., IBM (Jun 2005)
8. Adams, K., Agesen, O.: A comparison of software and hardware techniques for x86 virtualization. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 2–13. ASPLOS XII, ACM, New York, NY, USA (2006)
9. Agmon Ben-Yehuda, O., Posener, E., Ben-Yehuda, M., Schuster, A., Mu'alem, A.: Ginseng: Market-driven memory allocation. In: Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments. pp. 41–52. VEE '14, ACM, New York, NY, USA (2014)
10. Alpern, B., Augart, S., Blackburn, S.M., Butrico, M., Cocchi, A., Cheng, P., Dolby, J., Fink, S., Grove, D., Hind, M., McKinley, K.S., Mergen, M., Moss, J.E.B., Ngo, T., Sarkar, V.: The jikes research virtual machine project: building an open-source research community. IBM Syst. J. 44, 399–417 (January 2005), http://dx.doi.org/10.1147/sj.442.0399
11. Amdahl, G.M., Blaauw, G.A., Brooks, F.P.: Architecture of the IBM system/360. IBM J. Res. Dev. 8, 87–101 (April 1964), http://dx.doi.org/10.1147/rd.82.0087
12. Amit, N., Tsafrir, D., Schuster, A.: Vswapper: A memory swapper for virtualized environments. In: Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 349–366. ASPLOS '14, ACM, New York, NY, USA (2014), http://doi.acm.org/10.1145/2541940.2541969
13. Andreasson, E., Hoffmann, F., Lindholm, O.: To collect or not to collect? Machine learning for memory management. In: Proceedings of the 2nd Java Virtual Machine Research and Technology Symposium. pp. 27–39. USENIX Association, Berkeley, CA, USA (2002)
14. Arnold, M., Fink, S.J., Grove, D., Hind, M., Sweeney, P.F.: A survey of adaptive optimization in virtual machines. In: Proceedings of the IEEE, 93(2), 2005. Special Issue on Program Generation, Optimization, ans Adaptation. pp. 449–466 (2005)
15. Back, G., Hsieh, W.C.: The KaffeOS Java runtime system. ACM Trans. Program. Lang. Syst. 27, 583–630 (July 2005), http://doi.acm.org/10.1145/1075382.1075383
16. Baker, H.G.: Thermodynamics and garbage collection. SIGPLAN Not. 29, 58–63 (April 1994), http://doi.acm.org/10.1145/181761.181770
17. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the art of virtualization. SIGOPS Oper. Syst. Rev. 37, 164–177 (October 2003), http://doi.acm.org/10.1145/1165389.945462

18. Beloglazov, A., Buyya, R.: Energy efficient resource management in virtualized cloud data centers. In: Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on. pp. 826–831 (May 2010)
19. Binder, W., Hulaas, J., Moret, P., Villazón, A.: Platform-independent profiling in a virtual execution environment. Softw. Pract. Exper. 39, 47–79 (January 2009), http://portal.acm.org/citation.cfm?id=1464245.1464249
20. Blackburn, S.M., Garner, R., Hoffmann, C., Khang, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J.E.B., Moss, B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., Wiedermann, B.: The DaCapo benchmarks: Java benchmarking development and analysis. In: OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications. pp. 169–190. ACM, New York, NY, USA (2006)
21. Blake, C., Rodrigues, R.: High availability, scalable storage, dynamic peer networks: Pick two. In: Jones, M.B. (ed.) HotOS. pp. 1–6. USENIX (2003)
22. Bobroff, N., Westerink, P., Fong, L.: Active control of memory for Java virtual machines and applications. In: 11th International Conference on Autonomic Computing (ICAC 14). pp. 97–103. USENIX Association, Philadelphia, PA (Jun 2014), https://www.usenix.org/conference/icac14/technical-sessions/presentation/bobroff
23. Brewer, E.A.: A certain freedom: thoughts on the CAP theorem. In: Richa, A.W., Guerraoui, R. (eds.) PODC. p. 335. ACM (2010)
24. Buyya, R., Yeo, C.S., Venugopal, S., Broberg, J., Brandic, I.: Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. Future Gener. Comput. Syst. 25(6), 599–616 (Jun 2009)
25. Cheng, L., Wang, C.L.: vbalance: Using interrupt load balance to improve i/o performance for smp virtual machines. In: Proceedings of the Third ACM Symposium on Cloud Computing. pp. 2:1–2:14. SoCC '12, ACM, New York, NY, USA (2012)
