TÉCNICO LISBOA

An Adaptive Java Runtime Environment for Cloud Computing

José Manuel de Campos Lages Garcia Simão
(Mestre)

Proposta de dissertação a apresentar à
Comissão de Acompanhamento de Tese

Programa Doutoral em
Engenharia Informática e de Computadores

Júri

Presidente: Doutor David Martins de Matos

Orientador: Doutor Luís Manuel Antunes Veiga

Vogal: Doutor Paulo Jorge Pimenta Marques

Agosto de 2013

Abstract. This work investigates how an adaptable managed execution environment can be deployed in cloud environments to effectively explore the large amount of resources available in these infrastructures. Cloud infrastructures execute workloads from different tenants supported by a non-trivial virtualization stack, including high level language virtual machines (HLL-VMs), operating system (OS) services and system-level virtual machines (Sys-VMs). Currently, for a generic workload to be deployed in cloud-like infrastructures, it must have to deal with the intricateness of manually managing resources, available over different Sys-VMs. In this scenario, the allocation of physical resources to both Sys-VMs and HLL-VMs (e.g. overall memory, heap size, CPU cores) and the distribution of execution flows is done in an equal, mostly non-transparent fashion, missing the opportunity to manage the available resources in a more efficient and application-centric way.

This document starts by surveying the state of the art of adaptable resource management in virtual machines, framing them into a novel and thorough classification. It then presents and discusses the architecture of the ongoing work on a comprehensive adaptable Java runtime, with its enforcement mechanisms, and an Economics-inspired model to govern resource allocation. Relevant implementation details of the resource management mechanisms on which this model can act are presented, with an increasing level of abstraction: (i) low-level resource management, (ii) state checkpoint for migration and (iii) distribution of execution flows. These enforcement mechanisms were implemented and evaluated, measuring their benefits to the workload execution (i.e. effectiveness in the distribution of resources, total execution time) but also the overhead they add to the baseline execution. Results show the benefits of the proposed approach globally overcome the cost they introduce.

Contents

List of Figures

1

Introduction

1.1 Motivation

In today's scenarios of large scale computing and service providing, the deployment of software workloads in distributed infrastructures, namely computer clusters, is a very active research area. In recent years, the use of Grids, Utility and Cloud Computing, shows that these are approaches with growing interest and applicability, as well as scientific and commercial impact [66, 59, 11, 48, 62].

Managed languages (e.g., Java, C#) are becoming increasingly relevant in the development of large scale solutions, leveraging the benefits of a virtual execution environment (VEE) to provide secure, manageable and component-oriented solutions. Relevant examples include work done in various areas such as web application hosting, data processing, enterprise services, supply-chain platforms, implementation of functionality in service-oriented architectures. The field of e-Science also shows an increasing interest in Java for physics simulation, economics/statistics, network simulation, chemistry, computational biology and bioinformatics [46, 36, 52], showing that to some extend, high performance and high throughput computing have also been ported to managed languages.

To extend the benefits of a local VEE, while allowing scale-out regarding performance and memory requirements, several solutions have been proposed to federate Java virtual machines [83, 5, 85], aiming to provide a single system image where the managed application can benefit from the global resources of the cluster. If this system image has elasticity, in the sense that resources are made available proportionally to the effective need, and if these resources are accounted/charged as they are used, we can provide an high-level language virtual machine (HLL-VM) across the cluster, as an utility. If these changes are made dynamically (instead of explicitly by their users) we will have an adaptive and resource-aware virtual machine, that can be offered as a value-added Platform-as-a-Service (PaaS).

Often, when the execution of one of these applications is terminated abruptly due to a failure (regardless of it being caused by hardware of software fault, lack of available resources etc.), all of its work already carried out is simply lost and, when the application is later reexecuted with the same parameters and input (e.g., as in the case of a data-processing job), it has to restart its work from scratch, wasting resources and time, while still being prone to another failure, to delay its completion with no deadline guarantees. A possible solution to solve these problems, is through mechanisms of checkpoint and migration of applications made available through a resource-aware enabled HLL-VM. With these mechanisms, an application becomes more robust, as most of the work or calculations already performed can be recovered and execution resumed from an earlier point in time. It gains flexibility by being able to move to other nodes, without intervention from the programmer, regulated by a global policy enforced to each HLL-VM.

A HLL-VM cluster-enabled environment can execute applications with very different resource requirements. This leads to the use of selected algorithms for runtime and system services, aiming to maximize the performance of the applications running on the cluster. However, for other applications, for example the ones owned by restricted users, it can be necessary to impose limits to their resource consumption. These two requirements can only be fulfilled if the cluster can monitor and control the resources it uses both at the HLL-VM and distributed level, and whether the several local HLL-VM, each running on its node, are able to cooperate to manage resources overall.

1.2 Resource allocation and effective progress

In a shared computer cluster, applications running on a given node compete for the finite resources of that machine (e.g. CPU, memory, I/O). Each application is running to produce a set of results on behalf of a given user, but not all users have the some execution requirements or the some priority to complete their work. The work of Silva et al. [65] classifies users in four different types, in order to apply differentiating policies to the work of these users. In academic institutions, for example, the same grid can be used to run e-science applications by students in different academic levels. Using the same infrastructure will have fewer costs and will be easier to maintain. Nevertheless, the managers of the infrastructure will want to impose a high level policy and give distinguished execution quality to different types of students. The mechanisms to obtain this can range from restraining resource consumption (e.g. CPU usage, physical memory allocated), to the migration of the application to another node.

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

As more applications target managed runtimes, HLL-VMs is a relevant abstraction layer that has not been properly explored to enhance resource usage, control, and effectiveness, with increased rich semantics and flexibility. Therefore, managed runtimes, executing the workloads of multiple tenants, must adapt themselves to the execution of applications, with different (and sometimes dynamically changing) requirements in regard to their *quality-of-execution* (QoE).

QoE aims at capturing the adequacy and efficiency of the resources provided to an application according to its needs. Several metrics can be used to infer how applications are making progress given the resources they are using. It can be inferred coarsely from application execution time for medium running applications, or request execution times for more service driven ones such as those web-based, or from critical situations such as thrashing or starvation. Also, it can be derived in a more fine-grained way from incremental indicators of application progress, such as amount of input processed, disk and network output generated, execution phase detection or memory pages updates. Still, initially, the application execution times (or a way to estimate it [21]) is a relevant metric.

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

1.3 Current shortcomings

Existing approaches to cluster-enabled runtimes adaptability and internal mechanisms such as checkpointing and resource-awareness, are still not adequate for this intended scenario as they have not been combined into a single infrastructure for unmodified applications.

Existing public runtimes are not resource-aware in the sense that the use of resources is mostly bounded by the underlying operative system. On the other hand, in the research community, the proposed runtimes are focused on accounting resource usage to avoid application's bad behavior, and do not support the desired reconfigurability of their inner mechanisms [32, 12]. There is no notion of *resource effectiveness* in the sense that when there are scarce resources, there is no attempt to determine where to take such resources from applications (i.e. either isolation domains or the whole VM) where they hurt performance the least. Others have recently shown the importance of adaptability at the level of HLL-VMs, either

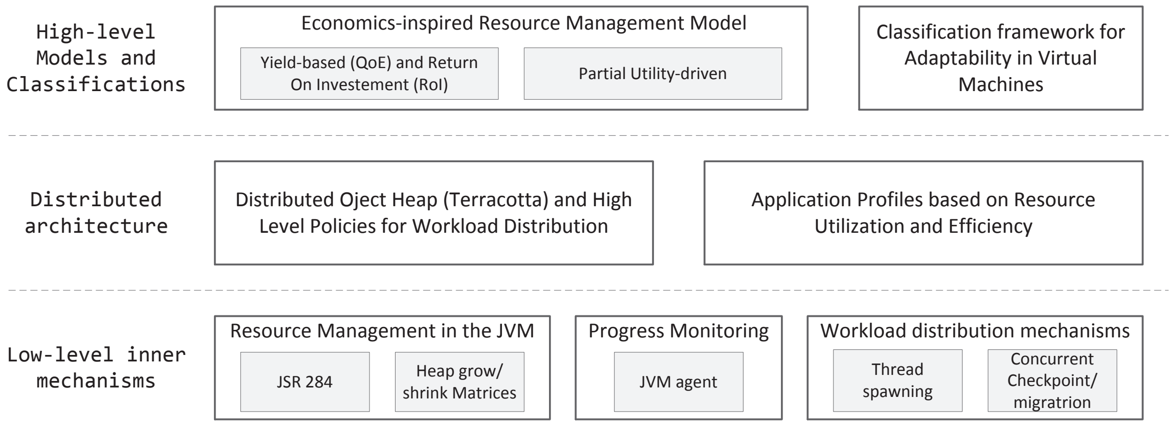

High-level Models and Classifications
Economics-inspired Resource Management Model
Yield-based (QoE) and Return On Investement (RoI)
Partial Utility-driven
Classification framework for Adaptability in Virtual Machines
Distributed architecture
Distributed Oject Heap (Terracotta) and High Level Policies for Workload Distribution
Application Profiles based on Resource Utilization and Efficiency
Low-level inner mechanisms
Resource Management in the JVM
JSR 284
Heap grow/ shrink Matrices
Progress Monitoring
JVM agent
Workload distribution mechanisms
Thread spawning
Concurrent Checkpoint/ migratrion

Fig. 1.1. Layered view of the researched topics

based on the application performance [56] or by changes in their environment [30]. Nevertheless, they are either dependent on a global optimization phase, limited to a single given type of resource, or make the application dependent on a new programming interface.

Furthermore, traditional mechanisms of checkpoint and migration are supported at process level or at system virtual machine. These approaches are insufficient because they either require to store/transfer information that is not on the application itself (e.g. information on the operating system on which it runs), or limit the portability of it.

Previous works in distributed execution environment have researched the semantics of single system image, mainly regarding data access. However they are neither elastic nor resource aware, and therefore are not fit for the multi-tenant scenarios we address. Several new programming models and languages [20, 73, 75, 51] have been proposed. They require the program to be bounded to yet an other programming interface, which invalidates the use of previous working solutions by non programming-expert users, e.g. some groups of e-science researchers. The architecture presented in [26] federates the multi-task virtual machine [27], forming a cluster where there are local and global resources that can be monitored and constrained. However, Czajkowski's work lacks the capacity to relocate workload across the cluster. Regarding policies, theirs are only defined programmatically and cannot be changed without recompiling the programs/libraries responsible by clustering mechanisms (e.g. load balancer).

1.4 Contributions

Because our system proposal is an adaptable, resource-aware, distributed execution environment, the contributions include all these topics with different emphasis. Figure 1.1 presents a schematic view of the topics which are currently under research, design and evaluation.

The following list presents a short description of the specific contributions covered in this document.

1. **VM's adaptability framework.** The first contribution is a framework to classify adaptability in virtual machines. It describes the adaptability loop of virtual machines discussing their mechanisms and techniques. It then proposes a way to classify each of those according to their responsiveness, i.e. the capacity to 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.

2. **Adaptability model.** We have design an Economics-inspired model to drive adaptability in environments where resources are shared by several tenants. Our adaptability model is used to determine from which tenants resource scarcity will hurt performance the least, putting resources where they can do the most good to applications and the cloud infrastructure provider. We describe the integration of this model into *QoE-JVM* , a distributed execution environment, where nodes cooperate to make an efficient management of the available local and global resources.

   Currently, we developed (and is under evaluation) a yield-based model to measure the results of different strategies regarding: a) the heap size, based on the relation between the ratio of live objects and the time spent in GC; b) CPU allocation in a per workload way; c) number of threads available in workloads with flexible configuration.

   We are also investigating how a partial utility-driven cost model can be used in scheduling at a lower system level, i.e. system-wide virtual machines. This is important for providers (either public or private) of the Infrastruture-as-a-Service (IaaS) service model, where system-wide virtual machines are rented to different clients.

3. **Low-level resource management.** As the fundamental building block of the distributed execution environment, we propose a HLL-VM with the ability to monitor base mechanisms (e.g. garbage collection performance, memory or network consumptions) in order to assess an application's performance and resource usage, and reconfigure these mechanisms in run-time according to previously defined resource allocation policies (or *quality-of-execution* specifications). These policies regulate how resources should be used by the application in the cluster, leading to the adaptation of components at different levels of the cluster in order to enforce them. High-level policies are evaluated by consumption points inside the HLL-VM but are defined externally to the execution environment, that is, by the application owner or the provider.

4. **Concurrent checkpointing.** We propose a novel solution to Java applications with long execution times, by incorporating checkpoint and migration mechanisms in a Java VM (Jikes RVM [2]). It is able to checkpoint multi-threaded applications, ensuring the checkpoint is a consistent snapshot of the execution taking into account thread concurrency

and synchronization, while avoiding application pause by performing the checkpoint concurrently (or incrementally) alongside with the application execution. Our techniques rely on two base mechanisms: on-stack-replacement (OSR) and yield points, existent in many other VM implementations (e.g. Sun HotSpot). Therefore, our techniques could be applied to other VMs. The main objectives are focused on the problems of transparency and completeness, and how these mechanisms can be activated according to low-level resource management and monitoring driven by policies.

5. **Cluster-wide thread placement.** We use a middleware that aggregates the heap space of individual VMs to form, within the cluster, a distributed shared object space assigned to a specific application. It gives running applications support for single system image (SSI) semantics, across the cluster, regarding objects address space and thread allocation. Techniques like bytecode enhancement/instrumentation or rewriting are used, so that unmodified applications can operate in a partitioned global address space, where some objects exist only as local copies and others are shared in a global heap. The middleware has been extended to provide SSI semantics regarding the execution of individual threads of each application. Furthermore, we present how the spawning of new threads can be controlled by high-level policies.

1.5 Published work

The work described in this document was partially published in the following journals, international conferences and associated events, i.e. workshops and poster sessions:

- **Journals**

  J.1 José Simão and Tiago Garrochinho and Luís Veiga, *A Checkpointing-enabled and Resource-Aware Java VM for Efficient and Robust e-Science Applications in Grid Environments*, Concurrency and Computation: Practice and Experience, 24(13), pp. 1421-1442, Sep. 2012, Wiley. *ISI Web of Knowledge IF* 0.84*, RADIST A.*

  J.2 José Simão and Carlos Nuno da Cruz Ribeiro and Paulo Ferreira and Luís Veiga, *Jano: Location-Privacy Enforcement in Mobile and Pervasive Environments through Declarative Policies* , Journal of Internet Services and Applications (JISA), 3(3), pp. 291-310, Dec. 2012, Springer. *SCImago SJR* 1.6*, Quartile 1 in category "Computer Networks and Communications", RADIST A.*

- **Conferences**

  C.1 José Simão and João Lemos and Luís Veiga, $A^2 - VM$*: A Cooperative Java VM with Support for Resource-Awareness and Cluster-Wide Thread Scheduling*, 19th International Conference on Cooperative Information Systems (CoopIS 2011), Sep. 2011, LNCS, Springer. *RADIST B, Core A, Acceptance ratio* ≈ 20%.

C.2 José Simão and Luís Veiga, *QoE-JVM: An Adaptive and Resource-Aware Java Runtime for Cloud Computing*, 2nd International Symposium on Secure Virtual Infrastructures (DOA-SVI 2012), OTM Conferences 2012, Sep. 2012, Springer, LNCS. *Acceptance ratio* ≈ 30%.

- **Workshops**

W.1 José Simão and Paulo Ferreira and Carlos Nuno da Cruz Ribeiro and Luís Veiga, *Jano - Specification and Enforcement of Location Privacy in Mobile and Pervasive Environments*, Workshop on Middleware for Pervasive Mobile and Embedded Computing (M-MPAC 2010), in Middleware 2010., Dec. 2010, ACM.

W.2 José Simão and Luís Veiga, *VM Economics for Java Cloud Computing: An Adaptive and Resource-Aware Java Runtime with Quality-of-Execution*, The 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2012) - Doctoral Symposium: Cloud Scheduling, Clusters and Data Centers, May. 2012, IEEE.

W.3 José Simão and Luís Veiga, *A Classification of Middleware to Support Virtual Machines Adaptability in IaaS*, 11th International Workshop on Adaptive and Reflective Middleware (ARM 2012), In conjunction with Middleware 2012, Dec. 2012, ACM.

- **Posters and talks**

P.1 José Simão and Luís Veiga, *Towards an Adaptive and Resource-Aware Java Runtime for Cloud Computing with Quality-of-Execution*, Poster session of the 17th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Mar. 2011. *Main conference is CORE A, RADIST A.*

T.1 José Simão and Luís Veiga, Invited talk in the Middleware'12 Doctoral Symposium, Dec. 2012.

P.2 José Simão and Axel Domingues and Luís Veiga, *Flexible SLAs in the Cloud With Partial-Utility Scheduling*, Poster Session of EuroSys 2013, Apr. 2013. *Main conference is CORE A, RADIST A.*

The following list shortly presents the publications and communications that support this document:

1. **C.1** describes the architecture and implementation details of the resource aware runtime and the cluster enabling mechanisms. It shows how general low-level resource accounting and thread placement can be determined by declarative policies loaded by the JVM.
2. **J.1** presents the extensions made to a JVM to support concurrent checkpoint and migration of the virtual machine and application's state. The paper also presents policies that can be used to govern checkpoint and/or migration.

3. **P.1**, **W.2** and **C.2**, describes an Economics-inspired adaptability model and the adaptation mechanisms for heap size and CPU allocation.
4. **W.3** presents a novel classification framework for adaptive resource management in system-level virtual machines.
5. **P.2** presents preliminary research on the partial utility-driven scheduling.

During the PhD program I have also published further extensions to the policy-based work presented in my MSc. dissertation. It focus on the efficient evaluation of high-level policies in the context of a location service. **W.1** and **J.2** describe the usage and extension of the Security Policy Language [61] to support a location service where location information is disclosed conditioned by the enforcement of history-based and discretionary security policies.

1.6 Outline

The rest of the document is organized in the following chapters:

- **Chapter 2 - The Adaptability Loop of Virtual Machines**. In this chapter, we present the fundamental building blocks of virtual machines which are used and extended in the literature in order to be adapted to the particularities of workloads. Next, a novel and systematic approach for the classification of adaptable resource-aware virtual machines or their components is described. Several state of the art systems are evaluated according to this classification framework.
- **Chapter 3 - Architecture and design of an adaptable and distributed execution environment**. This chapter starts by presenting the building blocks of the adaptive runtime environment. It describes each building block requisite in order to support transparent adaptation regarding the application programming model and execution. Section 3.2 presents a rationale to drive adaptability so that resources can be transferred from applications that use them poorly to the ones that can use them more efficiently. Finally, Section 3.3 delves into some relevant implementation details of the proposed mechanisms.
- **Chapter 4 - Evaluation**. This chapter starts by evaluating how effective the adaptability model is, demonstrating that the tailored allocating of resources to each application as benefits. Follows the evaluation of overheads and improvements over the baseline execution of the three adaptation mechanisms, showing their advantages.
- **Chapter 5 - Conclusions**. This chapter concludes the document discussing the overall work and some future directions which we consider as important improvements or extensions, either to the resource management models or the inner mechanisms.

2

The Adaptability Loop of Virtual Machines

2.1 Introduction

Virtual machines (VM) are being used today both at the system and programming language level. At the system level they virtualize the hardware, giving the ability to guest multiple instances of an operating system on multi-core architectures, sharing computational resources in a secure way. Regarding high level programming languages, and similarly to the system level virtual machines, these VMs abstract from the underlying hardware resources, introducing a layer that can be used for fine grained resource control. Furthermore, they promote portability through dynamic translation of an intermediate representation to a specific instruction set. High level language virtual machines (HLL-VM) are also an important building block in the organization of modern applications, due to techniques such as runtime component loading or automatic memory management.

System level VMs, or hypervisors, are strongly motivated by the sharing of low-level resources. In result of this, many research and industry work can be found about how resources are to be delivered to each guest operating system. The partition is done with different reasonings, ranging from a simple *round robin* algorithm, to autonomic behavior where the hypervisor automatically distributes the available resources to the guests that, given the current workload, can make the best out of them. Among all resources, CPU [84, 22, 64, 76] and memory [35, 78, 44] are the two for which a larger body of work can be found. Nevertheless, other resources such as storage and network are also target of adaptation.

