TÉCNICO LISBOA

A Java Runtime Environment for Cloud Computing

José Manuel de Campos Lages Garcia Simão

59714/D

Fevereiro de 2012

Contents

Abstract

place holder text from papers

Applications running on cluster-enabled infrastructures (e.g. Cloud computing) are supported by different levels of virtualization. In these environments, applications run on high language virtual machines (e.g. JVM, CLR), which make use of (guest) operating system services. At this stack level, common hardware is shared through system virtual machine (e.g. Xen, VMWare Server).

The effective allocation of resources to fit the application's needs (e.g. execution time, monetary cost) and owner's privileges is a challenging task. System virtual machines provide some 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).

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

Chapter 1

Introduction

In today's scenarios of large scale computing and service providing, the deployment of 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.

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 componentized 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, and even in e-Science fields (e.g., with more and more usage of Java in the context of physics simulation, economics/statistics, network simulation, chemistry, computational biology and bio-informatics [Holland et al., 2008, Gront and Kolinski, 2008, López-Arévalo et al., 2007], there being already many available Java-based APIs such as Neobio).[1]

To extend the benefits of a local VEE, while allowing scale-out regarding performance and memory requirements, many solutions have been proposed to federate Java virtual machines [Zhang et al., 2008, Aridor et al., 1999, Zhu et al., 2002], aiming to provide a single system image where the managed application can benefit from the global resources

[1] http://www.bioinformatics.org/neobio/

of the cluster. If this 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 object oriented virtual machine (OO-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).

A VEE 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 on their resource consumption. These two non functional requirements can only be fulfilled if the cluster can monitor and control the resources it uses both at the VEE and distributed level, and whether the several local VEE, each running on its node, are able to cooperate to manage resources overall.

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.

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. Therefore, as the majority of the object-oriented programming languages execute their applications on object-oriented virtual machines (OO VM, also known as HLL VM[2], e.g. Java VM, .NET CLR), this thesis proposes proposes an approach to the checkpoint and migration mechanisms at this level.

Existing public runtimes (Java, CLR) are not resource-aware. They are mostly bounded by the underlying operative system. On the other hand, in the research community, the proposed runtimes are focused on accounting resource usage to avoid application's bad behavior, and do not support the desired reconfigurability of their inner mechanisms [Geoffray et al., 2009, Binder et al., 2009]. There is no notion of *re-*

[2]High-Level Language Virtual Machine.

source effectiveness in the sense that when there are scarce resources, there is no attempt to determine where to take such resources from applications (i.e. either isolation domains or the whole VM) where they hurt performance the least. Others have recently shown the importance of adaptability at the level of HLL-VMs, either based on the application performance [Michael Hines et al., 2011] or by changes in their environment [Duran-Limon et al., 2011]. Nevertheless, they are either dependent on a global optimization phase, limited to a given resource, or make the application dependent on a new programming interface.

Therefore, such a cluster-enabled managed environment can adapt itself to the execution of applications, from multiple tenants, with different (and sometimes dynamically changing) requirements in regard to their quality-of-execution (QoE). QoE aims at capturing the adequacy and efficiency of the resources provided to an application according to its needs. It can be inferred coarsely from application execution time for medium running applications, or request execution times for more service driven ones such as those web-based, or from critical situations such as thrashing or starvation. Also, it can be derived with more fine-grain from incremental indicators of application progress, such as execution phase detection [Nagpurkar et al., 2006], memory pages updates, amount of input processed, disk and network output generated. In our ongoing work, we are still focusing only on application execution times.

QoE can be used to drive a VM economics model, where the goal is to incrementally obtain gains in QoE for VMs running applications requiring more resources or for more privileged tenants. This, while balancing the relative resource savings drawn from other 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 some performance penalties (that should also be mitigated). In any case, these changes should be transparent to the developer and specially to the application's user.

1.1 Published work

The work described in this document was partially published in the following articles:

1. [Simão et al., 2011] describes the architecure and implementation details of the cluster enabling mechanisms and how they are activated by declarative policy loaded by the JVM.
2. [Simão et al., 2012] 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. [Simão and Veiga, 2012b] and the poster [Simão and Veiga, 2012a] describe ongoing work regarding a simple economic model and the adaptation mechanisms presented in Chapter 4.

After enrollment in the PhD program I have also published the work presented in my master dissertation.

1. [Simão et al., 2010]

1.2 Outline

Chapter 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. At the 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 because of techniques like runtime component loading or automatic memory management.

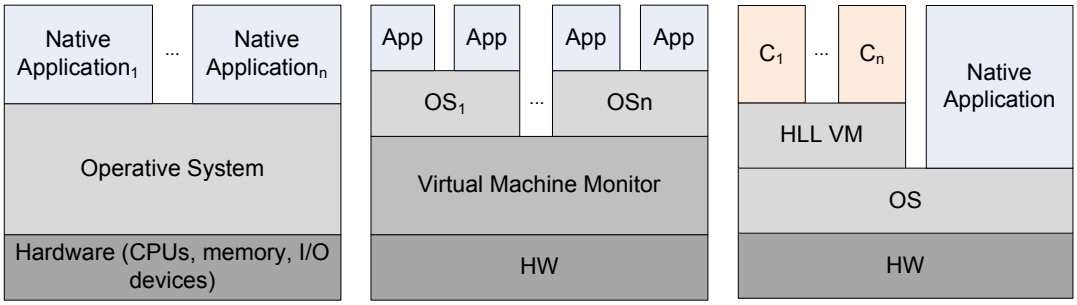
System level VMs, or hypervisors, are strongly motivated by the sharing of low-level resources. In result of this, many research and industry work can be found about how resources are to be delivered to each guest operating system. The partition is done with different reasonings, ranging from a simple *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 [Zhang et al., 2005, Cherkasova et al., 2007, Shao et al., 2009, VMware,] and memory [Govil et al., 1999, Waldspurger, 2002, Hines et al., 2011] 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. [Czajkowski and von Eicken, 1998, Czajkowski et al., 2005, Binder et al., 2009, Hulaas and Binder, 2008]). 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 or on the portability of the guests (i.e. applications).