26. Cherkasova, L., Gupta, D., Vahdat, A.: Comparison of the three cpu schedulers in xen. SIGMETRICS Perform. Eval. Rev. 35, 42–51 (September 2007), http://doi.acm.org/10.1145/1330555.1330556
27. Chiu, D.M., Jain, R.: Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. Comput. Netw. ISDN Syst. 17(1), 1–14 (Jun 1989)
28. Click, C., Tene, G., Wolf, M.: The pauseless gc algorithm. In: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments. pp. 46–56. VEE '05, ACM, New York, NY, USA (2005), http://doi.acm.org/10.1145/1064979.1064988
29. Czajkowski, G., Wegiel, M., Daynes, L., Palacz, K., Jordan, M., Skinner, G., Bryce, C.: Resource management for clusters of virtual machines. In: Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid - Volume 01. pp. 382–389. CCGRID '05, IEEE Computer Society, Washington, DC, USA (2005), http://portal.acm.org/citation.cfm?id=1169222.1169492
30. Czajkowski, G., von Eicken, T.: Jres: a resource accounting interface for java. In: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. pp. 21–35. OOPSLA '98, ACM, New York, NY, USA (1998), http://doi.acm.org/10.1145/286936.286944
31. Czajkowski, G., Hahn, S., Skinner, G., Soper, P., Bryce, C.: A resource management interface for the java platform. Softw. Pract. Exper. 35, 123–157 (February 2005), http://portal.acm.org/citation.cfm?id=1055953.1055955
32. Dantas, B., Fleitas, C., Francisco, A.P., Simão, J., Vaz, C.: Beyond NGS data sharing and towards open science (Nov 2016), https://doi.org/10.5281/zenodo.190489
33. Deutsch, L.P., Schiffman, A.M.: Efficient implementation of the smalltalk-80 system. In: Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. pp. 297–302. POPL '84, ACM, New York, NY, USA (1984), http://doi.acm.org/10.1145/800017.800542

34. Farahnakian, F., Pahikkala, T., Liljeberg, P., Plosila, J., Hieu, N.T., Tenhunen, H.: Energy-aware vm consolidation in cloud data centers using utilization prediction model. IEEE Transactions on Cloud Computing PP(99), 1–1 (2016)
35. Gidra, L., Thomas, G., Sopena, J., Shapiro, M.: A study of the scalability of stop-the-world garbage collectors on multicores. In: Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 229–240. ASPLOS '13, ACM, New York, NY, USA (2013)
36. Gidra, L., Thomas, G., Sopena, J., Shapiro, M., Nguyen, N.: Numagic: A garbage collector for big data on big numa machines. In: Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 661–673. ASPLOS '15, ACM, New York, NY, USA (2015), http://doi.acm.org/10.1145/2694344.2694361
37. Gog, I., Giceva, J., Schwarzkopf, M., Vaswani, K., Vytiniotis, D., Ramalingan, G., Murray, D., Hand, S., Isard, M.: Broom: Sweeping out garbage collection from big data systems. In: Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems. pp. 2–2. HOTOS'15, USENIX Association, Berkeley, CA, USA (2015), http://dl.acm.org/citation.cfm?id=2831090.2831092
38. Goldberg, R.P.: Survey of virtual machine research. Computer 7(9), 34–45 (Sep 1974)
39. Gong, Z., Gu, X., Wilkes, J.: Press: Predictive elastic resource scaling for cloud systems. In: Network and Service Management (CNSM), 2010 International Conference on. pp. 9 –16 (oct 2010)
40. Gordon, A., Amit, N., Har'El, N., Ben-Yehuda, M., Landau, A., Schuster, A., Tsafrir, D.: ELI: Bare-metal performance for I/O virtualization. In: Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 411–422. ASPLOS XVII, ACM, New York, NY, USA (2012)
41. Grzegorczyk, C., Soman, S., Krintz, C., Wolski, R.: Isla vista heap sizing: Using feedback to avoid paging. In: Proceedings of the International Symposium on Code Generation and Optimization. pp. 325–340. CGO '07, IEEE Computer Society, Washington, DC, USA (2007), http://dx.doi.org/10.1109/CGO.2007.20
42. Guan, X., Srisa-an, W., Jia, C.: Investigating the effects of using different nursery sizing policies on performance. In: Proceedings of the 2009 international symposium on Memory management. pp. 59–68. ISMM '09, ACM, New York, NY, USA (2009), http://doi.acm.org/10.1145/1542431.1542441