High level language VMs have also been designed as a way to isolate and abstract from the underlying environment. Despite this middleware position, 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. There are some works about controlling system resources usage in HLL-VMs, most of them targeting the Java runtime (e.g. [28, 27, 12, 47]). They use different approaches: 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).

Virtual machines are not only a isomorphism between the guest system and a host [69], but a powerful software layer that can adapt its behavior, or be instructed to adapt, in order to transparently improve their guests's performance, minimizing the virtualization cost. In order to do so, VMs, or systems augmenting their services, can be framed into the well known adaptation loop [63]: i) monitoring or sensing, ii) control and decision, and iii) enforcement or actuation. Monitoring determines which components of the VM are observed. Control and decision take these observations and use them in some simple or complex strategy to decide what has to be changed. Enforcement deals with applying the decision to a given component/mechanism of the VM.

In both types of VMs, adaptation is accomplished at different levels. As a consequence, monitoring, control and enforcement are applied in a way that has different impacts. For example, for the allocation of processing resources, the adaptation can be limited to the tuning of a parameter in the scheduling algorithm, the replacement of the algorithm, or the migration of the guest VM to another node.

In this chapter, we present a novel framework to classify resource monitoring and adaptation techniques in virtual machines, both at system and language level, using an alternative approach to existing survey work [7, 33, 81]. Section 2.2 presents the architecture of these VMs, depicting the building blocks that are used in research regarding resource usage. Section 2.3 presents several adaptation techniques found in the literature and frames them into the adaptability loop. In Section 2.4 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 use this framework to classify state of the art systems in Section 2.5, aiming to compare and better understand the benefits and limitations of each one. Section 2.6 closes this chapter presenting some conclusions based on the previous discussion.

2.2 Virtual Machines Fundamentals

Virtual machines have their roots in the 60's with the IBM 360 and 370 [4]. These systems provided a time-sharing environment where users had a complete abstraction of the underlying hardware resources. IBM goal was to provide better isolation among different users, providing *virtual machines* to each one. The architecture of the IBM System/370 was divided in three layers: the hardware, the control program (CP) and the conversational monitor system (CMS). The CP controlled the resource provision and the CMS delivered the services to the end user

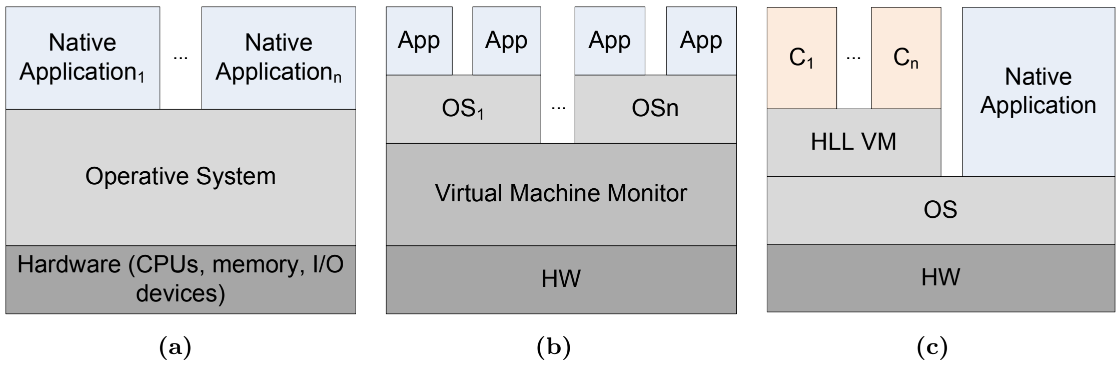


Fig. 2.1. Virtualization layers

underpinned on this resources. Today, the same architecture can be found in modern System VMs [10]. Figure 2.1 depicts these three layers, where CP's role is given to the virtual machine monitor (VMM). VMM purpose is to control the access of the guest operating systems running in each virtual machine to the physical resources, virtualizing processors, memory and I/O.

High Level Language VMs, highly influenced by the Smalltalk virtual machine [29], also provide a machine abstraction to their guest, which is an end-user application. This abstraction promotes portability in the sense that the source code is compiled not to a specific hardware but to a virtual ISA, whose running machine can be implemented in different ways by different operating systems and hardware.

The just in time (JIT) compiler is responsible for this translation and is in itself a source of adaption. Regarding its self-adaptive behavior, the JIT compiler adaptations are not driven by resource allocation but by the dynamics in the flow of execution (e.g. hot methods are compiled using more sophisticated optimizations). On the other hand, memory management has an high impact both at memory and CPU as computational resources. Research work has shown that the performance enabled by different garbage collectors algorithms is dependent on the behavior of the application as well as on the available resources (i.e. heap size). These observations motivated the development of heuristics to adapt the available heap size or the GC algorithm [55, 41, 71].

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 2.5 are based on the building blocks presented here, extending them towards self-adaptation based on resource usage.

2.2.1 Computation as a resource

In a VM, virtualization of computation concerns two distinct aspects: *i)* the translation of instructions if the guest and host use a different ISA *ii)* the scheduling of virtual CPUs to a physical CPU (or CPU core on Symmetric Multiprocessors - SMP). These two aspects have different degrees of importance in System and HLL VMs.

Instruction emulation (i.e. the translating from a set of instructions to another one) is common to both types of VMs. In System VMs, emulation is necessary to adapt different ISAs or in response to the execution of a privileged instruction (or a resource or behavior-sensitive instruction, even if not privileged) in the guest OS. Adaptation in binary, and byte code translation, is achieved by changing the translation technique (i.e. interpretation or compilation) and by replacing code previously translated with a more optimized one. These adaptations are driven by profiling information gather during program execution.

Typically HLL-VMs 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 [54] making it possible to reason about the program behavior. Regarding System VMs, because they operate directly above the hardware, the VMM must decide the mapping between the real CPUs and each running VM [10, 22]. The next section will discuss different types of algorithms to schedule VMs in physical CPUs.

System VMs scheduling

CPU scheduling is a well known issue in operating systems [74]. In single or multi-core systems, one of the operating system's task is to schedule runnable threads to a physical CPU. On a VMM running above the hardware, each guest VM is assigned one or more virtual CPUs (VCPU), whose total number can be larger than the available physical CPUs. When ready, the VCPU needs to be scheduled to a physical CPU. This results in a system with two layers of scheduling: inside the VMM and inside each guest OS.

A VMM scheduler has additional requirements when compared to the OS scheduler, namely the capacity to enforce a resource usage specified at the user's level. To achieve this, the CPU scheduler must take into account the *share* (or *weight*) given to each VM and make scheduling decisions proportional to this share [72, 22]. This family of schedulers are named *Proportional Share*. Operating systems have traditionally used a related type of share scheduling, named *Fair Share*.

In these schedulers, shares are not directly seen by the end user, making it hard to define a high level resource management policy [77]. Cherkasova et al. [22] 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. A preemptive scheduler can interrupt running VCPUs if a ready to run VCPU has a higher priority. Section 2.5 presents systems that dynamically change the scheduler parameters to give guest VMs the capacity that best fits their needs. If the scheduler cannot correctly enforce these decisions, this will lead to frequent changes of the scheduler parameters.

2.2.2 Memory as a resource

Memory is virtualized in both system and HLL-VMs with a similar goal: give the illusion to their guests of a virtually unbounded address space. Because memory is effectively limited, it will eventually become full and the guest (operating systems or application) will have to deal with memory shortage.

In System VMs, an extra level of indirection is added to the already virtualized environment of the guest operating systems. Operating systems give to their guests (i.e. processes) a dedicated address space, eventually larger than the real available hardware. As pointed out in the work [69], the VMM extra level of indirection generalizes the virtual memory mechanisms of operating systems.

In HLL-VMs, memory is requested on demand by the guest application, without the need to be explicit freed by it. When a given threshold is reached, a garbage collection activity is started to detect unreachable objects and reclaim their memory. There is no "one size fits all" garbage collector algorithm. We will next present more details about classical issues about memory management in systems and HLL-VMs.

Memory management in System VMs

The VMM can be managing multiple VMs, each with his guest OS instance and type. Therefore, the mapping between physical and real addresses must be extended because what is seen by an OS as a real address (i.e. machine address), can now change each time the VM hosting the OS is scheduled to run. The VMM introduces an extra level of indirection to the *virtual* → *real* mapping of each OS, keeping a *real* → *physical* to each of the running VMs. On the other hand, user level applications use a *virtual* address to accomplish their operations. To avoid a two folded conversion, the VMM keeps *shadow pages* for each process running on each VM, mapping *virtual* → *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* → *physical* addresses.

When the VMM needs to free memory it has to decide which page(s) from which VM(s) to reclaim. This decision might have a poor performance impact. If the wrong choice is made, the guest OS will soon need to access the reclaimed page, resulting in wasted time. 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.

Section 2.5 presents the way some relevant systems are built so that their choices are based on monitored parameters from the VM's memory utilization.

Automatic memory management in High Level Language VMs

The goal of memory's virtualization in HLL-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 clients of 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 [80, 69].

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). All these approaches strive 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 transversing all the heap. New objects are created in a smaller space (e.g. 512 KBytes). When this space fills up, live objects are promoted to a bigger space, leaving the nursery empty and ready for new allocations. These collectors are called *generational collectors*. The nursery space can be generalized and the heap organized in more than two generations.

Investigators have been analyzing the impact of different data inputs on the performance of garbage collectors [55]. Based on these observations, several adaptation strategies have been proposed [7], ranging from parameters adjustments (e.g. the nursery size [38]) to changing the algorithm itself in runtime [70]. Section 2.5 discusses these different approaches.

2.3 Adaptation techniques

In a software system, adaptation is regulated by monitoring, analyzing, deciding and acting [63]. Monitoring is fed by sensors and actions are accomplished by effectors, forming a process known as the *adaptation loop*. Virtual machines, regardless of their type, are no exception. Adaptability mechanisms are not only confined to VM's internal structures but also to systems that externally reconfigure VM's parameters or algorithms. An example of the former is the adaptive JIT compilation process of HLL-VMs [7]. An example of the latter is the work of Shao et al. [64] to regulate VCPU to CPU mapping based on the CPU usage of specific applications.

There are 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 (PID) controller and Additive-Increase/Multiplicative-Decrease (AIMD) rules to regulate one or more VM's parameters. Typically, when the analysis and decision is 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.

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

2.3.1 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 the 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.

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(s) will run next, and assign it to a CPU [10, 22]. 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) [84, 64], that use their own control strategy, such as linear optimization, control theory using a proportional-integral-derivative (PID) controller [60] or signal processing and statistical learning algorithms [34]. Nevertheless, such systems act on mechanisms inside the hypervisor (e.g. weight assigned to VMs, number of VCPUs).

Memory Management

The memory manager virtualizes hardware pages and determines how they are mapped to each VM. To establish which and how many pages each VM is using, the VMM can monitor page's utilization using either whole page or sub-page scope. The monitoring activities aim to reveal how pages are being used by each VM, and so information collected relates to: *i)* page utilization [78, 79], and *ii)* page contents equality or similarity [78, 10]. Application performance (either by modification of the application or external monitoring) is also considered [44].

Because operating systems do not support dynamic changes to physical memory, the maximum amount of memory that can be allocated is statically assigned to each VM. Nevertheless,

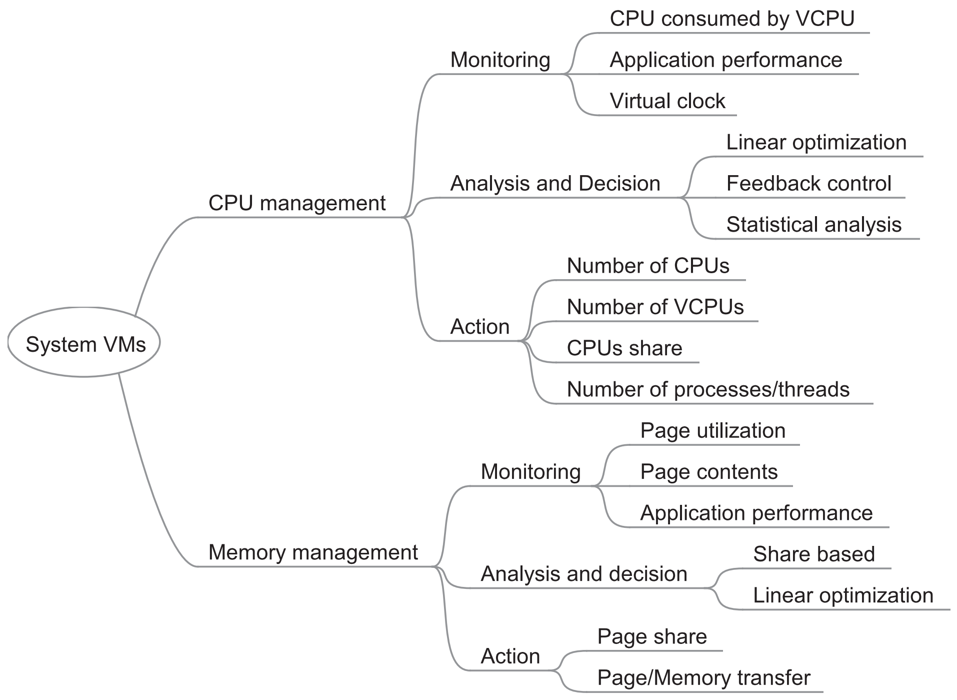


Fig. 2.2. Techniques used by System VMs to monitor, control and enforce

when total allocated memory exceeds the one that is physically available, the VMM must decide which clients must relinquish their allocated memory pages in favor of the current request. Decisions regarding memory pages allocation to each VM are made using: *i)* shares [78], *ii)* history pattern [79], or *iii)* linear programming [44].

After deciding that a new configuration must be applied to a set of VMs, the VMM can enforce: *i)* page sharing [78], or *ii)* page transfer between VMs. Page sharing relies on the mechanisms that exist at the VMM layer to map *real* $\rightarrow$ *physical* page numbers, as described in Section 2.2.2. On the other hand, the page transfer mechanism relies on the operating systems running at each VM, so that each operating system can use its own paging policy. This is accomplished using a balloon driver installed in each VM [10, 78].

Summary of adaptation loop techniques

Figure 2.2 presents the techniques used in the adaptation loop. They are grouped by the two major adaptation targets, CPU and memory, and then into the three major phases of the adaptability 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 [78] and Barham et al. [10] laid solid techniques for virtualizing and managing this resource. Recent work shows that, to improve perform of workloads regarding their use of memory is crucial to have more application-level information [79, 44].

2.3.2 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 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. Next we will analyze the most common strategies.

Just in time compilation

The JIT is mostly self contained in the sense that the monitoring process (also know 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 [3] or *ii)* instrumentation. In the first case, the execution stacks are periodically observed to determine hot methods. In the second case, method's code is instrumented so that its execution will fill the appropriate runtime profiling structures. Sampling is known to be more efficient [7] 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. 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 [6, 7]. 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 [3].

Adaptability techniques in the JIT compiler are used to produce native optimized code while minimizing impact in application's execution time. 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. So, the actions that can complete the adaptability loop are: *i)* partial or total method recompilation, *ii)* inlining or *iii)* deoptimization.

[1] The adaptive optimization system (AOS) in Jikes RVM [3] produces a log with approximately $700 Kbytes$ of information regarding call graphs, edge counters and compilation advices when running and JIT compiling one of DaCapo's benchmark [13] - `bloat`

Garbage collection

Tradicional GC algorithms are not adaptive in the sense that the strategy to allocate new objects, the kind of spaces used to do so, and the way garbage is detected, does not change during program's execution. Nevertheless, most research and comercial runtimes incorporate some form of adaptation strategy regarding memory management [7]. To accomplish these adaptations, monitoring is done by observing: *i)* memory structures dimensions (e.g. total heap size, nursery size) [67, 68], *ii)* the program behavior (e.g. allocation rate, stack height, key objects) [70] and, *iii)* relevant events in the operating systems (e.g. page faults, allocation stalls) [37, 43].

Decision regarding the adaptation of heap related structures are taken either *i)* offline or *ii)* inline with execution. Offline analysis takes in consideration the result of executing different programs so see which parameter or algorithm has the best performance for a given application. Inline decisions must be taken either based on a mathematic model or on some kind of heuristic. Some authors have elaborated mathematical models of objects' lifetimes. These models are mostly used to give a rationale of the GC behavior, rather than being used in a decision process [9]. So, most systems have a decision process based on some kind of heuristics. The decision process includes: *i)* machine learning, *ii)* PID controller, and *iii)* microeconomic theories such as the elasticity of demand curves.

Similarly to the JIT compiler, adaptability regarding memory management aims to improve overall system performance. Classic GC algorithms provide base memory virtualization. Recent works have been focused on optimizing memory usage and execution time, taking in consideration not only the program dynamics and but also the state of the execution environment [41]. Some work also adapts GC to avoid memory exhaustion in environments where memory is constrained [70]. To accomplish this, actions regarding GC adaptability are related to changing: *i)* heap size [67], *ii)* GC parameters [68] *iii)* GC algorithm [70].

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 [8, 27], or *ii)* byte code instrumentation [47]. In the former, it is possible to collect more information, both from a quantitative and qualitative perspective. A well know example is the Java Virtual Machine Tool Interface [58], which is mainly used by development environments to display debug information. 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 system objects like threads or files.

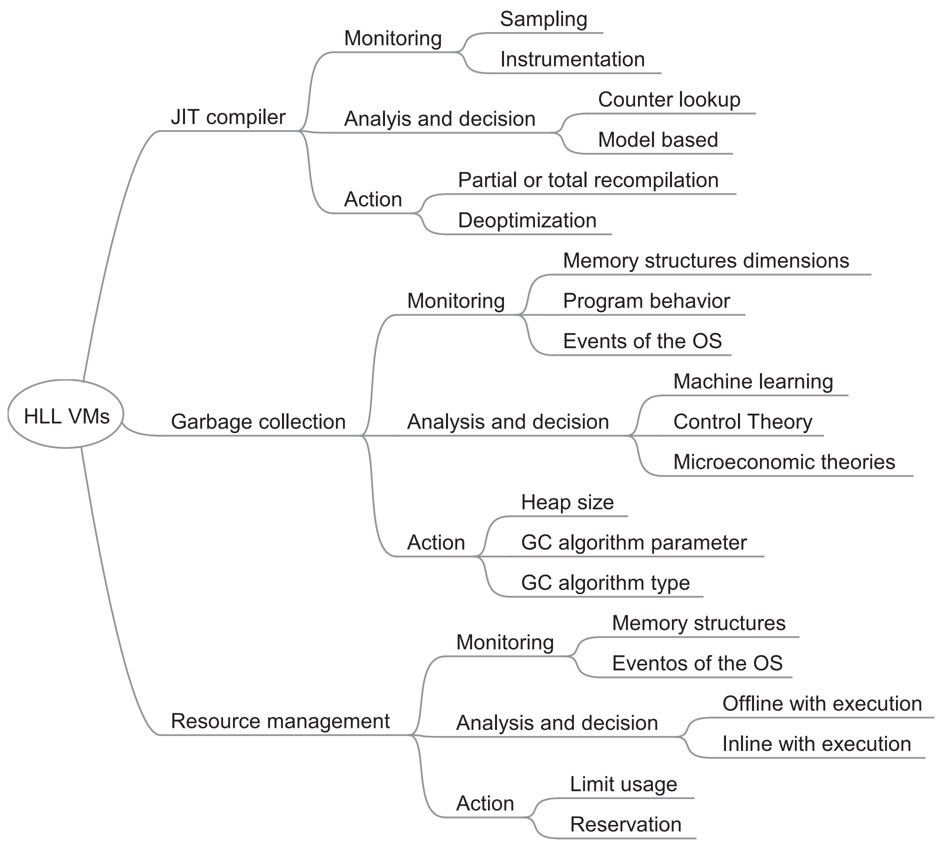


Fig. 2.3. Techniques used by HLL-VMs to monitor, control and enforce

This subsystem has to decide if 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 a internal policy, the reasoning is hard coded in the runtime, eventually 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 runtime. Research and commercial systems apply resource management actions to: *i)* limit resource usage and *ii)* resource reservation. Limiting resource usage aims to avoid denial of service, or to ensure that the (eventually payed) resource quota is not overused. The last scenario is less explored in the literature [27]. Resource reservation ensures that, when multiple processes are running in the same runtime, it's possible to ensure a minimum amount of resources to a given process.