Virtual machines are not only a isomorphism between the guest system and a host [Smith and Nair, 2005], 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 adaptive loop [Salehie and Tahvildari, 2009]: 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 have 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



(a) (b) (c)

Figure 2.1: Virtualization layers

the migration of the guest VM to another node.

In this work we present a framework to classify resource monitoring and adaptation techniques in virtual machines, both at system and language level. Section 2.2 presents the architecture of these VMs, depicting the building blocks that are used in research regarding resource usage and adaption. In Section 2.3 the classification framework is presented. For each of the resources considered, 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. Furthermore, systems are globally classified regarding the dependency on low level monitoring, the complexity of control techniques and the latency of the enforcement mechanisms. We use this framework to classify state of the art systems in Section 2.4. Section 2.5 closes the document 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 [Amdahl et al., 1964]. 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 underpinned on this resources.

Today, the same architecture can be found in modern System VMs [Barham et al., 2003]. 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 [Deutsch and Schiffman, 1984], 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 [Mao et al., 2009, Hertz et al., 2009, Soman et al., 2004].

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 2.4 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 [Manson et al., 2005] making it possible to rationalize 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 [Barham et al., 2003, Cherkasova et al., 2007]. 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 [Tanenbaum, 2007]. 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 bigger than the available physical CPUs. Similarly to a thread, the VCPU can be running, ready or waiting. 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 aditional requisites 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 [Stoica et al., 1996, Cherkasova et al., 2007]. This family of schedulers are named *Proportional Share*. Operating systems have traditionally used a related type of share scheduling, named *Fair Share*. However, in these schedulers, shares are not directly seen by the end user making it hard to define a high level resource management policy [Waldspurger, 1995].

In [Cherkasova et al., 2007], schedulers are further classified 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 2.4 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 language VMs with a similar goal: give the illusion to their guests of virtually unbounded address space. Because memory is effectively limited, it will eventually end 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 bigger than the real available hardware. As pointed out in [Smith and Nair, 2005], the VMM extra level of indirection generalizes the virtual memory mechanisms of operating systems.

In a language VM, 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 process 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 language VMs.

Memory management in System VMs

The VMM can be managing multiple VMs, each with his guest OS. 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 $virtual \rightarrow real$ mapping of each OS, keeping a $real \rightarrow physical$ 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 $virtual \rightarrow physical$ addresses. Access to the page table pointer is virtualized by the VMM, trapping read or write attempts and returning the corresponding table pointer of the running VM. The translation look aside buffer (TLB) continues to play his accelerating role because it will still cache the $virtual \rightarrow physical$ 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 2.4 present 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 Language VMs

The goal of memory's virtualization in high language VMs is to free the application from explicit dealing with memory deallocation, giving the perception of an unlimited address space. This avoids keeping track of 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 [Wilson, 1992, Smith and Nair, 2005].

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 searching all the heap. New objects are created in a small space (e.g. 512 KBytes). When this space fills, 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. Recently, and as parallel hardware becomes ubiquitous, the stop the world approach seems obsolete. Concurrent collectors have been designed and are used in multiprocessors platforms [Click et al., 2005].

Investigators have been analyzing the impact of different data inputs on the performance of garbage collectors [Mao et al., 2009]. Based on these observations, several adaptation strategies have been proposed [Arnold et al., 2005], ranging from parameters adjustments (e.g. the nursery size [Guan et al., 2009]) to changing the algorithm itself in runtime [Soman and Krintz, 2007]. Section 2.4 discusses these different approaches.

2.3 Adaptation techniques and taxonomy

In a software system, adaptation is regulated by monitoring, analyzing, deciding and acting [Salehie and Tahvildari, 2009]. Monitoring is feed 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 language VMs [Arnold et al., 2005]. An example of the latter is the work of Shao et al. [Shao et al., 2009] to regulate VCPU to CPU mapping based on the CPU usage of specific applications.

There is a broad range of strategies regarding the analysis and decision processes. Many solutions that augment System VMs use control theory 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 adaption processes: monitoring, analyzing/deciding and acting.

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 [Barham et al., 2003, Cherkasova et al., 2007]. 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) [Zhang et al., 2005, Shao et al., 2009], use their own control strategy, such as linear optimization, control theory [Padala et al., 2009] or statistical methods [Gong et al., 2010]. 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 aims to reveal how pages are being used by each VM and so information collected relates to *i)* page utilization [Waldspurger, 2002, Weiming and Zhenlin, 2009] and *ii)* page contents equality or similarity [Waldspurger, 2002, Barham et al., 2003]. Application performance (either by modification of the application or external monitoring) is also considered [Hines et al., 2011].

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, 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 [Waldspurger, 2002], *ii)* history pattern [Weiming and Zhenlin, 2009] or *iii)* linear programming [Hines et al., 2011].

After deciding that a new configuration must be applied to a set of VMs, the VMM can enforce *i)* page sharing [Waldspurger, 2002] 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 [Barham et al., 2003, Waldspurger, 2002].

Summary of adaptation loop techniques

Figure 2.2 presents the techniques used in the adaptation loop. They are grouped into the two major adaptation targets, CPU and memory, and then into the three major phases of the adaptability loop.