43. Gupta, D., Lee, S., Vrable, M., Savage, S., Snoeren, A.C., Varghese, G., Voelker, G.M., Vahdat, A.: Difference engine: harnessing memory redundancy in virtual machines. In: Proceedings of the 8th USENIX conference on Operating systems design and implementation. pp. 309–322. OSDI'08, USENIX Association, Berkeley, CA, USA (2008), http://dl.acm.org/citation.cfm?id=1855741.1855763
44. Hagimont, D., Mayap Kamga, C., Broto, L., Tchana, A., Palma, N.: DVFS aware CPU credit enforcement in a virtualized system. In: Middleware 2013, Lecture Notes in Computer Science, vol. 8275, pp. 123–142. Springer Berlin Heidelberg (2013)
45. Hale, K.C., Dinda, P.A.: Enabling hybrid parallel runtimes through kernel and virtualization support. In: Proceedings of the12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments. pp. 161–175. VEE '16, ACM, New York, NY, USA (2016), http://doi.acm.org/10.1145/2892242.2892255
46. Heo, J., Zhu, X., Padala, P., Wang, Z.: Memory overbooking and dynamic control of xen virtual machines in consolidated environments. In: Proceedings of the 11th IFIP/IEEE international conference on Symposium on Integrated Network Management. pp. 630–637. IM'09, IEEE Press, Piscataway, NJ, USA (2009), http://dl.acm.org/citation.cfm?id=1688933.1689025
47. Hertz, M., Bard, J., Kane, S., Keudel, E., Bai, T., Kelsey, K., Ding, C.: Waste not,want not: resource-based garbage collection in a shared environment. Tech. Rep. TR-2006-908, University of Rochester (2009)

48. Hertz, M., Kane, S., Keudel, E., Bai, T., Ding, C., Gu, X., Bard, J.E.: Waste not, want not: resource-based garbage collection in a shared environment. In: Proceedings of the international symposium on Memory management. pp. 65–76. ISMM '11, ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/1993478.1993487
49. Hinesa, M., Gordon, A., Silva, M., Silva, D.D., Ryu, K.D., Ben-Yehuda, M.: Applications know best: Performance-driven memory overcommit with ginkgo. In: CloudCom '11: 3rd IEEE International Conference on Cloud Computing Technology and Science. pp. 130–137 (2011)
50. Hoffmann, H., Eastep, J., Santambrogio, M.D., Miller, J.E., Agarwal, A.: Application heartbeats: a generic interface for specifying program performance and goals in autonomous computing environments. In: Proceedings of the 7th international conference on Autonomic computing. pp. 79–88. ICAC '10 (2010)
51. Hulaas, J., Binder, W.: Program transformations for light-weight cpu accounting and control in the java virtual machine. Higher Order Symbol. Comput. 21, 119–146 (June 2008), http://portal.acm.org/citation.cfm?id=1380610.1380611
52. Jones, R., Hosking, A., Moss, E.: The Garbage Collection Handbook: The Art of Automatic Memory Management. Chapman & Hall/CRC, 1st edn. (2011)
53. Kesavan, M., Gavrilovska, A., Schwan, K.: On disk i/o scheduling in virtual machines. In: Proceedings of the 2nd conference on I/O virtualization. pp. 6–6. WIOV'10, USENIX Association, Berkeley, CA, USA (2010), http://portal.acm.org/citation.cfm?id=1863181.1863187
54. Kulkarni, S., Cavazos, J.: Mitigating the compiler optimization phase-ordering problem using machine learning. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications. pp. 147–162. OOPSLA '12, ACM, New York, NY, USA (2012)
55. Liu, H., Jin, H., Liao, X., Deng, W., He, B., z. Xu, C.: Hotplug or ballooning: A comparative study on dynamic memory management techniques for virtual machines. IEEE Transactions on Parallel and Distributed Systems 26(5), 1350–1363 (May 2015)
56. Lublin, U., Kamay, Y., Laor, D., Liguori, A.: KVM: the Linux virtual machine monitor. In: Ottawa Linux Symposium (2007)
57. Maas, M., Asanović, K., Harris, T., Kubiatowicz, J.: Taurus: A holistic language runtime system for coordinating distributed managed-language applications. In: Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 457–471. ASPLOS '16 (2016)