Summary of adaptation loop techniques

Figure 2.3 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, and *iii)* resource management. Each adaptation target is then divided into the three phases of the adaptability loop. The garbage collection sub-tree has a higher number of elements when compared 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).

2.4 The RCI Framework for classification of VM's adaptation techniques

To understand and compare different adaptation processes we now introduce a framework for classification of VM's adaptation techniques. It addresses the three classical adaptation steps. Each of this steps makes use of the different techniques described earlier and depicted in Figure 2.2 and Figure 2.3.

The analysis and classification of the techniques for each of these steps revolves around three fundamental criteria: *Responsiveness*, *Comprehensiveness* and *Intricateness*. We call it `RCI framework`. *Responsiveness* represents how fast the system is able to adapt, thus it gets smaller as the following metrics increase: i) overhead of monitoring, ii) duration of the decision process, iii) the latency of applying adaptation actions.

Comprehensiveness takes into account the breadth and scope of the adaptation process. It gets greater as the following metrics increase. In particular, it regards: i) the quantity or quality of the monitored sensors, ii) the easiness to relate the decision process with the underlying system, and iii) the quantity or quality of the effectors that the system can engage.

Finally, *Intricateness* addresses the depth of the adaption process. In particular, it regards low-level implications, interference and complexity of: i) the monitoring sensors, ii) decision strategy, and iii) the enforcing sensors.

These aspects were chosen, not only because they encompass many of the relevant goals and challenges in VM adaptability research, but mainly because they also embody a fundamental underlying tension: *that a given adaptation technique aiming at achieving improvements on two of these aspects, can only do so at the cost of the remaining one*. We came across this observation during the process of analyzing and classifying the techniques and systems studied.

Initially, we realized that no technique was able to combine full comprehensiveness and full intricateness, and still be able to perform without significant overhead and latency (possibly even requiring off-line processing). Full responsiveness, for example, will potentially always implies some level of restriction either to comprehensiveness or to intricateness. This *RCI conjecture* is yet another manifestation in systems research of where the constant improvement on a given set of properties, or the behavior of a given set of mechanisms, can only come at an asymptotically increasing cost. This always forces designers to choose one of them to degrade in order to ensure the other two.

A paramount example is the CAP conjecture (or CAP theorem) [18], portraying the tension in large-scale distributed systems among (C)onsistency, (A)vailability, and tolerance to (P)artitions. Another example is the tension, in the domain of peer-to-peer systems, among high availability, scalability, and support for dynamic populations [16].

Additionally, we also note that the tension inherent in the RCI conjecture is also present, at a higher-level of abstraction, among monitoring, decision, and action. The more the emphasis (regarded as an aggregate value of all RCI aspects) is given to two of the steps in the control loop, the less emphasis is possible to the remaining one, without breaking the viability and feasibility of the approach. We call this derived conjecture that applies to whole systems (and not to individual adaptation techniques) the MDA conjecture, for *Monitoring*, *Decision* and *Action*.

In order to quantitatively compare different systems (e.g. more responsive or more comprehensive), each of the previously discussed metrics must be assigned with a quantitative value, which depend on the analyzed adaptation technique. Table 2.4 presents the nature of these metrics.

	Responsiveness	*Comprehensiveness*	*Intricateness*
Monitor	ISL	Q	SL
Decision	PT	Q	IC
Action	ISL	Q	SL

Table 2.1. Quantitative units of the classification metrics

Table 2.4 shows the meaning of each metric for each of the quantitative values that the framework allows techniques to be classified (1, 2 or 3). Quantitative (Q) intervals, Processing Times (PT), *System level* (SL), *Inverse system level* (ISL) and Intrinsic Complexity (IC) used in the framework are presented. *System level* (SL) represents the natural organization of a computer system, assigning 1 to hardware, 2 to OS and hypervisor and 3 to applications. *Inverse system level* (ISL) uses this scale in reverse order so that the term Responsiveness can be understood as described previously. Second, Regarding the decision characterization of the control step we adopted the criteria of Maggio et al. [53].

To better understand how the framework is used, hypothetical techniques ($T_a..T_f$) are presented in Table 2.3. After having a classification of each technique the framework builds the RCI of a system by aggregating each criteria' value. For a given system, S_α, the three criteria of the framework, responsiveness, comprehensiveness and intricateness, are represented

Level	1	2	3
Q	[1..2]	[3..4]	[4..N]
SL	hardware	hypervisor/OS	application
ISL	application	hypervisor/OS	hardware
PT	milliseconds	seconds	minutes
IC	simple	medium	complex

Table 2.2. Relation between classification levels (on top) and classification metrics

by $R(S_\alpha)$, $C(S_\alpha)$, $I(S_\alpha)$, respectively. The corresponding criteria of each technique (t) used by S_α is summed (e.g. $R(S_\alpha) = \sum_t responsiveness(t)$).

Phase	*Tecnhique*	*Responsiveness*	*Comprehensiveness*	*Intricateness*
Monitor	*Ta*	1	2	3
	Tb	2	3	1
Decision	*Tc*	3	2	3
	Td	1	1	2
Action	*Te*	2	3	1
	Tf	1	2	1

Table 2.3. Hypothetical techniques and their quantification

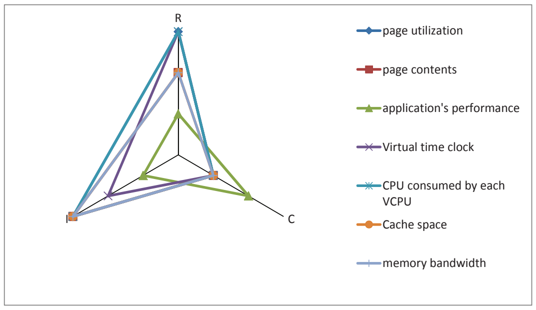
Using the mock techniques presented in Table 2.3, Table 2.4 presents, in the bottom row, the resulting RCI of S_α. Furthermore, the table also presents, in the most right column, the MDA characteristic of S_α.

System	*Phase*	*Responsiveness*	*Comprehensiveness*	*Intricateness*	*MDA*
S_α	Monitor	Ta(1)	Ta(2)	Ta(3)	6
	Decision	Tc(3)	Tc(2)	Tc(3)	8
	Action	Tf(1)	Tf(2)	Tf(1)	4
RCI		5	6	7	

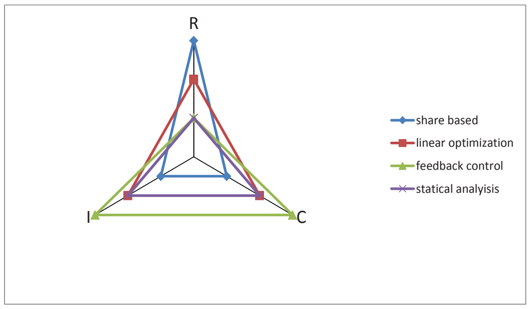
Table 2.4. RCI and MDA of hypothetical system S_α

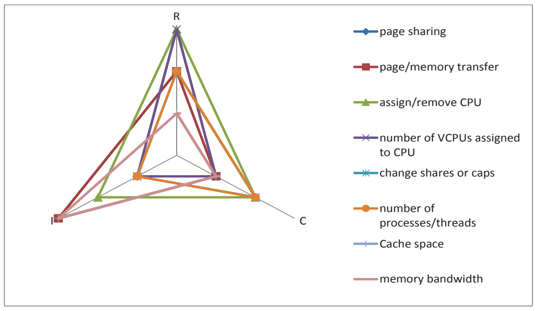
Figures 2.4 and 2.5 use a triangular chart to represent the techniques previously addressed in Section 2.3, regarding both system and high level virtual machines (see Figure 2.2 and 2.3). In each main figure, techniques are further categorized into the three phases of the adaptation loop - (a) monitoring, (b) decision, and (c) action.

In the next section, we analyze relevant works regarding monitoring and adaptability in virtual machines, both at system as well as managed languages level. The RCI framework



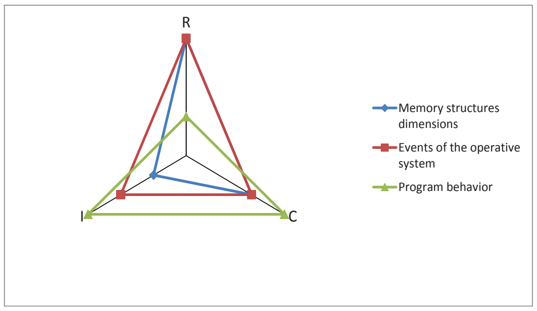
(a)



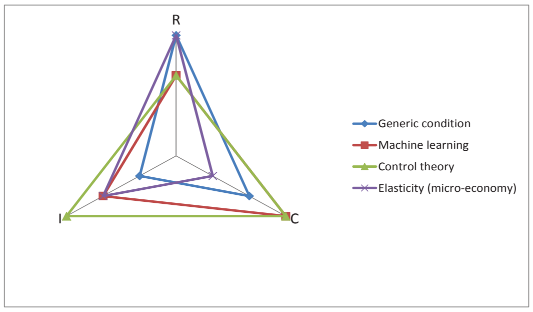


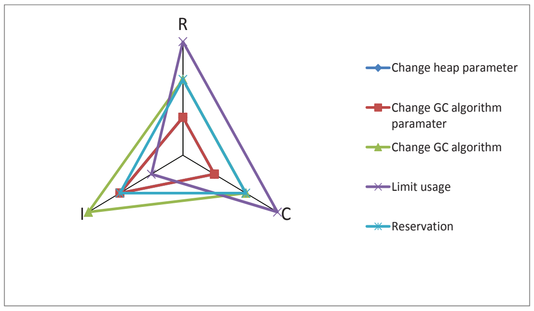
(b) **(c)**

Fig. 2.4. Relation of *responsiveness*, *comprehensiveness* and *intricateness* for the different techniques used in System VMs



(a)





(b) **(c)**

Fig. 2.5. Relation of *responsiveness*, *comprehensiveness* and *intricateness* for the different techniques used in HLL VMs

is used to compare different systems and better understand how virtual machine researchers have explored the tension between responsiveness, comprehensiveness and intricateness.

2.5 Systems and their classification

The first two sections survey several state of the art systems. The last section frames them into the classification framework presented in Section 2.4, classifying adaptability techniques and, afterwards, complete systems.

2.5.1 System Virtual Machine

Xen. In Xen [10] each VM is called a *domain*. A special *domain0* (called *driver domain*) handles I/O requests of all other domains (called *guest domain*) and runs the administration tools. Because Xen's core solution is developed by the open source community, several works have studied Xen's scheduling strategies, for example in face of intensive I/O. Others propose adaptation strategies to be applied by the VMM regarding CPU to VCPU mapping or dynamically changing the scheduling algorithms parameters.

Xen includes three scheduling algorithms: Borrow Virtual Time (BVT), Simple Earliest Deadline First (SEDF) and Credit[2] [22]. The former two are deprecated and will probably be removed. Credit is a proportional fair scheduler. This means that the interval of time allocated for each VCPU is proportional to its *weight*, excluding small allocation errors. Additionally to *weight*, each *domain* has a *cap* value representing the percentage of extra CPU it can consume if its quantum has elapsed and there are idle CPUs. At each clock tick, the running VPCUs are charged and eventually some will loose all their *credit* and tagged as *over*, while the others are tagged *under*. VCPUs tagged as *under* have priority in scheduling decisions. Picking the next VCPU to run on a given CPU, Credit looks, in this order for an *under* VCPU from the local running queue, an *over* VCPU from the local running queue or an *under* VCPU from the running queue of a remote CPU, in a work-stealing inspired fashion.

Friendly Virtual Machines (FVM). The Friendly Virtual Machines (FVM) [84] aims to enable efficient and fair usage of the underlying resources. Efficient in the sense that underlying system resources are neither overused nor underused. Fairness in the sense that each VM gets a proportional share of the bottleneck resource. Each VM is responsible for adjusting its demand of the underlying resources, resulting in a distributed adaptation system.

The adaptation strategy is done using feedback control rules such as Additive-Increase/Multiplicative-Decrease (AIMD), typically used in network congestion avoidance [23], driven by a single control signal - the *Virtual Clock Time* (VCT) to detect overload situations. VCT is the real

[2] http://wiki.xen.org/wiki/Credit_Scheduler, visited at 6-03-2013

time taken by the VMM to increment the virtual clock of a given VM. An increase in VCT means that the host VMM is taking longer to respond to the VM which indicates a contention on a bottlenecked resource. Depending on the nature of the resource, the VCT will evolve differently as more VMs are added to the system. For example, with more VMs sharing the same memory, more page faults will occur, and even a small increase in the number of page faults will result in a significant increase in VCT.

A VM runs inside a hosted virtual machine, the User Mode Linux, an so, two types of mechanisms are used to adapt VM's demand to the available underlying resources. FVM imposes upper bounds on: *i)* the Multi Programming Level (MPL), and on *ii)* the rate of execution. MPL 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.

HPC computing. Shao et al. [64] adapt the VCPU mapping of Xen [10], based on runtime information collected by a monitor that must be running inside each guest's operating system. They adjust the numbers 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 native applications representative of high performance computing applications.

Ginko. Ginko [44] is an application-driven memory overcommitment framework which allows cloud providers to run more System VMs with the same memory. For each VM, Ginkgo uses a profiling phase where it collects samples of the application performance, memory usage, and submitted load. Then, in production phase, instead of assigning the same amount of memory for each VM, Ginko takes the previously built model and, using a linear program, determines the VM ideal amount of memory to avoid violations of service level agreements. This means that the linear program will determine the memory allocation that, for the current load, maximizes the application performance (e.g. response time, throughput).

Auto Control. Padala et al. [60] propose a system which uses a control theory model to regulate resource allocation, based on multiple inputs and driving multiple outputs. Inputs are applications running in a VMM and can spawn several nodes of the data center (i.e. web and DB tier can be located in different nodes). Outputs are the resource allocation of CPU and disk I/O caps. For each application, there is an application controller which collects the application performance metrics (e.g. application throughput or average response time) and, based on the application's performance target, determines the new requested allocation. Because computational systems are non linear, the model is adjusted automatically, aiming to adapt to different operating points and workloads. Based on each application controller output, a per node controller will determine the actual resource allocation. It does so by solving the optimization problem of minimizing the penalty function for not meeting the performance targets of the applications. To evaluate their system, applications were instrumented to collect performance statistics. Xen monitoring tool (i.e. `xm`) was used to collect CPU usage and `iostat` was used to collect CPU and disk usage statistics. Enforcement is made by changing Xen's credit scheduler parameters and a proportional-share I/O scheduler [39].

PRESS. PRESS [34] is an online resource demand prediction system, which aims to handle both cyclic and non-cyclic workloads. It tries to allocate just enough resources to avoid service level violations while minimizing resource waste. PRESS tracks resource usage and predicts how resource demands will evolve in the near future. To detect repeating patterns it employs signal processing techniques (i.e. Fast Fourier Transform and the Pearson correlation), looking for a signature in the resource usage history. If a signature is not found, PRESS uses a discrete-time Markov chain. This technique allows PRESS to calculate how the system should change the resource allocation policy, by transitioning to the highest probability state, given the current state. The work in [34] the authors focus on CPU usage. Thus, the prediction scheme is used to set the CPU cap of the target VM. The evaluation was made based on a synthetic workload applied to the RUBiS benchmark, built from observations of two real world workloads.

VM^3. The work in VM^3 [49] aims at measuring, modeling and managing shared resources in virtual machines. It operates in the context of virtual machine consolidation in cloud scenarios proposing a benchmark (vConsolidate). It places emphasis on balancing quickness of adaptation, and the intricateness and low-level of the resources monitored, while sacrificing comprehensiveness, by being restricted to deciding migration of virtual machines among cluster nodes.

2.5.2 High Level Language Virtual Machines

Adaptation in high language virtual machines is made changing their building blocks parameters (e.g. GC heap size) or the actual algorithm used to perform certain operations. The cycle of adaptation begins with acquiring the usage of the relevant resources. Acquiring has always a cost that should be minimized using either low impact operations or resource counters already available in the system to accomplish other tasks. After collecting this information, the VM can either restrain usage or make adaptations to the building blocks. This section will present and discuss different strategies related to monitoring resources, controlling usage and adaptations policies in HLL VMs.

Two approaches have been used to collect resource usage information, one that relies on the VM privileged connection to the operating system, and runtime libraries contribution and another one which is independent of the VM platform and uses byte code instrumentation or transformation.

Aroma. Accounting for CPU usage is done inside the bytecode interpreter and is specified by the number of instructions allowed to execute in a certain time interval. Before each bytecode is executed, Aroma checks if the number of bytecodes per interval, previously calculated, have already been executed. If so, the interpreter goes into a passive wait until the remaining time of the interval elapses.

KaffeOS. Built on top of Kaffe virtual machine [1], KaffeOS [8] provides the ability to run Java applications isolated from each other and also to limit their resource consumption. FKaffeOS, 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. [28] 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 and when new threads are created. The only resources supported are the CPU usage (in miliseconds), the total amount of used memory (in bytes) and the number of bytes sent and received through a network interface. CPU time is accounted by instrumenting the run method of each new thread, placing the native thread identification in a global registry. Then, at regular intervals, the registry is traversed and native calls are used to ask the operating system for the time spent in each thread. Byte code rewriting is also used to know how much memory is used by objects allocated by each thread.

Multitask Virtual Machine (MVM). The MVM [27] 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 [27] 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 bounded 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. CPU accounting is done in a similar way to JRES [28] using native calls to the operating systems. On the other hand, memory accounting was done modifying the memory management system.

J-RAF2. Hulaas et al. [47] uses an instrumentation only solution to account for resources. Hulaas et al. discuss the limitations and overheads of their previous work (J-SEAL2), regarding CPU accounting, and present some techniques to optimize this process. Analysis of their first proposed transformation algorithm shows that most of the overhead is associated to finding the proper instruction counter, and to the frequents updates of this counter in each method. To minimize these overheads, J-RAF2 uses the following strategies. First, it changes every method's signature to receive the CPU accounting object, which is created and first used when the thread starts. Second, they design and implemented a new path prediction scheme to reduce the number of updates. The algorithm works by trying to predict, during the bytecode transformation phase, the outcome at runtime of the conditional branches. For example, instead of accounting all blocks inside a loop, they predict what will the runtime path be and add this path cost only once per loop. If the execution flow takes a different path (resulting in a miss prediction) account will be compensated by decreasing the non executed part of the composed block and the cost of the new path.

Lightweight VMs. Duran et al. [30] take a middleware-oriented approach by using a thin high-level language virtual machine to virtualize CPU and network bandwidth. Their goal is to provide an environment for resource management, that is, resource allocation or adaptation. Applications targeting this lightweight VM use a special purpose programming interface to specify reservations and adaptation strategies. When compared to more heavyweight approaches like System VMs, this lightweight framework can adapt more efficiently for I/O intensive applications. The approach taken in Duran's work, bounds however, the application to a given resource adaptation interface, raising transparency issues.

Garbage collection is known to have different performance impacts in different applications [70, 55]. The remainder of this section analyzes recent works belonging to one of the following categories: *i)* adjust heap related parameters (e.g. nursery size, total heap size) [41, 37, 67]; *ii)* algorithms that take execution environment events into account [42]; *iii)* VMs that switch the GC algorithm at runtime [70]. Common to all these solutions is the goal to decrease application's total execution time or, in some scenarios, to continue operation despite memory exhaustion.