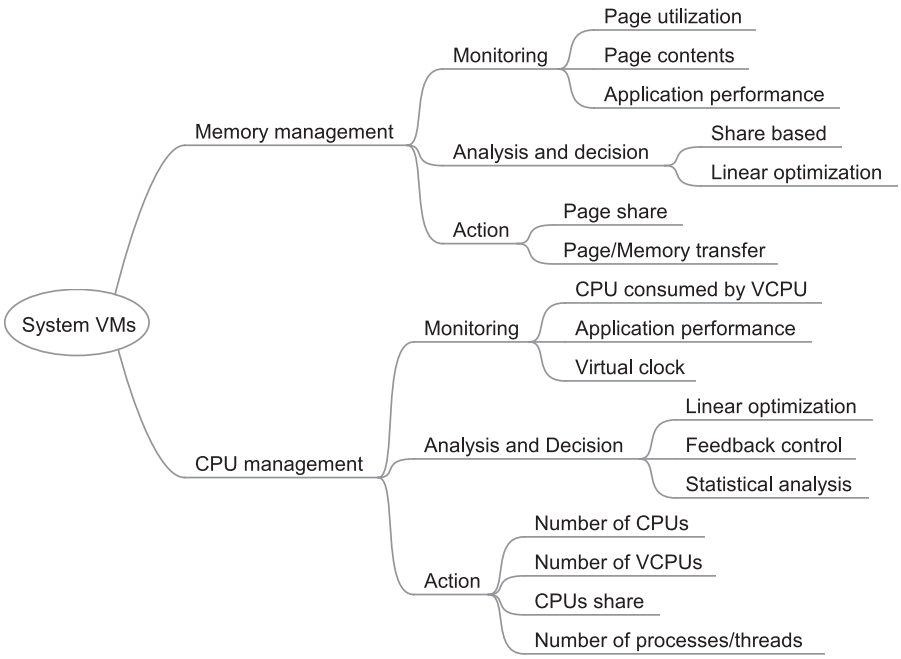


Figure 2.2: Techniques used by System VMs to monitor, control and enforce

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. Language 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 that, although it cannot be formally reasoned about, it gives good results when properly used. Nevertheless, some 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 [Alpern et al., 2005] 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 [Arnold et al., 2005] 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 [Arnold et al., 2004, Arnold et al., 2005]. 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 [Alpern et al., 2005].

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 close the adaptability loop are: *i)* partial or total method recompilation, *ii)* inlining or *iii)* deoptimization.

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 [Arnold et al., 2005]. To accomplish these adaptations, monitoring is done by observing: *i)* memory structures dimensions (e.g. total heap size, nursery size) [Singer et al., 2010, Singer et al., 2011], *ii)* the program behavior (e.g. alloca-

[1]The adaptive optimization system (AOS) in Jikes RVM [Alpern et al., 2005] produces a log with approximately $700Kbytes$ of information regarding call graphs, edge counters and compilation advices when running and JIT compiling one of DaCapo's benchmark [Blackburn et al., 2006] - `bloat`

tion rate, stack height, key objects) [Soman and Krintz, 2007] and, *iii)* relevant events in the operating systems (e.g. page faults, allocation stalls) [Grzegorczyk et al., 2007, Hertz et al., 2011].

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 have 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 object's lifetime. These models are mostly used to give a rationale of the GC behavior, rather than being used in a decision process [Baker, 1994]. So, most systems have a decision process based on some kind of heuristics. The decision process include *i)* machine learning *ii)* control theory 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 work have been focused on optimizing memory usage and execution time, taking in consideration not only the program dynamics and but also its execution environment. Some work also adapts GC to avoid memory exhaustion in environments where memory is constrained. To accomplish this, actions regarding GC adaptability are related to changing: *i)* heap size [Singer et al., 2010], *ii)* GC parameters [Singer et al., 2011] *iii)* GC algorithm [Soman and Krintz, 2007].

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 [Back and Hsieh, 2005, Czajkowski et al., 2005] or *ii)* byte code instrumentation [Hulaas and Binder, 2008]. In the former, it's possible to collect more information, both from a quantitative and qualitative perspective. A well know example is the Java Virtual Machine Tool Interface [Oracle,], which is mainly used by development environments to display debug information. Because language 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.

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 so, 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. 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 high level virtual machines.