58. Maggio, M., Hoffmann, H., Papadopoulos, A.V., Panerati, J., Santambrogio, M.D., Agarwal, A., Leva, A.: Comparison of decision-making strategies for self-optimization in autonomic computing systems. ACM Trans. Auton. Adapt. Syst. 7(4), 36:1–36:32 (Dec 2012), http://doi.acm.org/10.1145/2382570.2382572
59. Manson, J., Pugh, W., Adve, S.V.: The Java memory model. SIGPLAN Not. 40, 378–391 (January 2005), http://doi.acm.org/10.1145/1047659.1040336
60. Mao, F., Zhang, E.Z., Shen, X.: Influence of program inputs on the selection of garbage collectors. In: Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments. pp. 91–100. VEE '09, ACM, New York, NY, USA (2009), http://doi.acm.org/10.1145/1508293.1508307
61. Mian, R., Martin, P., Zulkernine, F., Vazquez-Poletti, J.L.: Estimating resource costs of data-intensive workloads in public clouds. In: Proceedings of the 10th International Workshop on Middleware for Grids, Clouds and e-Science. pp. 3:1–3:6. MGC '12, ACM, New York, NY, USA (2012)
62. Min, C., Kim, I., Kim, T., Eom, Y.I.: Vmmb: Virtual machine memory balancing for unmodified operating systems. J. Grid Comput. 10(1), 69–84 (Mar 2012), http://dx.doi.org/10.1007/s10723-012-9209-4
63. Ongaro, D., Cox, A.L., Rixner, S.: Scheduling I/O in virtual machine monitors. In: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments. pp. 1–10. VEE '08, ACM, New York, NY, USA (2008), http://doi.acm.org/10.1145/1346256.1346258

64. Oracle: Java management extensions (jmx) technology, visited 28-11-2016
65. Ousterhout, J.K.: Scheduling techniques for concurrent systems. In: ICDCS. pp. 22–30. IEEE Computer Society (1982)
66. Padala, P., Hou, K.Y., Shin, K.G., Zhu, X., Uysal, M., Wang, Z., Singhal, S., Merchant, A.: Automated control of multiple virtualized resources. In: Proceedings of the 4th ACM European conference on Computer systems. pp. 13–26. EuroSys '09, ACM, New York, NY, USA (2009), http://doi.acm.org/10.1145/1519065.1519068
67. Park, S.M., Humphrey, M.: Self-tuning virtual machines for predictable escience. In: Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid. pp. 356–363. CCGRID '09, IEEE Computer Society, Washington, DC, USA (2009), http://dx.doi.org/10.1109/CCGRID.2009.84
68. Ram, K.K., Santos, J.R., Turner, Y.: Redesigning xen's memory sharing mechanism for safe and efficient I/O virtualization. In: Proceedings of the 2Nd Conference on I/O Virtualization. WIOV'10, USENIX Association, Berkeley, CA, USA (2010)
69. Salehie, M., Tahvildari, L.: Self-adaptive software: Landscape and research challenges. ACM Trans. Auton. Adapt. Syst. 4, 14:1–14:42 (May 2009), http://doi.acm.org/10.1145/1516533.1516538
70. Salomie, T.I., Alonso, G., Roscoe, T., Elphinstone, K.: Application level ballooning for efficient server consolidation. In: Proceedings of the 8th ACM European Conference on Computer Systems. pp. 337–350. EuroSys '13, ACM, New York, NY, USA (2013), http://doi.acm.org/10.1145/2465351.2465384
71. Shao, Z., Jin, H., Li, Y.: Virtual machine resource management for high performance computing applications. Parallel and Distributed Processing with Applications, International Symposium on 0, 137–144 (2009)
72. Sharma, P., Chaufournier, L., Shenoy, P., Tay, Y.C.: Containers and virtual machines at scale: A comparative study. In: Proceedings of the 17th International Middleware Conference. pp. 1:1–1:13. Middleware '16, ACM, New York, NY, USA (2016), http://doi.acm.org/10.1145/2988336.2988337
73. Silva, J.N., Veiga, L., Ferreira, P.: A$^2$HA - Automatic and adaptive host allocation in utility computing for bag-of-tasks. J. Internet Services and Applications 2(2), 171–185 (2011)
74. Simão, J., Veiga, L.: A classification of middleware to support virtual machines adaptability in iaas. In: Proceedings of the 11th International Workshop on Adaptive and Reflective Middleware. pp. 5:1–5:6. ARM '12, ACM, New York, NY, USA (2012)