GC and the allocation stalls. Grzegorczyk et al. [37] takes into account *allocation stalls.* In Linux, a process will be stalled during the request of a new page if the system has very few free memory pages. If this happens, a resident page must be evicted to disk. This operation is done synchronously during page allocation. They have implemented an algorithm that grows the heap linearly when there are no allocation stalls. Otherwise, the heap shrinks and the growth factor for successive heap growth decisions is reduced, in an attempt to converge to a heap size that balances the tradeoff between paging and GC cost.

GC in shared environment. Hertz et al. [43] observe that the same application operating with different heap sizes can perform differently if the heap size is under or over dimensioned, resulting in many collections or many page faults, respectively. Based on this observation they have devised the time-memory curve, that is, the shortest running time of a program independently of his heap size for a given amount of physical memory. Their approach allows that the heaps of multiple applications remain small enough to avoid the negative impacts of paging, while still taking advantage of any memory that is available within the system. They have modified the slow path of the GC (i.e. code path that can result in tracing alive objects) to also take into account two conditions: *i)* if the resident set has decreased or, *ii)* if the number of page faults have increased. If any of these conditions is true a GC will be triggered. They call this situation a *resource-driven* garbage collection.

GC in a MapReduce environment. Singer et al. [68] propose to automatically change the GC configuration, in order to improve the performance of a MapReduce's Java implementation for multi-core hardware. For each relevant benchmark, machine learning techniques are used to find the best execution time for each combination of input size, heap size and number of threads in relation to a given GC algorithm (i.e. serial, parallel or concurrent). Their goal is to make a good decision about a GC policy when a new MapReduce application arrives. The decision is made locally to an instance of the JVM.

GC economics. In [67], Singer et al. discuss the economics of GC, relating heap size and number of collections with the price and demand law of micro-economics - with bigger heaps there will be less collections. This relation extends to the notion of elasticity to measure the sensitivity of the heap size to the size of the number of GCs. They devise an heuristic based on elasticity to find a tradeoff between heap size and execution time. The user of the VM provides a target elasticity. During execution, the VM will take into account this target to grow, shrink or keep the heap size. Doing so, the user can supply a value that will determine the growth ratio of the heap, independently of the application specific behavior.

GC switch. Soman et al. [70] add to the memory management system the capacity of changing the GC algorithm during program execution. The system considers program annotations (if available), application behavior, and resource availability, in order to decide when to switch dynamically, and to which GC it should switch to. The modified runtime incorporates all the available GC algorithms into a single VM image. At load time all possible virtual memory resources are reserved. The layout of each space (i.e. nursery, Mark-Sweep, High Semispace, Low Semispace) is designed to avoid a full garbage collection for as many different switches as possible. For example, a switch from Semi-Space to Generational Semi-Space determines that the allocation site will be done at a nursery space, but the two half-spaces are shared. Switching can be triggered by points statically determined by previous profiling the application execution, or by dynamically evaluating the GC load versus the application threads. If the 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).

2.5.3 Quantitative comparison of different adaptability techniques

In Figures 2.6 and 2.7, we analyze the *responsiveness*, *comprehensiveness* and *intricateness* aspects for the different adaptability techniques used in the two types of VMs. The global observation is that different systems have a different RCI coverage, which is the result of using diverse adaptations techniques. Regarding the tension described by the RCI conjecture, we note that, in system VMs, intricateness seems to dominate but responsiveness is also strong, while in HLL VMs, responsiveness and comprehensive seem to dominate over intricateness. Also in system VM deployments, they exhibit larger responsiveness and intricateness but are less comprehensive. In HLL-VMs intricateness is larger in deployments focused on GC. All this reinforce the observation and conjecture that the three properties cannot be covered to the same degree.

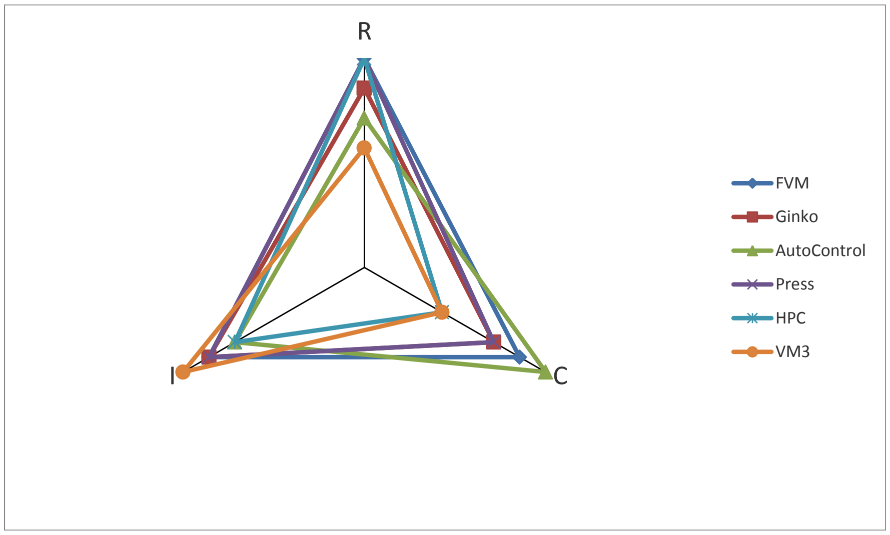


Fig. 2.6. Relationship of *responsiveness*, *comprehensiveness* and *intricateness* for the different adaptability techniques used in System VMs.

In Figures 2.8 and 2.9, we analyze the adaptation loop for the two types of VMs. The three phases, *monitoring*, *decision* and *action*, are quantitatively compared using the *responsiveness*,

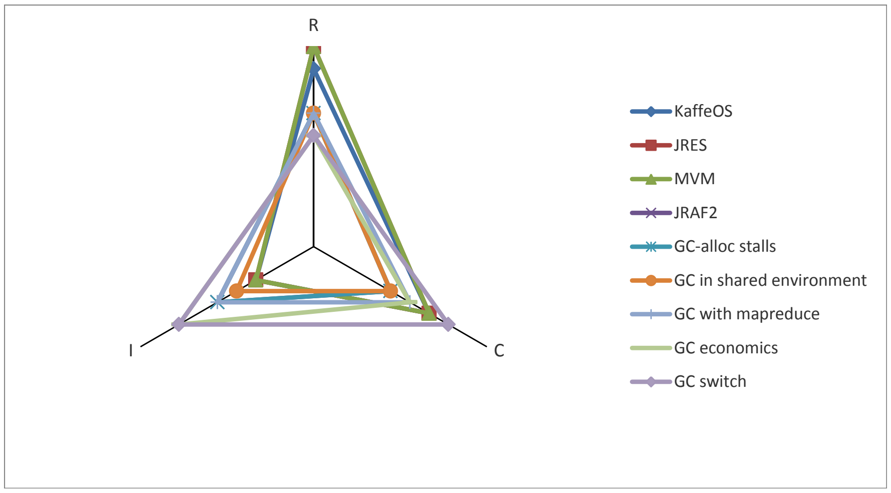


Fig. 2.7. Relationship of *responsiveness*, *comprehensiveness* and *intricateness* for the different adaptability techniques used in HLL VMs.

comprehensiveness and *intricateness* of the techniques used in each step. When analyzing the overall results, we can see that HLL-VMs use more uniform monitoring techniques, decision techniques in Sys-VMs are more alike and effectors used in the action phase are more uniform in Sys-VMs.

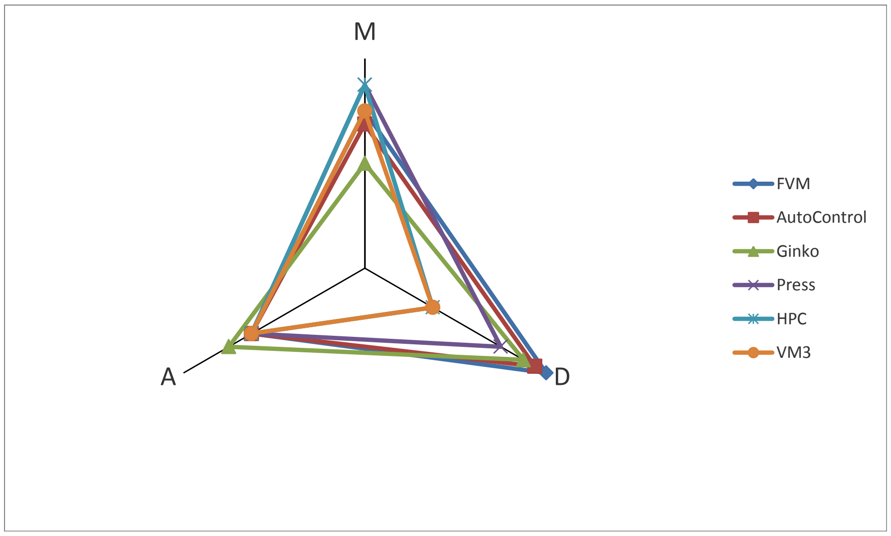


Fig. 2.8. Relationship of emphasis on *responsiveness*, *comprehensiveness* and *intricateness*, regarding the adaptation steps *monitoring*, *decision* and *action* in System VMs.

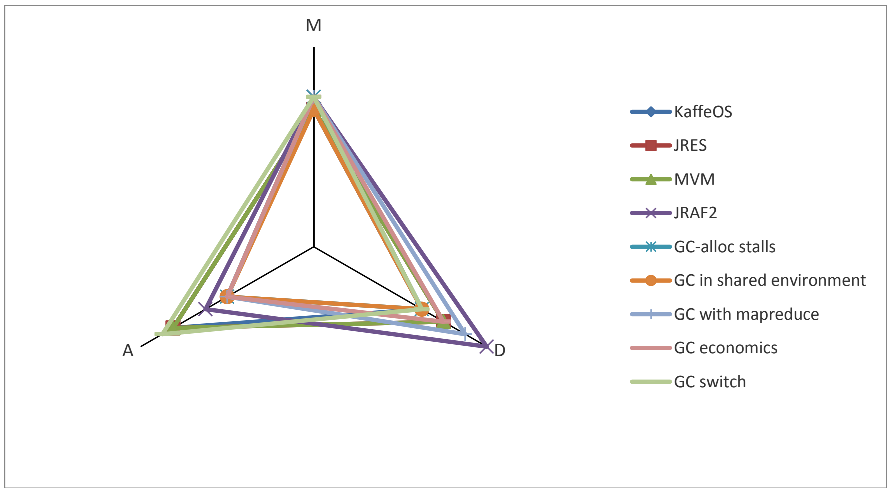


Fig. 2.9. Relationship of emphasis on *responsiveness*, *comprehensiveness* and *intricateness*, regarding the adaptation steps *monitoring*, *decision* and *action* in HLL VMs.

2.6 Summary

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 control loop (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. Then we presented a derived conjecture, the MDA conjecture identifying a related tension, in the context of whole systems, among emphasis on monitoring, decision and action.

3

Architecture and design of an adaptable and distributed managed execution environment

3.1 Architecture of the QoE-JVM

Our vision is that *QoE-JVM* will execute applications with different requirements regarding their QoE. Target applications have typically a long execution time and can dynamically spawn several execution flows to parallelize their work. This is common in the field of science supported by informatics like economics and statistics, computational biology and network protocols simulation [36, 82, 40].

Figure 3.1 presents the overall architecture of our distributed *platform as a service* for Cloud environments. *QoE-JVM* is supported by several runtime instances, eventually distributed by several computational nodes, each one cooperating to the sharing of resources. For an effective resource sharing, a coordinated mechanism must be in place to make weak (e.g. change parameters) or strong (e.g. change GC algorithm, migrate running application) adaptations [63]. *QoE-JVM* encompasses a distributed shared objects middleware, a reconfigurable high-level language virtual machine (HLL-VM), and, at the bottom, available reconfigurable mechanisms of system level virtual machine (Sys-VM). In this architecture, the operating system (OS) services are only used, not extended.

3.1.1 A cluster-wide execution environment

We consider a *cluster* as a typical aggregation of a number of *nodes* which are usually machines with one or more multi-core CPUs, with several GB of RAM, interconnected by a regular LAN network link (100 Mbit, 1 Gbit transfer rate). We assume there may be several applications, possibly from different users, running on the cluster at a given time, i.e., the cluster is not necessarily dedicated to a single application. The cluster (or each fraction of it) has one top-level coordinator, the *QoE Manager* that monitors the *Quality-of-Execution* of applications.

Each *node* is capable of executing several instances of a Java VM, with each VM holding part of the data and executing part of the threads of an application. As these VMs may compete for the resources of the underlying cluster node, there must be a *node manager*

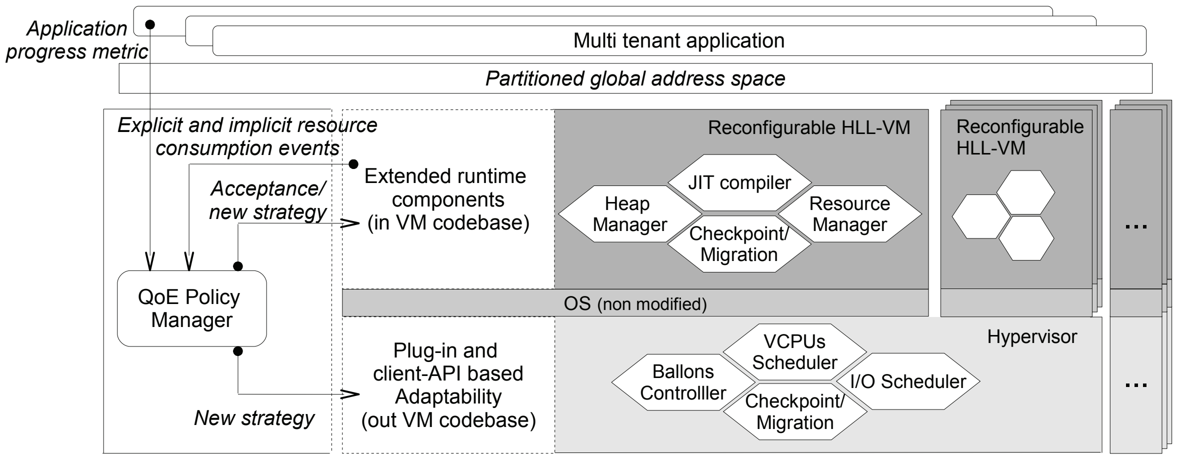


Fig. 3.1. Overall architecture

in each node, in charge of VM deployment, lifecycle management, resource monitoring and resource management/restriction. In order for the node and cluster manager to be able to obtain monitoring data and get their policies and decisions carried out, the Java VMs must be resource-aware, essentially, report on resource usage and enforce limits on resource consumption. Finally, cooperation among VMs is carried out via the *QoE Manager*, that receives information regarding resource consumption in each VM, by each application, and instructs VMs to allow or restrict further resource usage.

Each instance of an HLL-VM is enhanced with services that are not available in regular VMs. These services include: i) the accounting of resource consumption, ii) dynamic reconfiguration of internal parameters and/or mechanisms, and iii) mechanisms for checkpointing, restore and migration of the whole application. These services should and must be made available at a lower-level, inside an extended HLL-VM, for reasons of control, interception and efficiency. They are transparent to the application, and so, the extended HLL-VM continues to run existing applications as they are.

In summary, the responsibilities of each of these entities are the following, as depicted in Figure 3.2:

- Cluster QoE Manager
  - collect global data of cluster applications (i.e. partitioned across VMs and nodes)
  - deploy/regulate nodes based on user's QoE
- Node QoE Manager
  - report information about node load
  - deploy new policies on VMs
  - create or destroy new instances
  - collect VM's resource usage data
- (resource-aware) HLL-VM

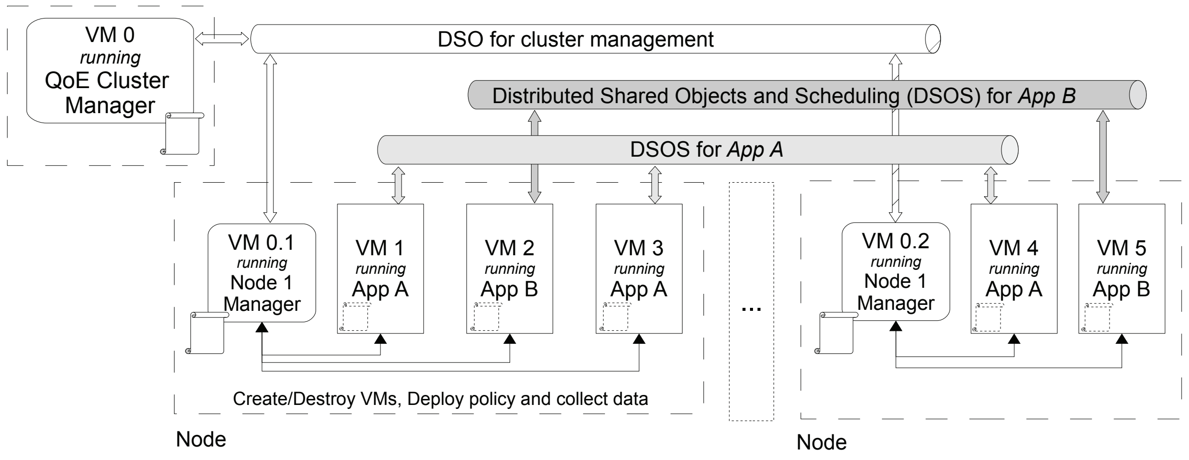


Fig. 3.2. A cluster-wide execution environment

- enforce resource usage limits
- give internal information about resource usage
- concurrent checkpoint and migration of execution state

QoE-JVM is thus comprised of several components, or building blocks. Each one gives a contribution to support applications with a global distributed image of a virtual machine runtime, where resource consumption and allocation are driven by a high-level policies, system-wide, application or user related. From a bottom-up point of view, the first building block above the operating system in each node is a process-level managed language virtual machine, enhanced with mechanisms and services that are not available in regular VMs. These include the accounting of resource consumption.

The second building block aggregates individual VMs, as the ones mentioned above, to form within the cluster a distributed shared object space assigned to a specific application. It gives running applications support for single system image semantics, across the cluster, with regard to the object address space, enabling a truly elastic heap for applications. Techniques like bytecode enhancement/instrumentation or rewriting must be used, so that unmodified applications can operate in a partitioned global address space, where some objects exist only as local copies and others are shared in a global heap.

The third building block turns *QoE-JVM* into a cluster-aware cooperative virtual machine. This abstraction layer is responsible for the global thread scheduling in the cluster, starting new work items in local or remote nodes, depending on a cluster wide policy and the assessment of available resources, possibly taking into account application-related information (profiling past executions). Mechanisms for checkpointing, restore and migration can also be activated in order to migrate whole instances of VMs to a different node. This layer is the *QoE-JVM* boundary that the cluster-enabled applications interface with (note that for the applications, the cluster looks like a single, yet much *larger* and larger, virtual machine). Similarly to the previous block, application classes are further instrumented/enhanced (although the two sets of instrumentation can be applied in a single phase), in order to guarantee correct behavior in the cluster. Finally, it exposes the underlying mechanisms to the adaptability policy engine, and accepts external commands that will regulate how the VM's internal mechanisms should behave.

The resource-aware HLL-VM, the distributed shared object layer, and the cluster level scheduler are all sources of relevant monitoring information to the policy engine of *QoE-JVM* . This data can be used as input to declarative policies in order to determine a certain rule outcome, i.e. what action to perform when a resource is exhausted or has reached its limit, regarding a user or application. The other purpose of collecting this data is to infer a profile for a given application. Such profiles will result in the automatic use of policies for a certain group of applications, aiming to improve their performance. The effects, positive or negative, of applying such policies are then used to confirm, or reject, the level of correlation between the profile and the applications.

On top of this distributed runtime are the applications, consuming resources on each node and using the services provided by the resource-aware VM that is executing on each one. *QoE-JVM* targets mainly applications with a long execution time and that may spawn several threads to parallelize their work, as usual in e-Science fields such as those mentioned before.