2.3.3 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

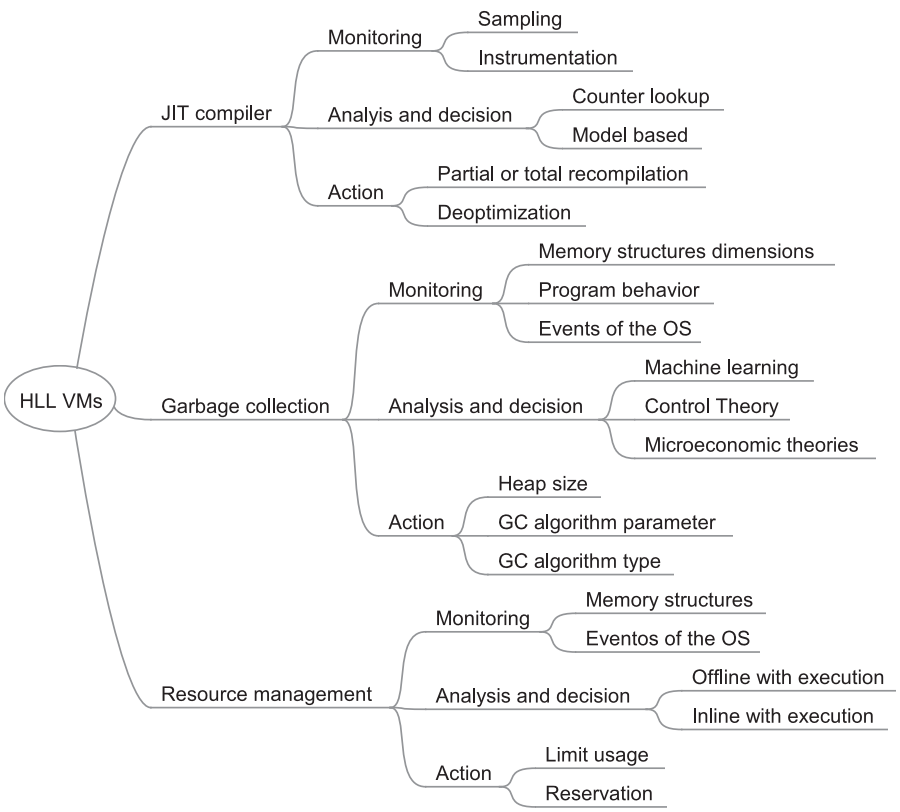


Figure 2.3: Techniques used by HLL VMs to monitor, control and enforce

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). Later, we confirmed that full responsiveness always implies some level of restriction either to comprehensiveness or to intricateness. This *RCI conjecture* is yet another manifestation in systems research 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) [Brewer, 2010], portraying the tension in large-scale distributed systems among (C)onsistency, (A)vailability, and tolerance to (P)artitions. Another example one is the tension, in the domain of peer-to-peer systems, among high availability, scalability, and support for dynamic populations [Blake and Rodrigues, 2003].

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 **??** 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 **??** shows the meaning of each metric for each of the quantitative values that the framework allows techniques to be classified (i.e. 1, 2 or 3). Quantitative (Q) intervals and Processing Times (PT) used in the framework are presented. Also, two notes are worth noting. First, *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, for the decision step of the control we adapt the criteria of Maggio et al. [Maggio et al., 2012].

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

Table 2.2: Add caption

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 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 MCA 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 this section, regarding both system and high level virtual machines (see Figure 2.2 and 2.3). In each figure, techniques are further categorized into the three phases of the adaptation loop - monitoring, decision, and 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 is used to compare different systems and better understand how virtual machine researchers have explored the tension between responsiveness, comprehensiveness and intricateness.

2.4 Systems and their classification

Virtual Machines in Consolidated Environments

2.4.1 System Virtual Machine

Xen In Xen [Barham et al., 2003] 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.

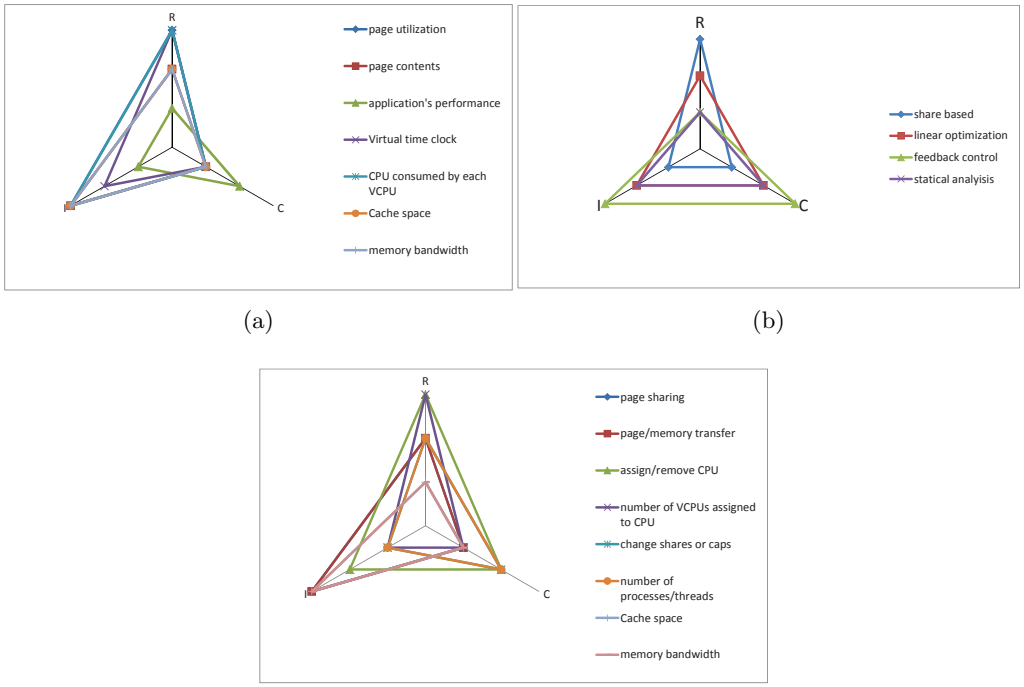
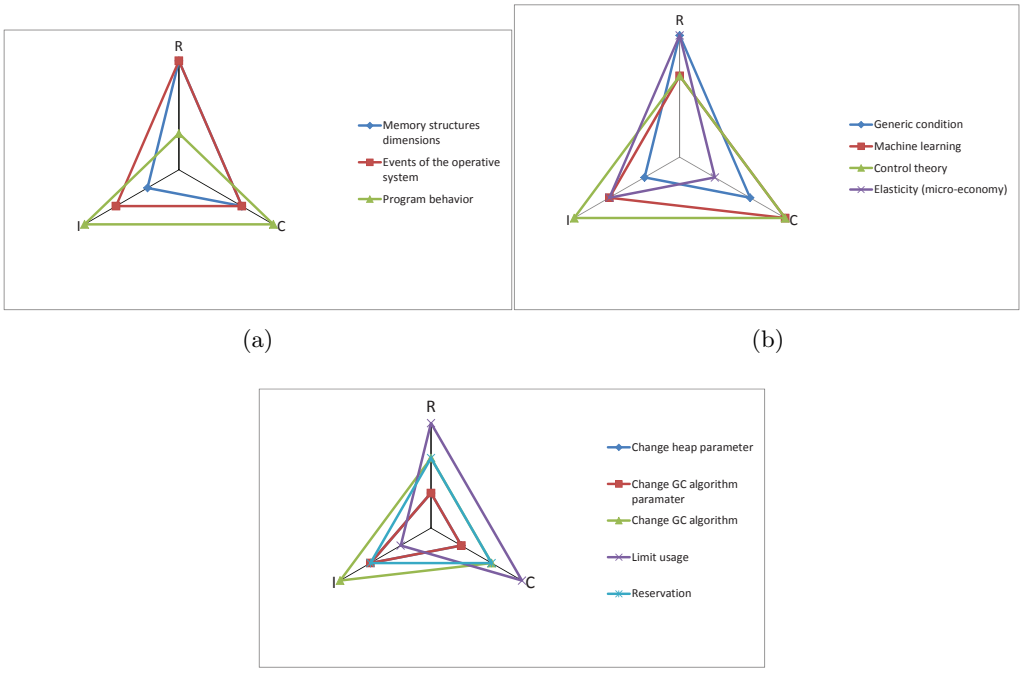