75. Simão, J., Lemos, J., Veiga, L.: A$^2$-VM a cooperative Java VM with support for resource-awareness and cluster-wide thread scheduling. In: 19th International Conference on Cooperative Information Systems (CoopIS 2011). LNCS, Springer (September 2011)
76. Simao, J., Rameshan, N., Veiga, L.: Resource-aware scaling of multi-threaded java applications in multi-tenancy scenarios. In: Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on. vol. 1, pp. 445–451. IEEE (2013)
77. Simão, J., Singer, J., Veiga, L.: A comparative look at adaptive memory management in virtual machines. In: IEEE CloudCom 2013. IEEE (December 2013)
78. Simão, J., Veiga, L.: Qoe-jvm: An adaptive and resource-aware java runtime for cloud computing. In: OTM Confederated International Conferences" On the Move to Meaningful Internet Systems". pp. 566–583. Springer Berlin Heidelberg (2012)
79. Simao, J., Veiga, L.: Vm economics for java cloud computing: An adaptive and resource-aware java runtime with quality-of-execution. In: Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012). pp. 723–728. IEEE Computer Society (2012)
80. Simão, J., Veiga, L.: Flexible slas in the cloud with a partial utility-driven scheduling architecture. In: IEEE 5th International Conference on Cloud Computing Technology and Science, CloudCom 2013, Bristol, United Kingdom, De-

cember 2-5, 2013, Volume 1. pp. 274–281. IEEE Computer Society (2013), http://dx.doi.org/10.1109/CloudCom.2013.43
81. Simão, J., Veiga, L.: 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. pp. 357–362. ACM (2013)
82. Simão, J., Veiga, L.: Partial utility-driven scheduling for flexible SLA and pricing arbitration in cloud. IEEE Transactions on Cloud Computing (Nov 2014)
83. Singer, J., Brown, G., Watson, I., Cavazos, J.: Intelligent selection of application-specific garbage collectors. In: Proceedings of the 6th international symposium on Memory management. pp. 91–102. ISMM '07, ACM, New York, NY, USA (2007), http://doi.acm.org/10.1145/1296907.1296920
84. Singer, J., Jones, R.: Economic utility theory for memory management optimization. In: Rogers, I. (ed.) Proceedings of the workshop on Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems. p. 4. ACM (July 2011), http://www.cs.kent.ac.uk/pubs/2011/3156, (Position paper)
85. Singer, J., Jones, R.E., Brown, G., Luján, M.: The economics of garbage collection. SIGPLAN Not. 45, 103–112 (June 2010), http://doi.acm.org/10.1145/1837855.1806669
86. Singer, J., Kovoor, G., Brown, G., Luján, M.: Garbage collection auto-tuning for java mapreduce on multi-cores. In: Proceedings of the International Symposium on Memory Management. pp. 109–118. ISMM '11, ACM, New York, NY, USA (2011)
87. Smith, J., Nair, R.: Virtual Machines: Versatile Platforms for Systems and Processes. Morgan Kaufmann (2005)
88. Soltesz, S., Pötzl, H., Fiuczynski, M.E., Bavier, A., Peterson, L.: Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. In: Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007. pp. 275–287. EuroSys '07, ACM, New York, NY, USA (2007), http://doi.acm.org/10.1145/1272996.1273025
89. Soman, S., Krintz, C.: Application-specific garbage collection. J. Syst. Softw. 80, 1037–1056 (July 2007), http://dx.doi.org/10.1016/j.jss.2006.12.566
90. Soman, S., Krintz, C., Bacon, D.F.: Dynamic selection of application-specific garbage collectors. In: Proceedings of the 4th international symposium on Memory management. pp. 49–60. ISMM '04, ACM, New York, NY, USA (2004), http://doi.acm.org/10.1145/1029873.1029880
91. Stoica, I., Abdel-Wahab, H., Jeffay, K.: On the duality between resource reservation and proportional share resource allocation. Tech. rep., Old Dominion University, Norfolk, VA, USA (1996)
92. Suri, N., Bradshaw, J.M., Breedy, M.R., Groth, P.T., Hill, G.A., Saavedra, R.: State capture and resource control for java: the design and implementation of the aroma virtual machine. In: Proceedings of the Symposium on JavaTM Virtual Machine Research and Technology Symposium. pp. 11–11. JVM'01, USENIX Association, Berkeley, CA, USA (2001), http://portal.acm.org/citation.cfm?id=1267847.1267858