3.1.2 Resource Awareness and Control

The Resource Aware virtual machine is the underlying component of the proposed infrastructure. It has two main characteristics: *i)* resource usage monitoring, and *ii)* resource usage restriction or limitation. Current virtual machines for managed languages can report about several aspects of their internal components, like used memory, number of threads, classes loaded [58, 57]. However they do not enforce limits on the resources consumed by their single node applications. In a cluster of collaborating virtual machines, because there is a limited amount of resources to be shared among several instances, some resources must be constrained in favor of an application or group of applications.

Extending a managed language VM to be aware of the existing resources must be done without compromising the usability (mainly portability) of application code. The VM must continue to run existing applications as they are. This component is an extended Java virtual machine with the capacity to extract high and low level VM parameters, e.g., heap memory, network and system threads usage. Along with the capacity to obtain these parameters, they can also be constrained to reflect a cluster policy. The monitoring system is extensible in the number and type of resources to consider.

3.1.3 Checkpointing and migration of the execution state

The counterpart of resource containment is when an application needs more resources but the node where it is running is highly loaded, exhausted. If the application is allowed some elasticity in the resources used, they should be made available in spite of the limitations on the current executing node. In that case, it may be necessary for that application, or others less demanding, to migrate to another node. This migration should be done without the need for an application restart. To this end, the resource aware VM includes a mechanism for checkpointing and migration, which enables the whole application to migrate to another node, where another resource-aware VM is running with the necessary amount of resources. The migration is performed without restarting the application, avoiding losing all the work previously done. This is particularly useful for applications with long execution times, as in various fields related with e-Science (mostly in the context of Grid and Cloud computing) where managed languages are becoming dominant, including chemistry, computational biology and bio-informatics [36, 52], with many available Java-based APIs (e.g., Neobio).

The checkpointing component of the architecture represents the necessary extensions to the HLL-VM so that fine-grained checkpointing, restore and migration of applications is possible. Our checkpoint mechanism can also run concurrently with the main program, preventing full pause of the application during checkpointing, thus further reducing the overhead experienced by applications. It addresses checkpoint consistency and excessive resource consumption (i.e. CPU, memory).

The activation of these mechanisms is regulated either by local rules, activated to provide a failure tolerant environment, or by a *Quality of Execution* Controller (QoE Controller). In the last case, based on execution requirements (e.g. CPU, memory and network usage), the QoE controller can apply two coarse grained measures: *i)* checkpoint and suspension of a VM, *ii)* migration of the application execution state to another node. Next, we describe the main aspects of the architecture of the checkpoint-enabled HLL-VM, consisting of a set of components that focus on transparency and completeness properties. There are three primary components:

- **Application Execution State Extraction**: Applications execute in the context of an extended VM with mechanisms to support checkpoint, restore and migration. State extraction captures the execution state related with all threads within the application. Checkpoint has the obligation to stop all threads (to guarantee consistency), then calls state extraction and finally saves the state persistently into a file system. Migration calls checkpoint and sends that execution state via network.

  State restoration has the responsibility to rebuild the execution state in a newly created application, which corresponds to reconstructing and resuming the execution of all stack frames. for all threads, and when ready, restart execution. Restore guarantees that the newly created application can initiate state restoration, and additionally if requested, obtains the state from a file system.

- **Checkpoint and Migration API**. The execution runtime must provide a local and remote interface so that the checkpoint or migrate operation can be trigger either by rules running on the local process or by an external controller.
- **Migration Service**: To receive an application state a HLL-VM must be running or be started on request on the destinaton node. To that end, a migration service must be present on all nodes to receive migrated applications.

3.1.4 Cluster-wide thread placement

To enable effective distribution of load among different nodes of the cluster, our system relies on a cluster level load balancer capable of spawning new threads (or work tasks) on any cluster node based on a cluster wide policy. When an application asks for a new thread to be created (e.g., by invoking the `start` method on a `Thread` object), the request can be either denied or granted based on the resource allocation decided for the cluster. If it is granted, the load balancer will create the new thread in the most appropriate node to fulfill the cluster policy. For example, if the application has a high priority order compared to other applications of the cluster, then the thread could be created in a lesser loaded node (preferably, one with a VM already assigned to the application's DSO; if needed and allowed, a new VM on any lesser loaded node). The decision on what node new threads are created is left to the policy engine to decide with current information. Nevertheless, the resource-aware VM has an important role in this process, by making it possible to impose a hard limit on resources, e.g., the number of running threads of the application at a specific node, or globally.

3.1.5 Adaptability and the Policy Engine

The policy engine is responsible for loading and enforcing the policies provided by administrators and possibly users regarding resource management. It achieves this by, globally, sending the necessary commands to the resource-aware HLL-VMs, in order for them to modify some runtime parameters, or the type of algorithm used to accomplish a cluster related task, as well as instructing them to spawn threads or activate checkpointing/restore and migration mechanisms. A special focus of this component of *QoE-JVM* is also on the improvement of applications' performance, and what can be adapted in the underlying resource-aware VMs in order to achieve it.

It operates autonomously or in reaction to a given resource outage in the VMs. Autonomous behavior is governed by maintaining knowledge about the applications' previous execution, and adjusting the VMs and cluster parameters to achieve better performance for that specific application. Reactive operation is driven by declarative policies that determine the response

```
<?xml version="1.0" encoding="UTF-8"?>
<RAMConfiguration>
  <ResourceAttributes name="NumberOfThreads" initalLimit="15" />
  <ResourceAttributes name="CpuUsage" initalLimit="75%" />
  ...
  <Rule target="NumberOfThreads">
    <!-- Determines how accumulation is done -->
    <OnConsume> <Counter/> </OnConsume>
    <!-- Determines what happens if limit is reached -->
    <OnLimit> <ResourceException/> </OnLimit>
    <!-- Determines what happens if consumption is successful -->
    <OnAfterComsumption>
      <UseCluster threshold="AllCpus"/>
    </OnAfterComsumption>
  </Rule>
  <Rule target="CpuUsage">
    <OnConsume> <HistoryAverage window="5"/> </OnConsume>
    <OnLimit> <Suspend miliseconds="500"/> </OnLimit>
  </Rule>
  ...
</RAMConfiguration>
```

Fig. 3.3. Declarative policy

to a resource outage. This response may result in a local adaptation (e.g. restrain the resources of another VM in the same node, or change the GC algorithm to consume less memory but eventually taking more time to execute) or have cluster wide impact (e.g. migrate the entire application to a VM in another node).

Figure 3.3 presents a declarative policy to be used by VM instances represented in Figure 3.2 (i.e. $VM_{1..5}$). It defines limits for CPU usage, and the number of threads and sockets the application is allowed to use. CPU usage and threads are monitored and managed by specific rules but using a similar, reusable approach: i) CPU usage is monitored with a sliding window in order to filter irrelevant peaks, while ii) the number of active threads is also monitored with a sliding window in order to trigger rescheduling only when the limit is consistently exceeded.

Summary

Section 3.2 presented the general view of the proposed distributed managed runtime, *QoE-JVM* , and some details regarding the organization of its core components. Regarding its operation, *QoE-JVM* resorts to a policy-driven adaptability engine that drives resource management, global scheduling of threads, and determines the activation of other coarse-grained mechanisms (e.g., checkpointing and migration among VMs). The goal of such an infrastructure is to provide more flexibility, control, scalability and efficiency to applications running in clusters.

3.2 Driving Adaptability with *Quality-of-Execution*

In this section we start by presenting how a simple and generic metric can be used to determine which runtime resource management strategy can be used for each workload in order to maximize it's performance in face of resource degradation. We then describe which kind of performance or progress metrics are relevant to be used. We finish by presenting the kind of resources that are relevant to be controlled, in order to have an elastic behavior, without breaking the application execution.

3.2.1 *QoE-JVM* Economics

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

As said, we are partially inspired by the Cobb-Douglas [25] production function (henceforth referred as equation) from Economics to motivate and to help characterize the QoE. The Cobb-Douglas equation, presented in Equation 3.1, is used in Economics to represent the *production* of a certain good.

$$Y = A \cdot K^{\alpha} \cdot L^{\beta} \tag{3.1}$$

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

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

Alternatively, labour can be regarded, not as a variable representing a measure of human work employed, but as a result, representing the efficiency of the capital invested, given the production output achieved, i.e., labour as a multiplier of resources into production output. This is usually expressed by representing Equation 3.1 in terms of L, as in Equation 3.2. For simplicity, we have assumed the three extra elements to be equal to one. First, the technological and civilization context does not apply, and since the data center economy is simpler, as there is a single kind of activity, computation, and not several, the relative weight of labour and capital is not relevant. Furthermore, we will be more interested in the variations (relative increments) of efficiency than on efficiency values themselves, hence the simplification does not introduce error.

$$L = \frac{Y}{K} \tag{3.2}$$

Now, we need to map these variables to relevant factors in a cloud computing site (a data center). *Production output* (Y) maps easily to application progress (the amount of computation that gets carried out), while *capital* (K), associated with money, maps easily to resources committed to the application (e.g., CPU, memory, or their pricing) that are usually charged to users deploying applications. Therefore, we can regard labour (considered as the *human factor*, the efficiency of the capital invested in a project, given a certain output achieved) as how effectively the resources were employed by an application to attain a certain progress.

While resources can be measured easily by CPU shares and memory allocated, application progress is more difficult to characterize. We are mostly interested in *relative variations* in application progress (regardless of the way it is measured), as shown in Equation 3.3, according to relative variations in resources (to assess *resource efficiency*), and their complementary variations in *production cost per unit*, *PCU*, as an approximation of the marginal cost (capital), in resources, to achieve the obtained progress (output). The term *unit* is a generic one because we want to apply this rationale to different kinds of resources, as described next.

$$\varDelta\ L \approx \frac{\varDelta\ Y}{\varDelta\ K}, \quad and \quad thus \quad \varDelta\ PCU \approx \frac{\varDelta\ K}{\varDelta\ Y} \tag{3.3}$$

We assume a scenario where, when applications are executed in a constrained (overcommitted) environment, the infrastructure may remove m units of a given resource from a set of resources R (e.g. memory size, CPU cores, bandwidth) and give them to another application that can benefit from this transfer. Examples of transferable units are $50MiBytes$ of heap size, 1 core and $2MiBytes$ of bandwidth. This transfer may have a negative impact in the application that offers resources and it is expected to have a positive impact in the receiving application. To assess the effectiveness of the transfer, the infrastructure must be able to measure the impact on the giver and receiver applications, namely somehow to measure the approximate savings in PCU, that is, the relation between employed resources and effective progress, as described next.

Variations in the PCU can be regarded as an opportunity for *yield* regarding a given resource r, and a *management strategy*. The term *strategy* generically identifies the currently in use and the available configuration options. Naturally, comparing strategy s_a and s_b only makes sense if they are of the same nature. For example, s_a and s_b can represent different kinds of garbage collection algorithms or different ratios to grow/shrink the heap size. So, the *yield* is a return or reward from applying a given strategy to some managed resource, during the time span ts, as presented in Equation 3.4.

$$Yield_r(ts, s_a, s_b) = \frac{Savings_r(s_a, s_b)}{Degradation(s_a, s_b)} \tag{3.4}$$

Because *QoE-JVM* is continuously monitoring the application progress, it is possible to incrementally measure the yield. Each partial $Yield_r$, obtained in a given time span ts, contributes to the total one obtained. This can be evaluated either over each time slice or globally when applications, batches or workloads complete. For a given execution or evaluation period, the total yield is the result of summing all significant partial yields, as presented in Equation 3.5.

$$TotalYield_r(s_a, s_b) = \sum_{ts=0}^{n} Yield_r(ts, s_a, s_b) \tag{3.5}$$

The definition of $Savings_r$ represents the savings of a given resource r when two allocation or management strategies are compared, s_a and s_b, as presented in Equation 3.6. The functions $U_r(s_a)$ and $U_r(s_b)$ relates the *usage* of resource r, given two allocation configurations, s_a and s_b. We allow only those reconfigurations which offer savings in resource usage to be considered in order to calculate yields.

$$Savings_r(s_a, s_b) = \frac{U_r(s_a) - U_r(s_b)}{U_r(s_a)} \tag{3.6}$$

Regarding *performance degradation*, it represents the impact of the savings, given a specific performance metric, as presented in Equation 3.7. Considering the time taken to execute an application (or part of it), the performance degradation relates the execution time of the original configuration, $P(s_a)$, and the execution time after the resource allocation strategy has been modified, $P(s_b)$.

$$Degradation(s_a, s_b) = \frac{P(s_b) - P(s_a)}{P(s_a)} \tag{3.7}$$

Each instance of the *QoE-JVM* continuously monitors the application progress, measuring the *yield* of the applied strategies. As a consequence of this process, QoE, for a given set of resources, can be enforced observing the *yield* of the applied strategy, and then keeping or changing it as a result of having a good or a bad impact. To accomplish the desired reconfiguration, the underlying resource-aware VM must be able to change strategies during execution, guided by the global QoE manager. Section 3.3 shows how existing high-level language virtual machines can be extended to accommodate the desired adaptability.

To effectively apply the economic model presented in this section it is necessary to quantify the application progress metric, what resources are relevant and which extensions points exist or need to be created inside the HLL-VM. The following section discuss these topics in further detail.

3.2.2 Progress monitoring

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

To compare different metrics to measure progress, we classify applications as request driven (or interactive) and continuous process (or batch). Request driven applications process work in response to an outside event (e.g. HTTP request, new work item in the processing queue). Continuous processing applications have a target goal that drives their calculations (e.g. align DNA sequences). For most non-interactive applications, measuring progress is directly related to the work done and the work that is still pending. For example, some algorithms to analyze graphs of objects have a visited/processed objects set, which typically will encompass all objects when the algorithm terminates (or at least, a significant part of it). If the rate of objects processed can be determined it will indicate how the application is making progress. Other examples would be applications to perform video encoding, where the number of frames processed is a measure of progress [45].

There is a balance and trade-off in measuring progress, using a metric that is close to the application semantics, and the transparency of progress measuring. The number of requests processed, for example, is a metric closely related to the application semantics, which gives an almost direct notion of progress. Nevertheless, it will not always be possible to acquire such information. On the other hand, low level activity, such as I/O or memory pages access, is always possible to acquire inside the VM or the OS. But relating this type of metrics to the application effective progress is a challenging task. The following are relevant examples of metrics that can be used to monitor the progress of an application, presented in a decreasing order of closeness to application semantics, but with an increasing order regarding the level of transparency.

- **Number of requests processed**: This metric is typically associated with interactive applications, such as web applications or with Bag-of-Tasks jobs;
- **Completion time**: For short and medium time living applications, where it is not possible to change the source code or no information is available to lead an instrumentation process, this metric will be the more effective one. This metric only requires the *QoE-JVM* to measure *wall clock time* when starting and ending the application (or alternatively measuring CPU time used);
- **Code: instrumented or annotated**: If information is available about the application high level structure, instrumentation can be used to dynamically insert probes at load time, so that the *QoE-JVM* can measure progress using a metric that is semantically more relevant to the application;
- **Mutator execution time**. When mutators (i.e. execution flows of applications) have high execution percentages, in proportion to the time spent in garbage collector, this indicates that the application is making more progress than others where garbage collection is using a higher percentage of total execution.
- **I/O: storage and network**: For applications dependent on I/O operations, changes in the quantity of data saved or read from files, or in the information sent and received from the network, can contribute to determine whether the application reached a bottleneck or is making progress;
- **Memory page activity**: Allocation of new memory pages is a low level indicator (collected from the OS or the VMM) that the application is making effective progress. A similar indication will be given when the application is writing in new or previously unused (or unmodified) memory pages.

Although *QoE-JVM* could read low level indicators as I/O storage and network activity or memory page activity, we currently use the metric **completion time** to measure performance degradation, as defined in Section 3.2.1. This is so because the applications used to demonstrate the benefits of our system are benchmarks that are representative of different types of workloads but tend to have a short execution time (more details in Chapter 4).

3.2.3 Resource types and usage

In the model presented at Section 3.2.1, $Savings_r$ refers to any computational resource (r) which applications consume to make progress. Resources can be classified as either *explicit* or *implicit*, regarding the way they are consumed. *Explicit* resources are the ones that applications request during execution, such as, number of allocated objects, number of network connections, number of opened files. *Implicit* resources are consumed as the result of executing the application, but are not explicitly requested through a given library interface. Examples include, the heap size, the number of cores or the network transfer rate.

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

Table 3.1. Implicit resources and their throttling properties

Both types of resource are relevant to be monitored and regulated. *Explicit* and *implicit* resources might be constrained as a protection mechanism against ill behaved or misusing applications [32]. For well behaved applications, restraining these resources further below the application contractual levels will lead to an execution failure. On the other hand, the regulation of *implicit* resources determines how the application will progress. For example, allocating more memory will potentially have a positive impact, while restraining memory will have a negative effect. Nevertheless, giving too much of memory space is not a guarantee that the application will benefit from that allocation, while restraining memory space will still allow the application to make some progress.

In this work, we focus on controlling some types of *implicit* resources because of their potential to provide elasticity to resource management. *QoE-JVM* can control the admission of these resources, that is, it can throttle resource usage. It gives more to the applications that will progress faster if more resources are allocated. Because resources are finite, they will be taken from (or not given to) other applications. Even so, the *QoE-JVM* will strive to choose the applications where progress degradation is comparatively smaller.

Table 3.1 presents *implicit* resources and the throttling properties associated to each one. These properties can be either counted values (e.g. x number of cores) or rates (e.g. y KiBytes/seconds). To regulate CPU and memory both types of properties are applicable. For example, CPU can be throttled either by controlling the number of cores or the *cap* (i.e. the maximum percentage of CPU a VM is able to use, even if there is available CPU time). Memory usage can be regulated either through a fixed limit or by using a factor to shrink or grow this limit. Although the heap size cannot be smaller than the working set of the application, the size of the extra allocated memory influences the application progress. A similar rationale can be made about resource pools, which are a common strategy to manage resources in applications handling multiple requests, such as web and database servers (e.g. thread pools, connection pools).

Summary

Section 3.2 presented a general resource allocation and adaptation schema that obeys to a VM economics model, based on aiming overall quality-of-execution (QoE) through resource efficiency. Essentially, *QoE-JVM* puts resources where they can do the most good to applications and the cloud infrastructure provider, while taking them from where they can do the least harm to applications.

3.3 Resource Management Mechanisms

In this chapter we present some implementation details of the three main resource management mechanisms that are supported by our execution platform. We focus on the necessary extensions to a high level virtual machine, regarding resource accounting, internal mechanisms adaptability and checkpointing. We also present the transparent integration with external middleware through byte code instrumentation for the mechanism that spawns threads across the cluster.

3.3.1 Resource accounting and adaptability

To implement our architecture we need to develop a managed language virtual machine with the capacity to monitor and restraint the use of resources based on a dynamic policy, defined declaratively outside the VM. Some work has been done in the past aiming to introduce resource-awareness in such high level virtual machines (which details were presented in Section 2.5.2). Nevertheless, to the best of our knowledge, none of them is publicly available or currently usable with popular software, operating systems and hardware architectures. Based on this observation, we have chosen to extend the Jikes RVM [3] to be resource-aware. Thus, in the next subsection we will describe different aspects of our current work on Jikes RVM. Later on, we describe the main implementation aspects of our system regarding the spawning and scheduling of threads in other nodes.