Figure 2.4: Relation of *responsiveness*, *comprehensiveness* and *intricateness* for the different techniques used in System VMs



(a) (b)

(c)

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

Xen includes three scheduling algorithms: Borrow Virtual Time (BVT), Simple Earliest Deadline First (SEDF) and Credit [xen, 2012, Cherkasova et al., 2007]. 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 his 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, a *under* VCPU from the local running queue, a *over* VCPU from the local running queue or a *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) [Zhang et al., 2005] aims to enable efficient and fair usage of the underlying resources. Efficient in the sense that underlying system resources are nor overused or 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), driven by a single control signal - the *Virtual Clock Time* (VCT) to detect overload situation. VCT is the real 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. [Shao et al., 2009] adapts the VCPU mapping of Xen [Barham et al., 2003] 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 [Hines et al., 2011] 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. [Padala et al., 2009] proposes 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 [Gulati et al., 2007].

PRESS PRESS [Gong et al., 2010] 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 transiting to the highest probability state, given the current state. In [Gong et al., 2010] the authors focus on CPU usage. So, 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 [Iyer et al., 2009] aims at measuring, modelling 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.

Summary

2.4.2 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 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 level operations or resource counters already available in the system to accomplish other tasks. After collecting this information, the VM can either restraint 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.

Account 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 [kaf, 2012], KaffeOS [Back and Hsieh, 2005] provides the ability to run Java applications isolated from each other and also to limit their resource consumption. KaffeOS, adds a process model to Java that allows a JVM to run multiple untrusted programs safely. The runtime system is able to account for and control all of the CPU and memory resources consumed on behalf of any process. Consumption of individual processes can be separately accounted for because the allocation and garbage collection activities of different processes are separated. To account memory, KaffeOS uses a hierarchical structure where each process is assigned a hard and 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. [Czajkowski and von Eicken, 1998] 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 systems 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 [Czajkowski et al., 2005] extends the Sun Hotspot JVM to support *isolates* and resource management. *Isolates* are similar to processes in KaffeOS. The distinguishing difference of MVM is his 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 [Czajkowski et al., 2005] 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 [Czajkowski and von Eicken, 1998] 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. [Hulaas and Binder, 2008] 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 presents 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. [Duran-Limon et al., 2011] 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 the application to a given resource adaptation interface.

Garbage collection is known to have different performance impacts in different application [Soman and Krintz, 2007, Mao et al., 2009]. 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) [Hertz et al., 2009, Grzegorczyk et al., 2007, Singer et al., 2010]; *ii)* algorithms that take execution environment events into account [Hertz et al., 2005]; *iii)* VMs that switch the GC algorithm at runtime [Soman and Krintz, 2007]. 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. [Grzegorczyk et al., 2007] 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. [Hertz et al., 2011] 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 in account two conditions: if the resident set has decreased or if the number of page faults have increased. If any of this 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. [Singer et al., 2011] proposes to automatically 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 deciding about a GC policy when a new MapReduce application arrives. The decision is made locally to an instance of the JVM.

GC economics In [Singer et al., 2010], 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.

Application-specific Garbage Collectors [Singer et al., 2007]

GC switch Soman et al. [Soman and Krintz, 2007] 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 to decide when to switch dynamically, and to which GC it should switch. The modified runtime incorporates all the available GCs 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 memory is available) to a Generational Mark-Sweep collector (which performs better when memory is constrained).

Summary

2.4.3 Quantitative comparison of different adaptability techniques

In Figures 2.6 and 2.7, we analyze the *responsiveness*, *comprehensiveness* and *intricateness* aspects for the different adaption 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 less comprehensive. In HLL VMs intricateness is larger in

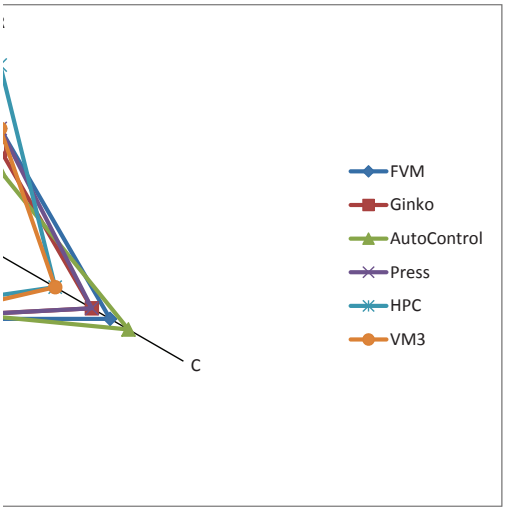


Figure 2.6: Relationship of *responsiveness*, *comprehensiveness* and *intricateness* for the different adaption techniques used in System VMs.

deployments focused on GC. All this shows that the three properties cannot be covered in the same degree.

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*, *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.

2.5 Conclusions

Currently, data centers in the context of cloud infrastructures make extensive use of virtualization to achieve workload isolation and efficient resource management. This is carried out primarily by means of virtualization technology, either at the system or high language level. While isolation is a static mechanism, that relies on hardware or operating system support to be enforced, resource management is dynamic and VMs must self-adapt or be instructed to adapt in order to fit their guest's needs.