93. Tanenbaum, A.S.: Modern Operating Systems. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edn. (2007)
94. Tay, Y.C., Zong, X., He, X.: An equation-based heap sizing rule. Perform. Eval. 70(11), 948–964 (Nov 2013)
95. Tchana, A., Palma, N.D., Safieddine, I., Hagimont, D., Diot, B., Vuillerme, N.: Software Consolidation as an Efficient Energy and Cost Saving Solution for a SaaS/PaaS Cloud Model, pp. 305–316. Springer Berlin Heidelberg, Berlin, Heidelberg (2015)
96. Tene, G., Iyengar, B., Wolf, M.: C4: The continuously concurrent compacting collector. SIGPLAN Not. 46(11), 79–88 (Jun 2011)
97. Vaquero, L.M., Rodero-Merino, L., Caceres, J., Lindner, M.: A break in the clouds: Towards a cloud definition. SIGCOMM Comput. Commun. Rev. 39(1), 50–55 (Dec 2008)

98. Veiga, L., Ferreira, P.: Incremental replication for mobility support in obiwan. In: Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on. pp. 249–256. IEEE (2002)
99. Veiga, L., Ferreira, P.: Poliper: policies for mobile and pervasive environments. In: Kon, F., Costa, F.M., Wang, N., Cerqueira, R. (eds.) Proceedings of the 3rd Workshop on Adaptive and Reflective Middleware, ARM 2003, Toronto, Ontario, Canada, October 19, 2004. pp. 238–243. ACM (2004), http://doi.acm.org/10.1145/1028613.1028623
100. VMware: VMware vSpher 4: The CPU Scheduler in VMware ESX 4 (2009)
101. Waldspurger, C.A.: Memory resource management in VMware ESX server. SIGOPS Oper. Syst. Rev. 36, 181–194 (December 2002), http://doi.acm.org/10.1145/844128.844146
102. Weidner, O., Atkinson, M., Barker, A., Filgueira Vicente, R.: Rethinking high performance computing platforms: Challenges, opportunities and recommendations. In: Proceedings of the ACM International Workshop on Data-Intensive Distributed Computing. pp. 19–26. DIDC '16, ACM, New York, NY, USA (2016), http://doi.acm.org/10.1145/2912152.2912155
103. Weiming, Z., Zhenlin, W.: Dynamic memory balancing for virtual machines. In: Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments. pp. 21–30. VEE '09 (2009)
104. Weng, C., Liu, Q., Yu, L., Li, M.: Dynamic adaptive scheduling for virtual machines. In: Proceedings of the 20th International Symposium on High Performance Distributed Computing. pp. 239–250. HPDC '11, ACM, New York, NY, USA (2011)
105. Weng, C., Wang, Z., Li, M., Lu, X.: The hybrid scheduling framework for virtual machine systems. In: Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments. pp. 111–120. VEE '09, ACM, New York, NY, USA (2009), http://doi.acm.org/10.1145/1508293.1508309
106. White, D.R., Singer, J., Aitken, J.M., Jones, R.E.: Control theory for principled heap sizing. In: Proceedings of the 2013 International Symposium on Memory Management. pp. 27–38. ISMM '13, ACM, New York, NY, USA (2013)
107. Wilson, P.R.: Uniprocessor garbage collection techniques. In: Proceedings of the International Workshop on Memory Management. pp. 1–42. IWMM '92, Springer-Verlag, London, UK (1992), http://portal.acm.org/citation.cfm?id=645648.664824
108. Xu, F., Liu, F., Jin, H., Vasilakos, A.: Managing performance overhead of virtual machines in cloud computing: A survey, state of the art, and future directions. Proceedings of the IEEE 102(1), 11–31 (2014)
109. Yang, T., Berger, E.D., Kaplan, S.F., Moss, J.E.B.: Cramm: Virtual memory support for garbage-collected applications. In: Proceedings of the 7th Symposium on Operating Systems Design and Implementation. pp. 103–116. OSDI '06, USENIX Association, Berkeley, CA, USA (2006)
110. Zhang, Y., Bestavros, A., Guirguis, M., Matta, I., West, R.: Friendly virtual machines: leveraging a feedback-control model for application adaptation. In: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments. pp. 2–12. VEE '05, ACM, New York, NY, USA (2005), http://doi.acm.org/10.1145/1064979.1064983