Figure 3.4 depicts further details on the architecture of the resource-aware VM we developed for *QoE-JVM* . The resource-aware HLL-VM has a specific module for each type of manageable resource (e.g., files, threads, CPU usage, connections, bandwidth, and memory). Each of the module exports to the Resource Awareness and Management Module (RAMM) an *attribute* that abstracts the specifics of the resource. This way, when the RAMM decides to limit, reduce or block the usage of a resource by the application, it can instruct the respective attribute without worrying about the details of applying limitation to that specific resource (e.g., disallowing file open, or take a thread out of scheduling). The RAMM consumes profile information from the main VM and *QoE-JVM* mechanisms (GC and JIT level, and

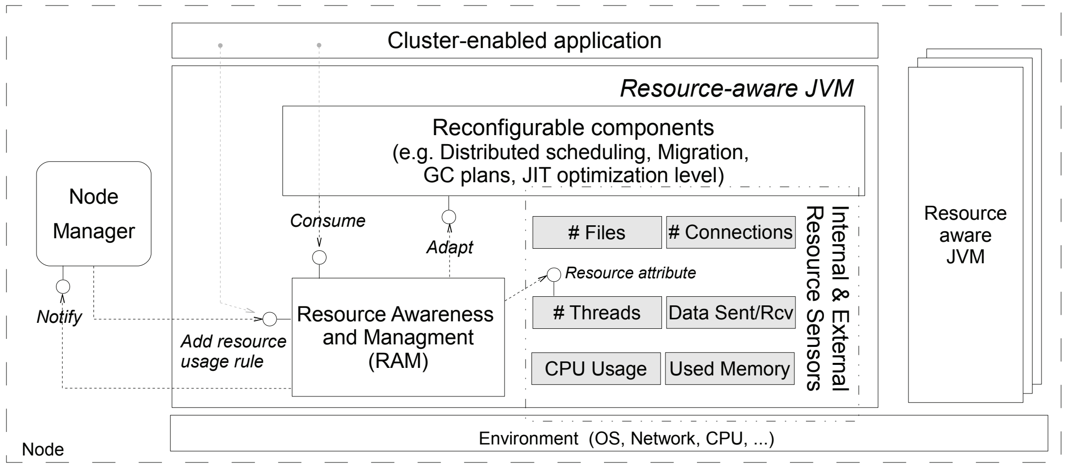

Cluster-enabled application
Resource-aware JVM
Reconfigurable components
(e.g. Distributed scheduling, Migration, GC plans, JIT optimization level)
Node Manager
Consume
Adapt
Notify
Add resource usage rule
Resource Awareness and Managment (RAM)
Resource attribute
# Files
# Connections
# Threads
Data Sent/Rcv
CPU Usage
Used Memory
Internal & External Resource Sensors
Resource aware JVM
Environment (OS, Network, CPU, ...)
Node

Fig. 3.4. Interactions with the Resource Awareness and Management Module

distributed scheduling and migration, respectively). These mechanisms can be adapted and reconfigured by command of the RAMM.

Resource management policies

Being RAMM the *engine* that enables awareness and adaptation, all its decisions are carried out according to the evaluation of rules in the policies loaded by the node manager. The node manager is also notified by the RAMM, in each VM, about the application's performance and outcome of RAMM's decisions.

The management of a given resource implies the capacity to monitor its current state and to be able to directly or indirectly control its use and usage. The resources that can be monitored in a virtual machine can be either specific of the runtime (e.g. number of threads, number of objects, amount of memory) or be strongly dependent of the underlying architecture and operating system (e.g. CPU usage). To unify the management of such disparate types of resources, we carried out the implementation of JSR 284 - The Resource Management API [31] in the context of Jikes RVM, previously not implemented in the context of any widely usable virtual machine.

The relevant elements to resource management as prescribed by JSR 284 are: *resources*, *consumers* and *resource management policies*. Resources are represented by their attributes. For example resources can be classified as *Bounded* or *Unbounded*. *Unbounded* resources are those that have no intrinsic limit (or if it exists, it is large enough to be essentially ignored) on the consumption of the resource (e.g. number of threads). The limits on the consumption of unbounded resources are only those imposed by application-level resource usage policies. Resources can also be *Bounded* if it is possible to reserve a priori a given number of units of a resource to an application.

```
public class ThreadsCreationNode implements Notification {
  long _threshold;
  public ThreadsCreationNode(long threshold) {
     _threshold = threshold;
  }
  public void postConsume(
    ResourceDomain domain,
    long previousUsage, long currentUsage) {
    if (currentUsage >= _threshold)
      Scheduler.getInstance().changeAllocationToCluster();
  }
}
```

Fig. 3.5. A sample notification handling to change thread allocation to the cluster

A *Consumer* represents an executing entity which can be a thread or the whole VM. Each consumer is bound to a resource through a *Resource Domain*. *Resource domains* impose a common resource management policy to all *consumers* registered. This policy is programmable through callback functions to the executing application. Although *consumers* can be bound to different *Resource Domains*, they cannot be associated to the same *resource* through different *Domains*.

When a resource is about to be consumed, the resource-aware VM, implementing JSR 284, delegates this decision, via a callback, that can be handled by RAMM, and either allowed, delayed or denied (with an exception thrown).

Figure 3.5 shows a notification, `ThreadsCreationNode`, which can be used to configure an *QoE-JVM* instance. This callback would be called on each local thread allocation (in Jikes RVM, Java threads are backed by a native class, `RVMThread` that is shown, and that interacts with the host OS threads). It determines that if the number of threads created in the local node reaches a certain threshold new threads will be created elsewhere in the cluster. The exact node where they will be placed is left to be determined by the distributed scheduler own policy.

Figure 3.6 shows a *constraint*, `HistoryAverage`, which can be used to regulate a CPU usage policy. Consider a scenario where the running application cannot use the CPU above a threshold for a given time window, because the remaining CPU available is reserved for another application (e.g., as part of the quality-of-execution awarded to it). In this case, when the CPU usage monitor evaluates this rule, it would suspend all threads (i.e. return 0 for the allowed usage) if the intended usage is above the average of the last `wndSize` observations. A practical case would be to suspend the application if the CPU usage is above 75% for more than 5 observations.

```
public class HistoryAverage implements Constraint {
  ...
  long[] _samplesHistory;
  public HistoryAverage(int wndSize, long maxConsumption)
  { ... }
  public long preConsume(ResourceDomain domain,
         long currentUsage, long proposedUsage) {
    long average = 0;
    if (_nSamples == _samplesHistory.length) {
      average = _currentSum / _nSamples;
      _currentSum -= _samplesHistory[_idx];
    }
    else { _nSamples  += 1; }
    _currentSum += proposedUsage;
    _samplesHistory[_idx] = proposedUsage;
    _idx = (_idx + 1) % _samplesHistory.length;
    return average > _maxConsumption ? 0 : proposedUsage;
  }
}
```

Fig. 3.6. Regulate consumption based on past *wndSize* observations

Changes to the VM and Classpath

Our first experiences were done in order to have control on the spawning of new threads, a common source of CPU contention and performance degradation when multiple applications are running. We made modifications to the Jikes runtime classes and extended the GNU classpath. The Jikes boot sequence was augmented with the setup of a *resource domain* to manage the creation of application level threads. VM threads (e.g. GC, finalizer) are not accounted. The Jikes component responsible for the creation and representation of system level threads was extended to use the callbacks of the previous mentioned *resource domain*, such that the number of new threads is determined by a policy defined declaratively outside the runtime.

All native system information, including CPU usage, is currently obtained using the kernel `/proc` filesystem. In the Jikes RVM, the interaction with OS system calls, are efficiently supported by the available JIT compilers. When a properly annotated method is called, the JIT compiler will generate a call to a “C” language *stub*, using the platform’s underlying calling convention. Our stub then reads from `/proc`, and returns the results.

Finally, a new package of classes was integrated in the GNU classpath in order for applications to be able to specify their policies. These classes interact with the resource-aware underlying VM so that the application can add their own resource consumption policies, if needed. Nevertheless, policies can be installed with total transparency to the application.

With this infrastructure, all consumable resources monitored, or directly controlled by the VM and class library, can be constrained by high-level policies defined externally to the VM runtime.

3.3.2 Concurrent checkpoint

Our checkpoint mechanism can also run concurrently with the main program, preventing full pause of the application during checkpointing, thus further reducing the overhead experienced by applications. There are two main implementation issues regarding concurrent (or incremental) checkpointing: i) ensuring checkpoint consistency, since the application continues executing while the checkpoint is created, and ii) avoiding excessive resource consumption (CPU, memory), due to the extra load of executing the application and the checkpointing mechanism simultaneously, that could lead to thrashing and preclude the very performance gains sought by executing the checkpointing concurrently.

The first issue is related with isolation and atomicity. The checkpoint, while being carried out concurrently, must still be atomic regarding the running application. This means it must reflect a snapshot of the execution state that would also be obtained with the application paused or suspended (while the application is not modifying its state). Otherwise, there could co-exist in the snapshot objects checkpointed at different times, making the whole object graph inconsistent and violating application invariants. In essence, the challenge in this operation is that the application's working set (and VM's internal structures) will change, while the checkpointing is being carried out. If the changes were to be reflected into the data being saved, the checkpoint would be useless for virtue of being inconsistent.

The second issue stems from the fact that if we want to simultaneously freeze a *clone* of the application state in time (to be able to save it in the checkpoint concurrently), while the application keeps executing and accessing the *original* object graph, it would potentially almost double the memory occupied by the virtual machine. Furthermore, performing the serialization of the *clone* object graph, will cause contention for the CPU, with the application code that is simultaneously being executed (although the OS is able to interleave their execution with some degree of efficiency).

Fortunately, two aspects of current architectures help when dealing with these issues: i) lazy memory duplication, as embodied in *copy-on-write* mechanisms provided by the memory management modules in modern operating systems, and ii) the increasing prevalence of multicore hardware, available in most computers today. These two aspects are leveraged to ensure concurrent checkpointing offers smaller overhead to applications running.

In fact, the *original* and *clone* version of the object graph need not exist physically in their entirety. To efficiently support this, we use the *copy-on-write* mechanism that allows two processes to share the whole of the address space, with pages modified by one of them

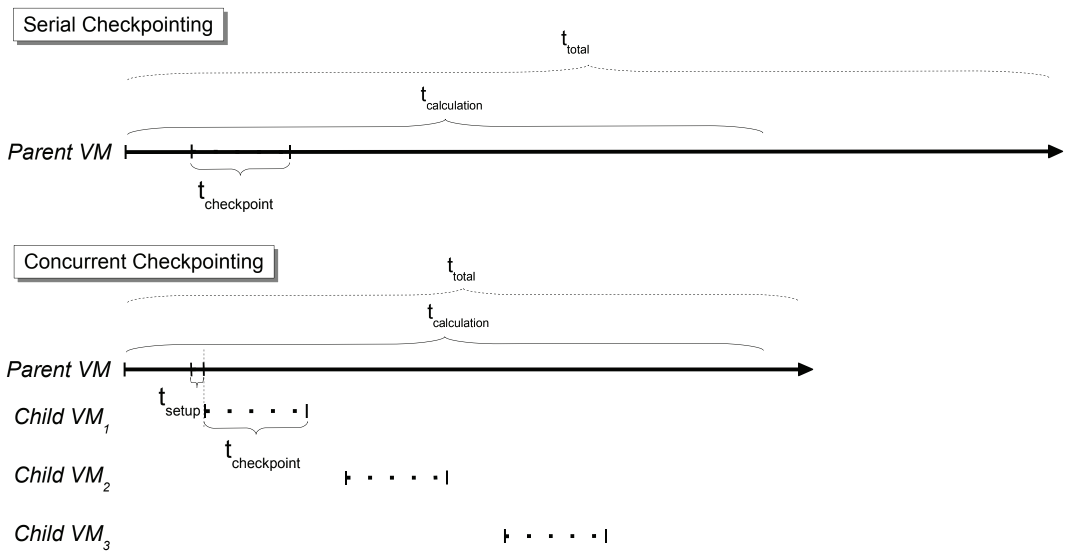


Fig. 3.7. Timelines of serial and concurrent checkpoint

copied on demand. Currently, our implementation in Linux relies on Linux's system call, `fork()`, which has the desired semantics [74]. In Windows, the same primitive and semantics is available through the POSIX subsystem, thus ensuring portability across the two operating systems. Therefore, the memory overhead will be bounded to the memory pages containing objects that are actually modified during the checkpointing. Due to the locality in memory accesses during application execution (locality-of-reference and working set principles), this amount is limited.

Figure 3.7 illustrates how the concurrent checkpoint progresses, along with the application, in comparison with the serial (non-concurrent) approach. $t_{calculation}$ is the free run time, without any checkpoint. t_{total} is the total execution time, considering either serial or concurrent checkpoint.

With serial checkpointing, the total execution time of an application is, expectably, the sum of the time performing its calculations or processing (hereafter calculation time), with the time to perform a checkpoint (once in the figure), assuming approximate times, multiplied by the number of checkpoints taken. Therefore, checkpointing is always in the critical path regarding the total execution time, precluding so frequent checkpointing (for instance, very large working sets, and not very long executions, probably only once at mid execution time).

With concurrent checkpointing, most of the checkpointing time is removed from the critical path regarding total execution time (only the time to setup the child-VM remains). This makes it feasible to perform checkpoints more frequently, without significantly penalizing application execution times, thus reducing even more the amount of lost computation (lost work) whenever a failure takes place.

3.3.3 Cluster-wide thread placement

Our mechanism to distribute threads among the cluster is built by leveraging and extending the Terracotta [17] Distributed Shared Objects. This middleware uses the client/server terminology and calls the application JVMs that are clustered together Terracotta *clients* or *Terracotta cluster nodes*. These clients run the same application code in each JVM and are clustered together by injecting cluster-aware bytecode into the application Java code at runtime, as the classes are loaded by each JVM. This bytecode injection mechanism is what makes Terracotta transparent to the application. Part of the cluster-aware bytecode injected causes each JVM to connect to the Terracotta server instances. In a cluster, a Terracotta server instance handles the storage and retrieval of object data in the shared clustered virtual heap. The server instance can also store this heap data on disk, making it persistent just as if it were part of a database. Multiple terracotta server instances can exist as a cohesive array.

In a single JVM, objects in the heap are addressed through references. In the Terracotta clustered virtual heap objects are addressed in a similar way, through references to clustered objects which we refer to as distributed shared objects or managed objects in the Terracotta cluster. To the application, these objects are just like regular objects on the heap of the local JVMs, the Terracotta clients. However Terracotta knows that clustered objects need to be handled differently than regular objects. When changes are made to a clustered object, Terracotta keeps track of those changes and sends them to all Terracotta server instances. Server instances, in turn, make sure those changes are visible to all the other JVMs in the cluster as necessary. This way, clustered objects are always up-to-date whenever they are accessed, just as they are in a single JVM. Consistency is assured by enforcing the same synchronization semantics already present in Java language (with monitors), which turns into Terracotta transaction boundaries. Piggybacked on these operations, Terracotta injects code to update and fetch data from remote nodes at the beginning and end of these transactions.

Therefore we need to perform additional byte-code enhancement on application classes as a previous step to the byte-code enhancing performed by the Terracotta cluster middleware before applications are run. The class, method and field visitors of the ASM framework [19] are used to perform this task. Creation of threads in remote nodes is a result of invoking the Resource Awareness Management Module, described in Section 3.3.1, in order to attempt to consume a thread resource at that node. The most intricate aspects deal with the issue of enforcing thread transparency (regarding its actual running node) and identity across the cluster (regarding monitor ownership and thread synchronization operations such as *join*, etc.), as we explain next.

The instrumentation replaces Java type opcodes that have the Java Thread type as argument with equal opcodes with our custom type ClusterThread. It also replaces the `getfield` and `getstatic` opcodes type with ClusterThread instead of Thread. As the ClusterThread class extends the original Java Thread class, type compatibility is guaranteed. For the method calls, some of the methods belonging to the Thread class are final, and therefore cannot be overridden. To circumvent this, we aliased the final methods and replaced Thread method calls with the aliased method. For example, if we have an `invokevirtual` opcode that invokes the final "join" method of the Thread class, we invoke the "clusterJoin" method instead.

Summary

Section 3.3 presented relevant implementation details of three mechanisms for resource management. The incorporation of these mechanisms in the execution environment was made by changing the code base of a Java HLL-VM (Jikes RVM) and taking advantage of a shared objects middleware (Terracotta DSO). The mechanisms are transparent to the application developer, allowing the owner of the execution infrastructure to activate them autonomously.

4

Evaluation

In this chapter we show the current evaluation of the adaptability metric and adaptability mechanisms described Chapter 3. The evaluations were made in a cluster where nodes had a Intel(R) Core(TM)2 Quad processors (with four cores) and 8GB of RAM, each running Linux Ubuntu 9.04. Jikes RVM [3] code base is version 3.1.1 and the *production* configuration was used to build the source code (both base version and modifications).

4.1 QoE applied to memory and CPU management

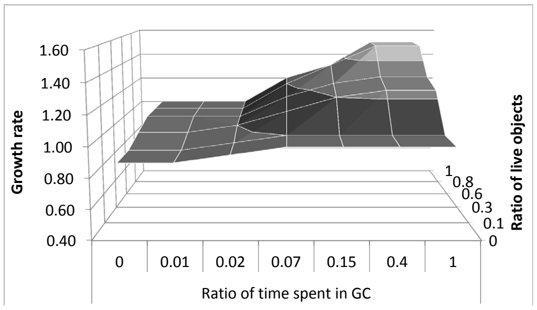
The resource management economics, presented in Chapter 3.2.1, were applied to manage the heap size and CPU usage regarding different types of workloads.

4.1.1 Heap Size

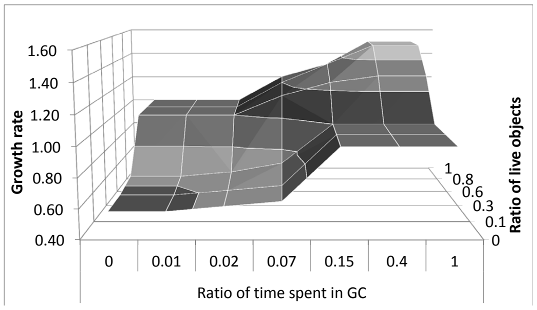
The default heap growing matrix (hereafter known as M_0) is presented in Figure 4.1.a. In this, and in the remaining matrices, 1.0 is the neutral value, representing a situation where the heap will neither grow nor shrink. Other values represent a factor of growth or shrink, depending if the value is greater or smaller than 1, respectively. To assess the benefits of our resource management economics, we have setup three new heap size changing matrices. The distinctive *factors* are the growth and decrease rates determined by each matrix.

Matrices M_1 and M_2, presented in Figure 4.1.b and 4.1.c, impose a strong reduction on the heap size when memory usage and management activity is low (i.e. few live objects and short time spent on GC). Nevertheless they provide very different growth rates, with M_1 having a faster rate when heap space is scarce. Finally, matrix M_3 makes the heap grow and shrink very slowly, enforcing a more rigid and conservative heap size until program dynamics reach a high activity point (i.e. high rate of live objects and longer time spent on GC) or decrease activity sharply.

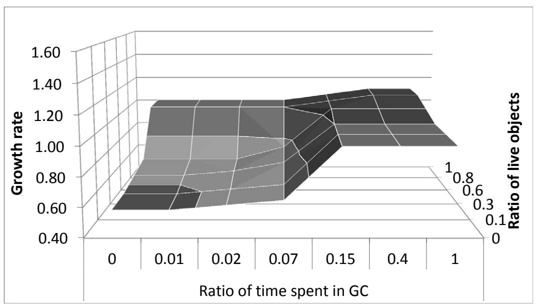
Each tenant using the Cloud provider infrastructure can potentially be running different programs. Each of these programs will have a different *production*, i.e. execution time, based



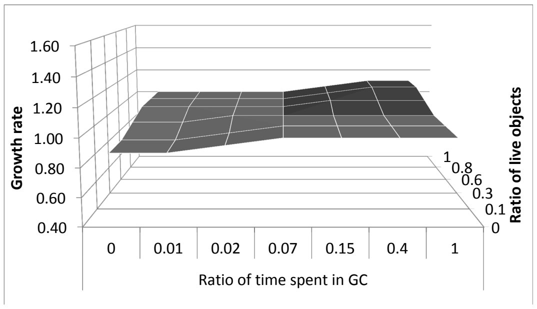
(a) M_0



(b) M_1



(c) M_2



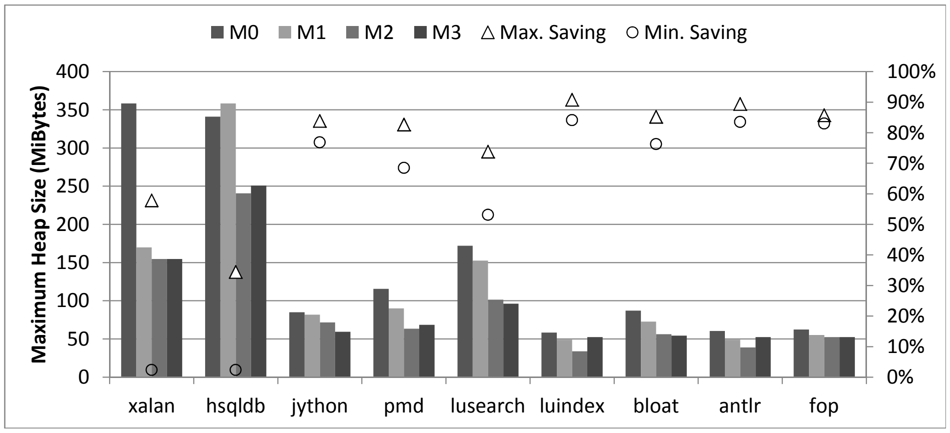
(d) M_3

Fig. 4.1. Default (M_0) and alternative matrices to control the heap growth.