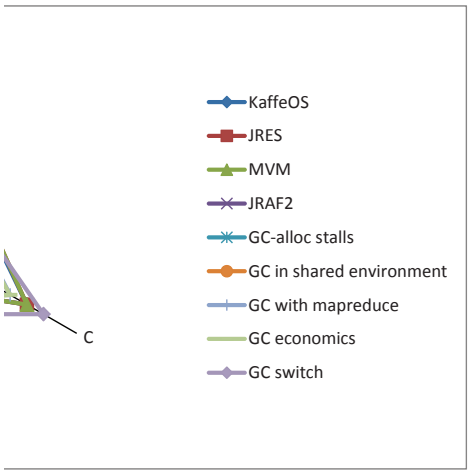


Figure 2.7: Relationship of *responsiveness*, *comprehensiveness* and *intricateness* for the different adaption techniques used in HLL VMs.

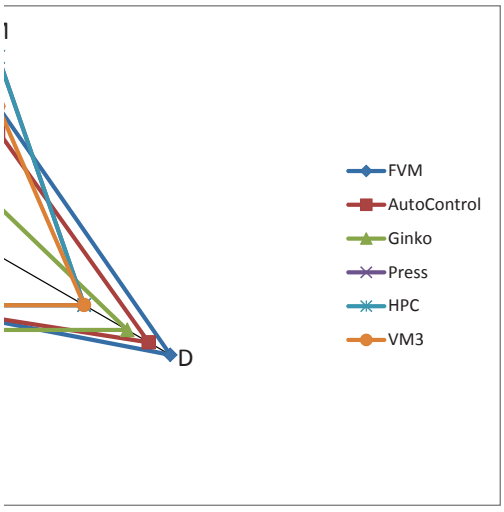


Figure 2.8: Relationship of emphasis on *responsiveness*, *comprehensiveness* and *intricateness*, regarding the adaptation steps *monitoring*, *decision* and *action* in System VMs.

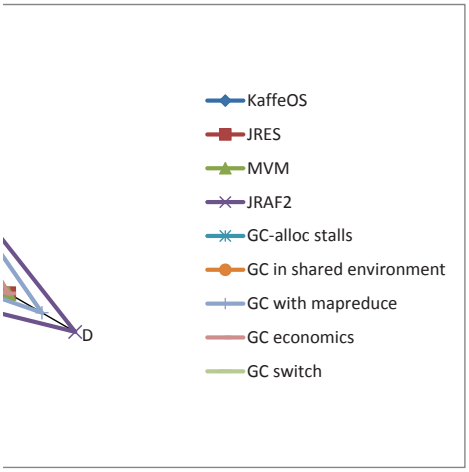


Figure 2.9: Relationship of emphasis on *responsiveness*, *comprehensiveness* and *intricateness*, regarding the adaptation steps *monitoring*, *decision* and *action* in HLL VMs.

Virtualization mechanisms to enforce resource management are present both at hypervisors (e.g. Xen, ESX) and high level virtual machines (e.g. CLR, Java). Although the services offered by each of these two software layers are used or extended in several works in the literature, the community lacks an organized and integrated perspective of the mechanisms and strategies used at each virtualization layer regarding resource management and focusing on adaptation.

In this work 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.

Chapter 3

Architecture of the QoE-JVM

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 has one top-level coordinator, the *QoE Manager* that monitors the *Quality-of-Execution* of applications.[1]

3.1 Overall system view

The overall architecture of *QoE-JVM* (*An adaptive and Resource-Aware Java Virtual Machine*) is presented in Figure **??**. 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* in each node, in charge of VM deployment, lifecycle management, resource monitoring and resource management/restriction. Finally, in order

[1] We opt for this notion instead of hard *service-level agreements* usually employed in commercial application hosting scenarios, because we intend to target several types of applications in shared infrastructures, without necessarily strict contractual requirements.

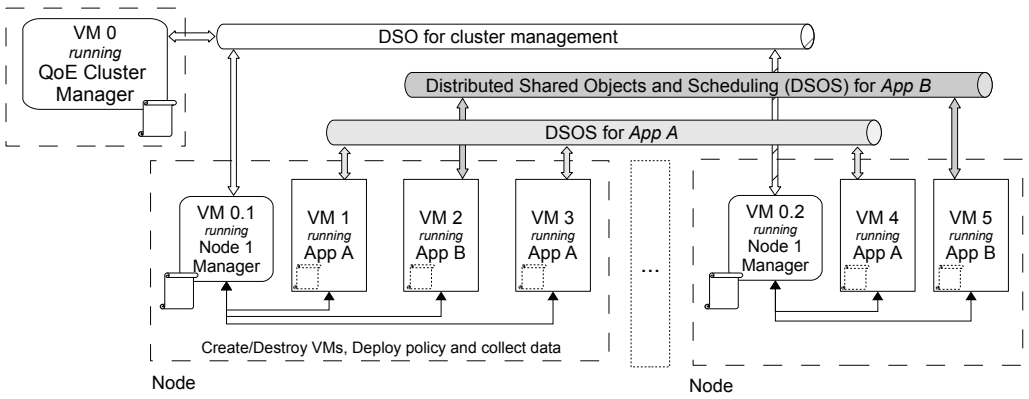


Figure 3.1: Overall architecture

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. 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.

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

In summary, the responsibilities of each of these entities are the following:

- 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) VM
  - 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 is 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. 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. 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*, 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 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.

3.2 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 (VM) for managed languages can report about several aspects of their internal components, like used memory, number of threads, classes loaded [Oracle, , Microsoft,]. 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 existent 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.

The counterpart of resource containment is when an application needs more resources but the node where it is running is exhausted, or out of resources. 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 bioinformatics [Gront and Kolinski, 2008, López-Arévalo et al., 2007], with many available Java-based APIs (e.g., Neobio).