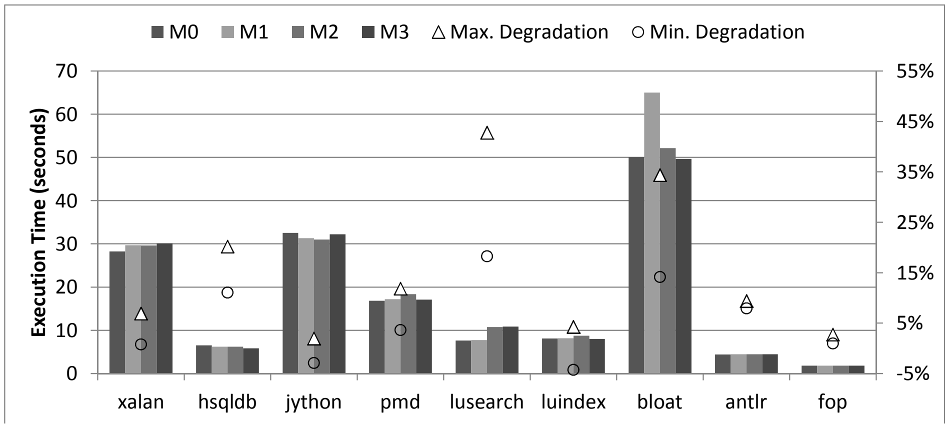
on the *capital* applied, i.e. growth rate of the heap. To represent this diversity, we used the well known DaCapo benchmarks [14], a set of programs which explore different ways of organizing programs in the Java language.

To measure the *yield* of each matrix we have setup an *identity matrix* (all 1's), that is, a matrix that never changes the heap size. Figures 4.2.a shows the maximum heap size (left axis) after running the DaCapo benchmarks with configuration `large` and a maximum heap size of 350 MiBytes, using all the matrices presented in Figure 4.1. In the right axis we present the maximum and minimum of *resource savings*, as defined in Equation 3.6. These values were obtained for each of the matrices when compared to the *identity matrix* with heap size fixed at 350 MiBytes. The *resource savings* are above 40% for the majority of the workloads, as can be seen in more detail in Table 4.1.

In Figure 4.2.b we present the evaluation time of the benchmarks (left axis) and the average *performance degradation* (right axis), as defined in Equation 3.7, regarding the use of each of the ratio matrices. Degradation of execution time reaches a maximum of 35% for lusearch, Apache's fast text search engine library, but stays below 25% for the rest of the benchmarks. Table 4.1, summarizes the *yield*, as defined in Equation 3.4, when using different matrices to manage the heap size.



(a) Maximum heap size and average savings percentage

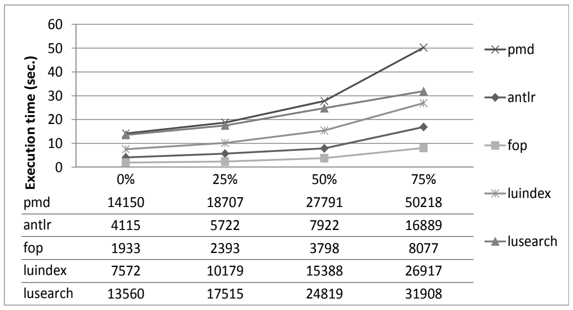


(b) Execution time and average performance degradation percentage

Fig. 4.2. Results of using each of the matrices ($M_{0..3}$), including savings and degradation when compared to a fixed heap size.

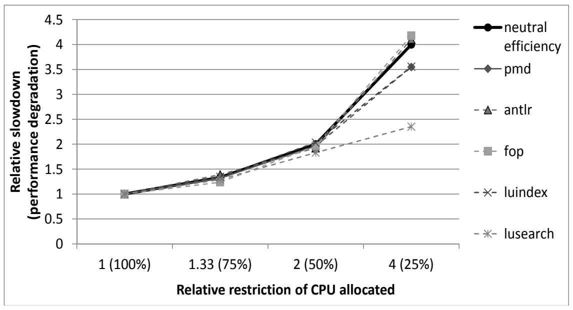
Matrix	M0			M1			M2			M3		
	Sav	Deg	**Yield**	Sav	Deg	**Yield**	Sav	Deg	**Yield**	Sav	Deg	**Yield**
xalan	2.3%	0.8%	*3.1*	53.7%	5.5%	*9.7*	57.9%	5.4%	***10.7***	57.9%	6.9%	*8.4*
hsqldb	7.1%	20.2%	*0.4*	2.3%	16.4%	*0.1*	34.4%	16.2%	*2.1*	31.6%	11.1%	***2.9***
jython	76.8%	1.9%	*39.9*	77.7%	-1.9%	*-40.4*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

Table 4.1. The *yield* of the matrices presented in Figure 4.1



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

(a) Effects of restraining CPU by 25%, 50% and 75%



(b) Relative slowdown

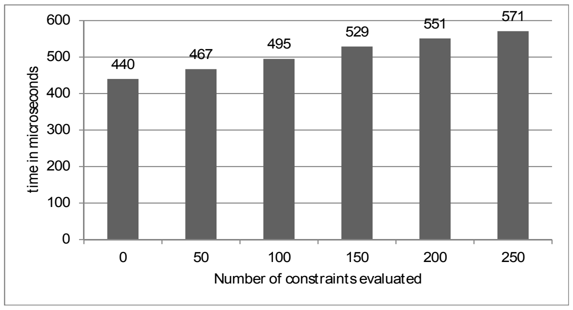
Fig. 4.3. Effects of restraining CPU and corresponding relative slowdown

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

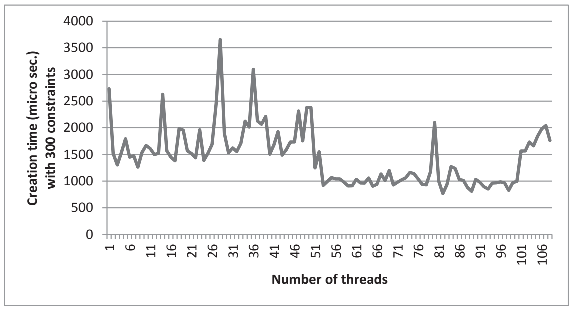
4.1.2 CPU

Our system also takes advantage of CPU restriction in a coarse-grained approach. Figure 4.3.a shows how five different Java workloads (taken from the DaCapo benchmarks) react to the deprivation of CPU (in steps of 25%), regarding their total execution time. Figure 4.3.b shows the relative performance slowdown, which represents the *yield* of allocating 75%, 50% and 25%, comparing with 100% of CPU allocation. Note that, comparing with previous graphics, some applications have longer execution times with 0% CPU taken because they are multithreaded and we used only 1 core for this test.

As expected, the execution time grows when more CPU is taken. This enables priority applications (e.g. paying users, priority calculus applications) to run efficiently over our runtime, having the CPU usage transparently restricted and given to others (a capability in itself currently unavailable in HLL-VMs). Finally, we note that 3 applications (`pmd`, `luindex` and `lusearch`) have yields greater than 1 when CPU restriction is equal or above 50%, as they stay below the neutral efficiency line in Figure 4.3.b, due to memory or I/O contention.



(a) Thread creation time with increasing number of constraints to evaluate



(b) Thread creation time during execution with 200 constraints (GC spikes omitted)

Fig. 4.4. Policy evaluation cost

4.2 Resource consumption constraints

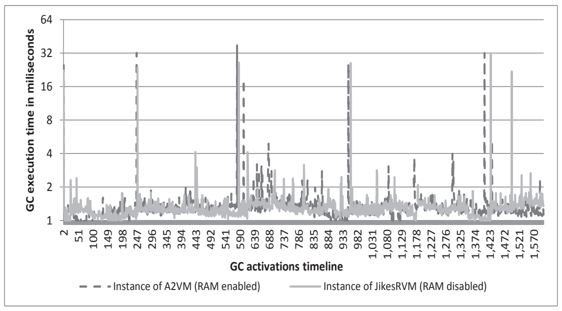
The first part of our performance evaluation regards the resource-aware VM and its impact on rules' evaluation during regular VM operations.Therefore we conducted a series of tests, measuring different aspects of a running application: i) the overhead introduced in the consumption of a specific resource and ii) policy evaluation in a complete benchmark scenario. All these evaluations are made locally in a single modified Jikes RVM (version 3.1.1), compiled with the `production` profile[1].

In Figure 4.4.a we can observe the evolution of the overhead introduced to thread creation, by measuring average thread creation and start time, as the policy engine has increasingly larger numbers of rules to evaluate, up to 250 (simulating a highly complex policy). The graph shows that this overhead, while increasing, does not hinder scalability as it is very small. When evaluating 50 constrains (which would correspond to a more reasonable but still complex policy), each checking if a certain limit has been reached, the increment is 27 microseconds in each thread creation, which represents an increase of $\approx 6\%$ to the baseline thread creation time (i.e. no constraints evaluated).

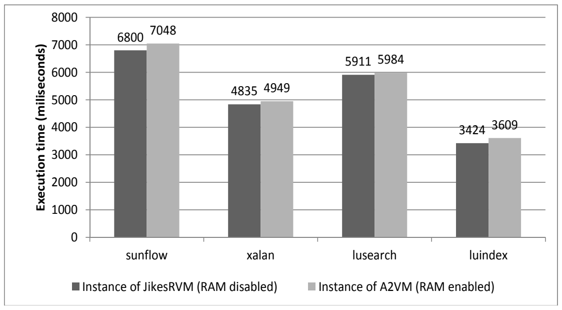
In Figure 4.4.b we evaluate whether resource monitoring and policy evaluation (with 300 constraints) introduce any kind of performance degradation as more and more threads are created, resources consumed. Figure 4.4.b clearly shows (omitting Garbage Collection spikes) that thread creation time does not degrade during application execution, being below 1 millisecond; although subject to some variation, it presents no lasting degradation.

The previous results were obtained monitoring only a single resource, i.e. number of application threads. For other counted resources, e.g. number of bytes sent and received, similar results are expected. Although the allocation of new objects can also be seen as a counted resource, e.g. number of bytes allocated in heap, it is more efficient to evaluate it differently. The cost of checking for constraints regarding object allocation was thus transferred to the

[1] it includes a two-generation garbage collector [15] and the optimized and adaptive compilation system.



(a) GC execution time during Dacapo's LuSearch benchmarck



(b) Four Dacapo's multi threaded benchmarks with RAM enabled and disabled

Fig. 4.5. Macro evaluation of an instance of *QoE-JVM*

garbage collection process, leaving the very frequent allocation operation new of additional verifications.

Figure 4.5.a presents the duration of each GC cycle during the execution of DaCapo's benchmark [14] [2]. `lusearch`, with and without evaluating constraints on heap consumption (i.e. RAM enabled and disabled). The `lusearch` benchmark was configured with a `small` data set, one thread for each available processor (i.e. four threads) and the convergence option active, resulting in some extra warm up runs before the final evaluation.

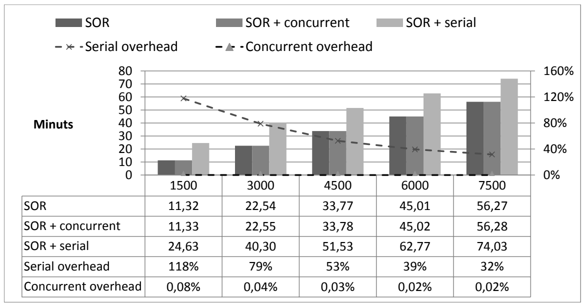
Because of the generational garbage collection algorithm used in our modified Jikes RVM, we can observe many small collection cycles, interleaved with some full heap transversal and defragmentation operations. The two runs share approximately the same average execution time and a similar average deviation: $1.38 \pm 0.31ms$ and $1.38 \pm 0.27ms$, where the former value is when the RAM module is enabled and the last when RAMM is disabled. With these results we conclude that performance of object allocation and garbage collection is not diminished with the extra work introduced.

To conclude the evaluation of the RAM module, we stressed an instance of our resource-aware VM with four macro benchmarks, as presented in Figure 4.5.b. These four benchmarks are multi-threaded applications, which allows us to do a macro evaluation of the proposed modifications. During the execution of these benchmarks there were three resources being monitored (and eventually constrained): the number of threads, the total allocated memory and the CPU usage. The constraints used in evaluation did not restrain the usage of resources so that the benchmarks could properly assess the impact of monitoring different resources simultaneously in real applications (as opposed to the specific benchmarks presented previously in Figures 4.4 and 4.4). The results show only a negligible overhead: 3% in average.

[2] The version 9.12 used in the evaluation of *QoE-JVM* 's RAMM is available at http://www.dacapobench.org/

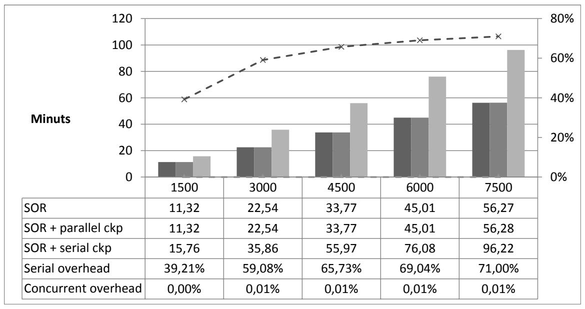
4.3 Concurrent checkpoint

To evaluate the concurrent checkpoint specifically, we set up two different checkpoint scenarios using SOR, which we identify as *Test 1* and *Test 2* checkpoints. These tests use a large array of equations so that larger amounts of data need to be saved, while keeping the number of iterations to 7500, for running times close to 2 hours. We intend to show that concurrent checkpointing makes it feasible to checkpoint larger applications and more frequently. Thus, for each of these classes of tests, SOR was run with a matrix of 3000, 3600 and 4200 equations. The two available cores were used to fully exploit the concurrent checkpointing. We averaged 5 executions of each test.



	1500	3000	4500	6000	7500
SOR	11,32	22,54	33,77	45,01	56,27
SOR + concurrent	11,33	22,55	33,78	45,02	56,28
SOR + serial	24,63	40,30	51,53	62,77	74,03
Serial overhead	118%	79%	53%	39%	32%
Concurrent overhead	0,08%	0,04%	0,03%	0,02%	0,02%

(a) Test 1 - 3000 equations



	1500	3000	4500	6000	7500
SOR	11,32	22,54	33,77	45,01	56,27
SOR + parallel ckp	11,32	22,54	33,77	45,01	56,28
SOR + serial ckp	15,76	35,86	55,97	76,08	96,22
Serial overhead	39,21%	59,08%	65,73%	69,04%	71,00%
Concurrent overhead	0,00%	0,01%	0,01%	0,01%	0,01%

(b) Test 2 - 3000 equations

Fig. 4.6. Checkpointing experiences

The distinguishing factor between these two types of tests is the event or reason triggering each checkpoint. In *Test 1*, the checkpoint is done when a percentage of the work is completed. In Figure 4.6.a checkpoint is done at 20%, 40%, 60% and 80% of computation progress. From this data we conclude that i) the overhead of concurrent checkpoint is negligible - less than 1% in all configurations, and ii) the overhead of the serial checkpoint has a decreasing impact on the application's execution time, as the number of total iterations increases. This evident decrease is due to the fact that, as computation time increases, the fixed number of serial checkpoints taken (4) will have progressively smaller impact on the total execution time.

Nevertheless, as application total execution time increases, triggering checkpoint with percentage of progress may lead, in case of a failure, to significant loss of work performed and data (i.e., all the computation done since the previous checkpoint and its results). Furthermore, the percentage of progress may be difficult to estimate in most applications, and would require explicit checkpoint invocation by programmers.

To avoid all this, the checkpoint should be triggered whenever a given time has elapsed, e.g., roughly every 5 minutes. This scenario is represented by *Test 2* checkpointing. Results are presented in Figures 4.6.b. Here, since longer executions imply more checkpoints taken (with 5 minute periodicity), the serial checkpoint now increasingly stretches the total execution time of the application (up to 70% more, broadly), while the overhead introduced by the concurrent checkpoint always remains very low.

So, to applications that need frequent checkpoint, given their longer total execution time and larger working set size, the concurrent checkpoint is a very effective alternative. Furthermore, given that all approaches described in the literature are serial in nature, their performance would always be much worse than our new proposal, added to the fact that they also lack on transparency and completeness, namely: i) either imposing the usage of an API, or ii) requiring extension of class code by programmers, or iii) not supporting multithreaded and cooperative, synchronized applications.

4.4 Cluster-wide thread placement

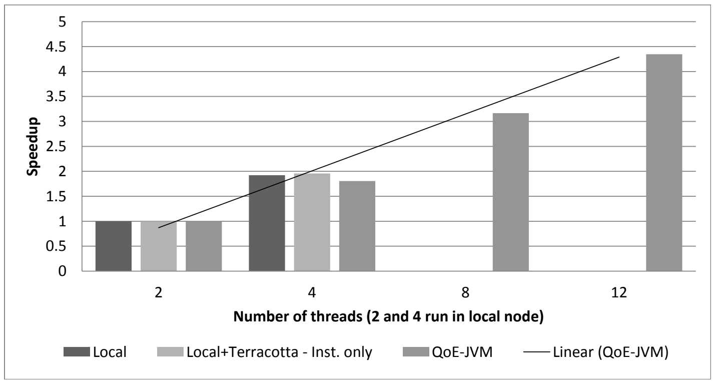


Fig. 4.7. Fibonacci - Execution times

For the execution time measurements, we configured an application to compute the first 1200 numbers of the Fibonacci sequence, with a number of threads equal to the number of cores available. With two and four threads, the application is executed in a single node. In Figure 4.7 the first two series shows the results in a standard local JVM, for comparison purposes with our distributed solution. Next, we tested our application using only the Terracotta middleware, to have a general idea of how the usage of the original Terracotta platform impacts the performance (this is the price to pay for the memory scalability and elasticity it provides).

Finally, the *QoE-JVM* series includes the Terracota instrumentation and our shared structures to schedule threads across a total of 3 node (8 threads take 2 nodes and 12 threads will need 3 nodes). The overhead introduced is low, as we only share a relatively small array in each thread for storing the Fibonacci numbers, along with some auxiliary variables. By adding our middleware, we introduce an extra overhead which is not very significant, and as such, it is possible to obtain smaller execution times by adding more nodes to the Terracotta cluster.

Summary

In this chapter we have demonstrated the potential of the three major mechanisms to manage resources in a cluster where several instances of a HLL-VM run in competition for a limited number of resources. Although all mechanisms impose a degradation penalty to the execution time of applications, this penalty is in most cases small.

5 Conclusions and Future Work

In this document, we described the ongoing research to design a distributed execution environment, where each node executes an extended resource-aware runtime for a managed language, Java, and where resources are allocated based on their effectiveness for a given workload. We presented the architecture of *QoE-JVM* with the ability to monitor base mechanisms (e.g. CPU, memory or network consumptions) in order to assess application's performance and reconfigure these mechanisms in runtime.

Resource allocation and adaptation obeys to a VM economics model, based on aiming overall quality-of-execution (QoE) through resource efficiency. Our adaptation model, based on the yield obtained from applying different strategies to each tenant's workload, aims at putting resources where they can do the most good to applications and the cloud infrastructure provider, while taking them from where they can do the least harm to applications.

A more coarse-grained resource management action is checkpoint and migrate the application execution state to another node in the cluster. We have extended a Java VM with checkpointing (serial and concurrent), restore and migration mechanisms that can be employed with transparency to the programmers which need not modify their applications. The proposed solution was implemented and we evaluated its adequacy and performance, with encouraging results.

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

Semantically, this execution environment provides a partitioned global address space where an application uses resources in several nodes, where objects are shared, and threads are spawned and scheduled globally. Regarding its operation, *QoE-JVM* resorts to a policy-driven adaptability engine that drives resource management, global scheduling of threads, and determines the activation of other coarse-grained mechanisms (e.g., checkpointing and migration among VMs).

5.1 Future Work

Several topics can be regarded as future work. First, overall new research directions that have the potential to bring benefits to our system. Second, some of the current design choices can also be improved. The following are examples that fit into these two areas.

Scheduling with Client Selected Utility Functions

The work presented in Section 3.2 aims to identity the workloads which will have a smaller impact when resources are removed. To have a more complete scheduling decision we should take into account the clients perceived utility to service degradation (e.g. virtual machine with less capacity). He can consider different classes of users, each with a different proportion between utility and service degradation. These different classes (i.e. utility functions) would have a corresponding price, which would be higher for functions that with a small degradation in service have a large degradation in utility.

Integration with state of the art cloud scheduling simulators

Currently, state of the art cloud scheduling simulators, such as the CloudSim [11], assume that workloads make progress only based on the assigned CPU (i.e. number of instructions per second). We believe that the simulator progress model is still too simple, because it does not account for degradation due to memory or even bandwidth shortage, trashing or congestion. There must be further investigating in how we can simulate such behavior, taking into account the progress measurements versus allocated resources, based on real workloads.

Statistical methods applied to the selection of heap growth matrices.

Statistical learning methods, also known as machine learning, can be used to select the matrix that best fits an unknown workload. For each of the current evaluated workloads, a classification of their static and dynamic structure is described in the DaCapo benchmark paper [13]. Each of the metrics has the potential to be an analysis variable of a supervised learning experience. Using, these variables, and the performance results of using different matrices, we could then use the results from previous learning experiences and choose what is expected to be the best matrix for the current workload.

Correlation of threads for placement and migration

In a multi-threaded application there will be threads interacting among them selfs more than with others. Thread interaction can be measured by counting the number (frequency, etc.) of accesses to objects protected by monitors. Identifying these threads is important to collocate them when threads are spawned over the cluster. Also, during the execution of long running programs, highly correlated threads could be migrated, in groups, to less loaded nodes. Currently, our checkpoint and migration solution takes into account all the HLL-VM state but it would be useful to migrate only selected threads.

Automatic Confinement of Thread Local Variables

The thread scheduling solution still needs to be better accessed with applications whose threads have some interaction. This will surely show the need for manual or automatic identification of memory accessed by multiple threads. Offline automatic processes like thread escape analysis [24, 50], and online dynamic analysis regarding the locality and accesses to objects (which can be leveraged from information available at the JIT compiler) have great potential.

References

1. Kaffe virtual machine, http://www.kaffe.org/, visited 29-05-2012, 2012.
2. B. Alpern, C.R. Attanasio, J.J. Barton, M.G. Burke, P. Cheng, J.D. Choi, A. Cocchi, S.J. Fink, D. Grove, M. Hind, et al. The Jalapeno virtual machine. *IBM Systems Journal*, 39(1):211, 2000.
3. B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, and V. Sarkar. The jikes research virtual machine project: building an open-source research community. *IBM Syst. J.*, 44:399–417, January 2005.
4. G. M. Amdahl, G. A. Blaauw, and F. P. Brooks. Architecture of the ibm system/360. *IBM J. Res. Dev.*, 8:87–101, April 1964.
5. Yariv Aridor, Michael Factor, and Avi Teperman. cjvm: a single system image of a jvm on a cluster. In *In Proceedings of the International Conference on Parallel Processing*, pages 4–11, 1999.
6. Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Architecture and Policy for Adaptive Optimization in Virtual Machines. Technical Report 23429, IBM Research, November 2004.
7. Matthew Arnold, Stephen J. Fink, David Grove, Michael Hind, and Peter F. Sweeney. A survey of adaptive optimization in virtual machines. In *Proceedings of the IEEE, 93(2), 2005. Special Issue on Program Generation, Optimization, ans Adaptation*, 2005.
8. Godmar Back and Wilson C. Hsieh. The kaffeos java runtime system. *ACM Trans. Program. Lang. Syst.*, 27:583–630, July 2005.
9. Henry G. Baker. Thermodynamics and garbage collection. *SIGPLAN Not.*, 29:58–63, April 1994.
10. Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37:164–177, October 2003.
11. Anton Beloglazov and Rajkumar Buyya. Optimal online deterministic algorithms and adaptive heuristics for energy and performance efficient dynamic consolidation of virtual machines in cloud data centers. *Concurrency and Computation: Practice and Experience*, 24(13):1397–1420, 2012.
12. Walter Binder, Jarle Hulaas, Philippe Moret, and Alex Villazón. Platform-independent profiling in a virtual execution environment. *Softw. Pract. Exper.*, 39:47–79, January 2009.
13. S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 169–190, New York, NY, USA, October 2006. ACM Press.

14. Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 169–190, New York, NY, USA, 2006. ACM.
15. Stephen M. Blackburn and Kathryn S. McKinley. Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 22–32, New York, NY, USA, 2008. ACM.
16. Charles Blake and Rodrigo Rodrigues. High availability, scalable storage, dynamic peer networks: Pick two. In Michael B. Jones, editor, *HotOS*, pages 1–6. USENIX, 2003.
17. Jonas Bonér and Eugene Kuleshov. Clustering the Java Virtual Machine using Aspect-Oriented Programming. In *AOSD '07: Industry Track of the 6th international conference on Aspect-Oriented Software Development*. Conference on Aspect Oriented Software Development, March 2007.
18. Eric A. Brewer. A certain freedom: thoughts on the cap theorem. In Andréa W. Richa and Rachid Guerraoui, editors, *PODC*, page 335. ACM, 2010.
19. Eric Bruneton, Romain Lenglet, and Thierry Coupaye. Asm: A code manipulation tool to implement adaptable systems. In *In Adaptable and extensible component systems*, 2002.
20. Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. *SIGPLAN Notices*, 40(10):519–538, October 2005.
21. Lydia Y Chen, Giuseppe Serazzi, Danilo Ansaloni, Evgenia Smirni, and Walter Binder. What to expect when you are consolidating: effective prediction models of application performance on multicores. *Cluster Computing*, pages 1–19, 2013.
22. Ludmila Cherkasova, Diwaker Gupta, and Amin Vahdat. Comparison of the three cpu schedulers in xen. *SIGMETRICS Perform. Eval. Rev.*, 35:42–51, September 2007.
23. Dah-Ming Chiu and Raj Jain. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Comput. Netw. ISDN Syst.*, 17(1):1–14, June 1989.
24. Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for java. In *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '99, pages 1–19, New York, NY, USA, 1999. ACM.
25. C.W. Cobb and P.H. Douglas. A theory of production. *The American Economic Review*, 18(1):139–165, 1928.
26. G. Czajkowski, M. Wegiel, L. Daynes, K. Palacz, M. Jordan, G. Skinner, and C. Bryce. Resource management for clusters of virtual machines. In *Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid - Volume 01*, CCGRID '05, pages 382–389, Washington, DC, USA, 2005. IEEE Computer Society.
27. Grzegorz Czajkowski, Stephen Hahn, Glenn Skinner, Pete Soper, and Ciarán Bryce. A resource management interface for the java platform. *Softw. Pract. Exper.*, 35:123–157, February 2005.
28. Grzegorz Czajkowski and Thorsten von Eicken. Jres: a resource accounting interface for java. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '98, pages 21–35, New York, NY, USA, 1998. ACM.

29. L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the smalltalk-80 system. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '84, pages 297–302, New York, NY, USA, 1984. ACM.
30. H.A. Duran-Limon, M. Siller, G.S. Blair, A. Lopez, and J.F. Lombera-Landa. Using lightweight virtual machines to achieve resource adaptation in middleware. *IET Software*, 5(2):229–237, 2011.
31. Grzegorz Czajkowski et al. Java specification request 284 - resource consumption management api, 2009.
32. N. Geoffray, G. Thomas, G. Muller, P. Parrend, S. Frenot, and B. Folliot. I-JVM: a Java Virtual Machine for component isolation in OSGi. In *IEEE/IFIP International Conference on Dependable Systems & Networks*, 2009.
33. Robert P. Goldberg. Survey of virtual machine research. *Computer*, 7(9):34–45, September 1974.
34. Zhenhuan Gong, Xiaohui Gu, and J. Wilkes. Press: Predictive elastic resource scaling for cloud systems. In *Network and Service Management (CNSM), 2010 International Conference on*, pages 9 –16, oct. 2010.
35. Kinshuk Govil, Dan Teodosiu, Yongqiang Huang, and Mendel Rosenblum. Cellular disco: resource management using virtual clusters on shared-memory multiprocessors. *SIGOPS Oper. Syst. Rev.*, 33:154–169, December 1999.
36. Dominik Gront and Andrzej Kolinski. Utility library for structural bioinformatics. *Bioinformatics*, 24(4):584–585, 2008.
37. Chris Grzegorczyk, Sunil Soman, Chandra Krintz, and Rich Wolski. Isla vista heap sizing: Using feedback to avoid paging. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, pages 325–340, Washington, DC, USA, 2007. IEEE Computer Society.
38. Xiaohua Guan, Witawas Srisa-an, and Chenghuan Jia. Investigating the effects of using different nursery sizing policies on performance. In *Proceedings of the 2009 international symposium on Memory management*, ISMM '09, pages 59–68, New York, NY, USA, 2009. ACM.
39. Ajay Gulati, Arif Merchant, Mustafa Uysal, and Peter J. Varman. Efficient and adaptive proportional share i/o scheduling. Technical report, HP Laboratories Palo Alto, 2007.
40. Mahantesh Halappanavar, John Feo, Oreste Villa, Antonino Tumeo, and Alex Pothen. Approximate weighted matching on emerging manycore and multithreaded architectures. *Int. J. High Perform. Comput. Appl.*, 26(4):413–430, November 2012.
41. Matthew Hertz, Jonathan Bard, Stephen Kane, Elizabeth Keudel, Tongxin Bai, Kirk Kelsey, and Chen Ding. Waste not,want not: resource-based garbage collection in a shared environment. Technical Report TR-2006-908, University of Rochester, 2009.
42. Matthew Hertz, Yi Feng, and Emery D. Berger. Garbage collection without paging. *SIGPLAN Not.*, 40:143–153, June 2005.
43. Matthew Hertz, Stephen Kane, Elizabeth Keudel, Tongxin Bai, Chen Ding, Xiaoming Gu, and Jonathan E. Bard. Waste not, want not: resource-based garbage collection in a shared environment. In *Proceedings of the international symposium on Memory management*, ISMM '11, pages 65–76, New York, NY, USA, 2011. ACM.
44. Michael Hines, Abel Gordon, Marcio Silva, Dilma Da Silva, Kyung Dong Ryu, and Muli Ben-Yehuda. Applications know best: Performance-driven memory overcommit with ginkgo. In *CloudCom '11: 3rd IEEE International Conference on Cloud Computing Technology and Science*, 2011.
45. Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin C. Rinard. Dynamic knobs for responsive power-aware computing. In *ASPLOS*, pages 199–212, 2011.
46. Richard C. G. Holland, Thomas A. Down, Matthew R. Pocock, Andreas Prlic, David Huen, Keith James, Sylvain Foisy, Andreas Dräger, Andy Yates, Michael Heuer, and Mark J. Schreiber. Biojava: an open-source framework for bioinformatics. *Bioinformatics*, 24(18):2096–2097, 2008.

47. Jarle Hulaas and Walter Binder. Program transformations for light-weight cpu accounting and control in the java virtual machine. *Higher Order Symbol. Comput.*, 21:119–146, June 2008.
48. Vatche Ishakian, Raymond Sweha, Azer Bestavros, and Jonathan Appavoo. Cloudpack: Exploiting workload flexibility through rational pricing. In Priya Narasimhan and Peter Triantafillou, editors, *Middleware 2012*, volume 7662 of *Lecture Notes in Computer Science*, pages 374–393. Springer Berlin Heidelberg, 2012.
49. Ravi Iyer, Ramesh Illikkal, Omesh Tickoo, Li Zhao, Padma Apparao, and Don Newell. Vm3: Measuring, modeling and managing vm shared resources. *Computer Networks*, 53(17):2873–2887, December 2009.
50. Pramod G. Joisha, Robert S. Schreiber, Prithviraj Banerjee, Hans J. Boehm, and Dhruva R. Chakrabarti. A technique for the effective and automatic reuse of classical compiler optimizations on multithreaded code. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 623–636, New York, NY, USA, 2011. ACM.
51. Philipp Leitner, Benjamin Satzger, Waldemar Hummer, Christian Inzinger, and Schahram Dustdar. Cloudscale: a novel middleware for building transparently scaling cloud applications. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 434–440, New York, NY, USA, 2012. ACM.
52. Ivan López-Arévalo, René Bañares-Alcántara, Arantza Aldea, and A. Rodríguez-Martínez. A hierarchical approach for the redesign of chemical processes. *Knowl. Inf. Syst.*, 12(2):169–201, 2007.
53. Martina Maggio, Henry Hoffmann, Alessandro V. Papadopoulos, Jacopo Panerati, Marco D. Santambrogio, Anant Agarwal, and Alberto Leva. Comparison of decision-making strategies for self-optimization in autonomic computing systems. *ACM Trans. Auton. Adapt. Syst.*, 7(4):36:1–36:32, December 2012.
54. Jeremy Manson, William Pugh, and Sarita V. Adve. The java memory model. *SIGPLAN Not.*, 40:378–391, January 2005.
55. Feng Mao, Eddy Z. Zhang, and Xipeng Shen. Influence of program inputs on the selection of garbage collectors. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '09, pages 91–100, New York, NY, USA, 2009. ACM.
56. Abel Gordon Michael Hines, Marcio Silva, Dilma Da Silva, Kyung Dong Ryu, and Muli Ben-Yehuda. Applications know best: Performance-driven memory overcommit with ginkgo. In *CloudCom '11: 3rd IEEE International Conference on Cloud Computing Technology and Science*, 2011.
57. Microsoft. Clr profiler for the .net framework 2.0.
58. Oracle. Java virtual machine tool interface (JVMTI), http://download.oracle.com/javase-/6/docs/technotes/guides/jvmti/.
59. Simon Ostermann and Radu Prodan. Impact of variable priced cloud resources on scientific workflow scheduling. In Christos Kaklamanis, Theodore Papatheodorou, and Paul Spirakis, editors, *Euro-Par 2012 Parallel Processing*, volume 7484 of *Lecture Notes in Computer Science*, pages 350–362. Springer Berlin / Heidelberg, 2012.
60. Pradeep Padala, Kai-Yuan Hou, Kang G. Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, and Arif Merchant. Automated control of multiple virtualized resources. In *Proceedings of the 4th ACM European conference on Computer systems*, EuroSys '09, pages 13–26, New York, NY, USA, 2009. ACM.
61. Carlos Ribeiro, André Zúquete, Paulo Ferreira, and Paulo Guedes. Spl: An access control language for security policies with complex constraints. In *In Proceedings of the Network and Distributed System Security Symposium*, pages 89–107, 1999.
62. Mohsen Amini Salehi, Bahman Javadi, and Rajkumar Buyya. Resource provisioning based on preempting virtual machines in distributed systems. *Concurrency and Computation: Practice and Experience*, pages n/a–n/a, 2013.
63. Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4:14:1–14:42, May 2009.

64. Zhiyuan Shao, Hai Jin, and Yong Li. Virtual machine resource management for high performance computing applications. *Parallel and Distributed Processing with Applications, International Symposium on*, 0:137–144, 2009.
65. J.N. Silva, P. Ferreira, and L. Veiga. Service and resource discovery in cycle-sharing environments with a utility algebra. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–11, 2010.
66. João Nuno Silva, Luís Veiga, and Paulo Ferreira. A^2ha - automatic and adaptive host allocation in utility computing for bag-of-tasks. *J. Internet Services and Applications*, 2(2):171–185, 2011.
67. Jeremy Singer, Richard E. Jones, Gavin Brown, and Mikel Luján. The economics of garbage collection. *SIGPLAN Not.*, 45:103–112, June 2010.
68. Jeremy Singer, George Kovoor, Gavin Brown, and Mikel Luján. Garbage collection auto-tuning for java mapreduce on multi-cores. In *Proceedings of the international symposium on Memory management*, ISMM '11, pages 109–118, New York, NY, USA, 2011. ACM.
69. Jim Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann, 2005.
70. Sunil Soman and Chandra Krintz. Application-specific garbage collection. *J. Syst. Softw.*, 80:1037–1056, July 2007.
71. Sunil Soman, Chandra Krintz, and David F. Bacon. Dynamic selection of application-specific garbage collectors. In *Proceedings of the 4th international symposium on Memory management*, ISMM '04, pages 49–60, New York, NY, USA, 2004. ACM.
72. I. Stoica, H. Abdel-Wahab, and K. Jeffay. On the duality between resource reservation and proportional share resource allocation. Technical report, Norfolk, VA, USA, 1996.
73. Jimmy Su and Katherine Yelick. Automatic support for irregular computations in a high-level language. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05)*, IPDPS '05, Washington, DC, USA, 2005. IEEE Computer Society.
74. Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007.
75. Enric Tejedor, Montse Farreras, David Grove, Rosa M. Badia, Gheorghe Almasi, and Jesus Labarta. A high-productivity task-based programming model for clusters. *Concurrency and Computation: Practice and Experience*, pages 2421–2448, 2012.
76. VMware. Vmware vspher 4: The cpu scheduler in vmware esx 4.
77. Carl A. Waldspurger. *Lottery and stride scheduling: flexible proportional-share resource management*. PhD thesis, 1995. AAI0576752.
78. Carl A. Waldspurger. Memory resource management in vmware esx server. *SIGOPS Oper. Syst. Rev.*, 36:181–194, December 2002.
79. Zhao Weiming and Wang Zhenlin. Dynamic memory balancing for virtual machines. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '09, pages 21–30, 2009.
80. Paul R. Wilson. Uniprocessor garbage collection techniques. In *Proceedings of the International Workshop on Memory Management*, IWMM '92, pages 1–42, London, UK, 1992. Springer-Verlag.
81. Jie Xu, Paul Townend, Junaid Arshad, and Wei Jie. Cloud computing security: Opportunities and pitfalls. *Int. J. Grid High Perform. Comput.*, 4(1):52–66, January 2012.
82. Xiang Zeng, Rajive Bagrodia, and Mario Gerla. Glomosim: a library for parallel simulation of large-scale wireless networks. *SIGSIM Simul. Dig.*, 28(1):154–161, July 1998.
83. Hua Zhang, Joohan Lee, and Ratan Guha. Vcluster: a thread-based java middleware for smp and heterogeneous clusters with thread migration support. *Softw. Pract. Exper.*, 38:1049–1071, August 2008.

84. Yuting Zhang, Azer Bestavros, Mina Guirguis, Ibrahim Matta, and Richard West. Friendly virtual machines: leveraging a feedback-control model for application adaptation. In *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, VEE '05, pages 2–12, New York, NY, USA, 2005. ACM.

85. Wenzhang Zhu, Cho-Li Wang, and Francis C. M. Lau. Jessica2: A distributed java virtual machine with transparent thread migration support. *Cluster Computing, IEEE International Conference on*, 0:381, 2002.