3.3 Checkpointing and migration of the execution state

3.4 Cluster-wide execution space

Our mechanism to distribute threads among the cluster is built by leveraging and extending the Terracotta [Bonr and Kuleshov, 2007] 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 are clustered together by injecting cluster-aware bytecode into the application Java code at runtime, as

the classes are loaded by each JVM. Part of the cluster-aware bytecode injected causes each JVM to connect to the Terracotta server instances, which handles the storage and retrieval of object data in the shared clustered virtual heap. To the application, the objects living in the distributed shared space are just like regular objects on the heap of the local JVMs.

However Terracotta handles clustered objects differently from 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. Consistency is assured by using the synchronization present in the Java application (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.

Chapter 4

Resource Management Mechanisms for Managed Runtimes

4.1 Resource accounting and adaptability

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. Checkpointing, restore and migration mechanisms, are used for more coarse-grained load-balancing across the cluster.

Current high level virtual machines (HLL-VM) can report about several aspects of their internal components, like used memory, number of threads, classes loaded [Oracle, , Microsoft,]. 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

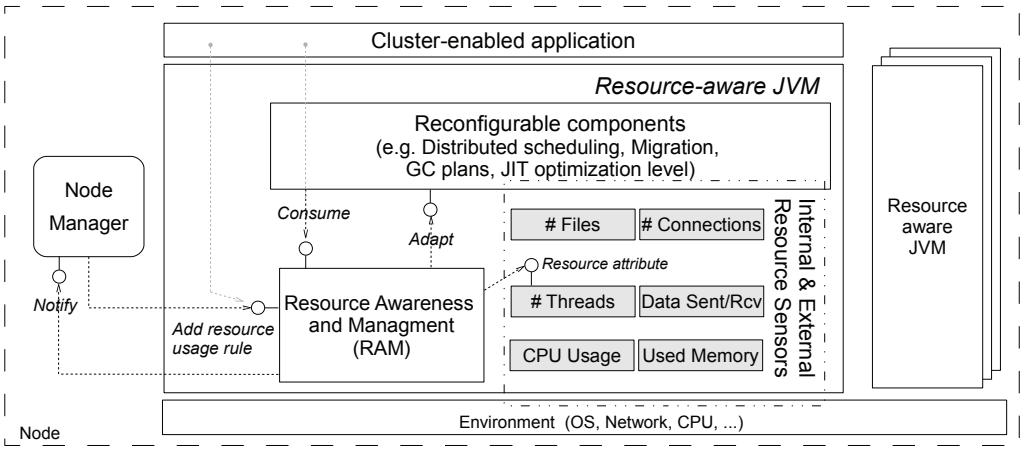


Figure 4.1: Overall architecture

VM must continue to run existent 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.

The management of a given resource implies the capacity to monitor its current state and to be directly or indirectly in control of 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) or be strongly dependent on the underlying architecture and operating system (e.g. CPU usage).

To unify the management of such disparate types of resources, we have started the implementation of JSR 284 - The Resource Management API [et al., 2009] in the context of Jikes.

The JSR 284 elements are *resources*, *consumers* and *resource management policies*. Resources are represented by their attributes (`ResourceAttributes` interface). For example, resources can be classified as *Bounded* or *Unbounded*. *Unbounded* resources have no intrinsic limit 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. A *Consumer* represents an execut-

ing 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*.

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. Calls are made using the mechanism already presented in Section 4.2.

Finally, a new package of classes was integrated in the GNU classpath in order for applications to be able to specify their policies. This 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.

4.2 Concurrent checkpoint and migration

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 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.

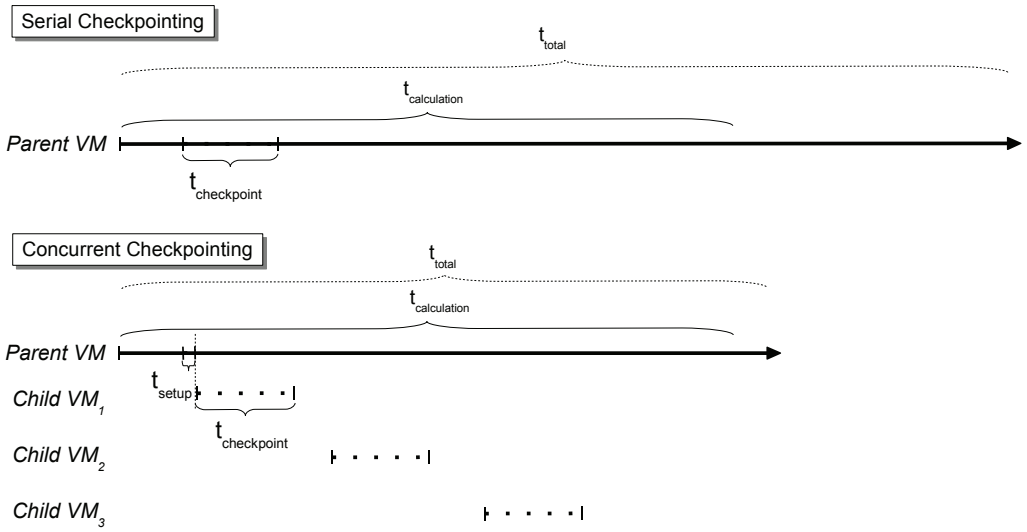


Figure 4.2: Each concurrent checkpoint runs in a *child* VM. $t_{calculation}$ is the free run time, without any checkpoint. t_{total} is the total execution time, considering either serial or concurrent checkpoint.

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 copied on demand. Currently, our implementation in Linux relies on Linux's system call, `fork()`, which has the desired semantics [Tanenbaum, 2007]. 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 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 4.2 illustrates how the concurrent checkpoint progresses, along with the application, in comparison with the serial (non-concurrent) approach. 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) 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.

4.3 Cluster-wide thread scheduling

Our mechanism to distribute threads among the cluster is built by leveraging and extending the Terracotta [Bonr and Kuleshov, 2007] 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 using the synchronization present in the Java application (with monitors), which turns into Terracotta transaction boundaries. Piggybacked on these operations, Terracotta in-

jects 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. To do this we used the ASM framework [Bruneton et al., 2002]. Creation of threads in remote nodes is a result of invoking JSR 284 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, 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.

4.4 Evaluation

Chapter 5

Driving Adaptability with *Quality of Execution*

5.1 *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 the Cobb-Douglas [**?**] production function from Economics to motivate and to help characterize the QoE, as decribed next.

We are partially inspired by the Cobb-Douglas [**?**] production function (henceforth referred as equation) from Economics to motivate and to help characterize the QoE. The Cobb-Douglas equation, presented in Equation 5.1, is used in Economics to represent the *production* of a certain good.

$$P = A \cdot K^{\alpha} \cdot L^{\beta} \tag{5.1}$$

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

invested.

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

Alternatively, labour can be regarded, not as a variable representing a measure of human work employed, but as a result, representing the efficiency of the capital invested, given the production output achieved, i.e., labour as a multiplier of resources into production output. This is usually expressed by representing Equation 5.1 in terms of L, as in Equation 5.2. For simplicity, we have assumed all the three factors to be equal to one. First, the technological and civilization context does not apply, and since the data center economy is simpler, as there is a single kind of activity, computation, and not several, the relative weight of labour and capital is not relevant. Furthermore, we will be more interested in the variations (relative increments) of efficiency than on efficiency values themselves, hence the simplification does not introduce error.

$$L = \frac{P}{K} \tag{5.2}$$

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

to attain a certain progress. While resources can be measured easily by CPU shares and memory allocated, application progress is more difficult to characterize. We give details in Section 4 but we are mostly interested in *relative variations* in application progress (regardless of the way it is measured), as shown in Equation 5.3, and their complementary variations in *production cost per unit*, PCU.

$$\Delta\ L \approx \frac{\Delta\ P}{\Delta\ K}, \quad and \quad thus \quad \Delta\ PCU \approx \frac{\Delta\ K}{\Delta\ P} \tag{5.3}$$

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

Variations in the PCU can be regarded as an opportunity for *yield* regarding a given resource r, and a management strategy or allocation configuration s_x, i.e., a return or reward from applying a given strategy to some managed resource, as presented in Equation 5.4.

$$Yield_r(ts, s_a, s_b) = \frac{Savings_r(s_a, s_b)}{Degradation(s_a, s_b)} \tag{5.4}$$

Because QoE-JVM is continuously monitoring the application progress, it is possible to incrementally measure the yield. Each partial $Yield_r$, obtained in a given time span ts, contributes to the total one obtained. This can be evaluated either over each time slice or globally when applications, batches or workloads complete. For a given execution or evaluation period, the total yield is the result of summing all significant partial yields, as presented in Equation 5.5.

$$TotalYield_r(s_a, s_b) = \sum_{ts=0}^{n} Yield_r(ts, s_a, s_b) \tag{5.5}$$

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 5.6. 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.

$$Savings_r(s_a, s_b) = \frac{U_r(s_a) - U_r(s_b)}{U_r(s_a)} \tag{5.6}$$

Regarding *performance degradation*, it represents the impact of the savings, given a specific performance metric, as presented in Equation 5.7. 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, $P(s_b)$.

$$Degradation(s_a, s_b) = \frac{P(s_b) - P(s_a)}{P(s_a)} \tag{5.7}$$

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

To effectively apply the economic model presented in Section 5.1 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.

5.1.1 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 will encompass all objects when the algorithm terminates. If the rate of objects processed can be determined it will indicate how the application is making progress. Other examples would be applications to perform video encoding, where the number of frames processed is a measure of progress [**?**].

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

- **Number of requests processed**: This metric is typically associated with interactive applications, like Bag-of-tasks environments or Web applications;
- **Completion time**: For short and medium time living applications, where it is not possible to change the source code or no information is available to lead an instrumentation process, this metric will be the more effective one. This metric only

requires the *QoE-JVM* to measure *wall clock time* when starting and ending the application;

- **Code: instrumented or annotated**: If information is available about the application high level structure, instrumentation can be used to dynamically insert probes at load time, so that the *QoE-JVM* can measure progress using a metric that is semantically more relevant to the application;
- **Mutator execution time**. When mutators (i.e. execution flows of applications) have high execution percentages, in proportion to the time spent in garbage collector, this indicate that the application is making more progress than others where garbage collection is using a higher percentage of total execution.
- **I/O: storage and network**: For application dependent on I/O operations, changes in the quantity of data saved or read from files or in the information sent and received from the network, can contribute to determine if the application reached a bottleneck or is making progress;
- **Memory page activity**: Allocation of new memory pages is a low level indicator (collected from the OS or the VMM) that the application is making effective progress. A similar indication will be given when the application is writing in new or previous memory pages.

Although *QoE-JVM* can measure low level indicators as I/O storage and network activity or memory page activity, Section **??** uses the metric **completion time** to measure performance degradation, as defined in Section 5.1. This is so because the applications used to demonstrate the benefits of our system are benchmarks that are representative of different types of workloads and have a short execution time. However, the metric **mutator execution time** also has an important role because the strategies to manage the heap, described in Section **??**, take into account the dual of this metric - the ratio of time spent in garbage collection.

5.1.2 Resource types and usage

In the model presented at Section 5.1, $Savings_r$ refers to any computational resource (r) which applications consume to make progress. Resources can be classified as either

	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 5.1: Implicit resources and their throttling properties

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.

Both types of resource are relevant to be monitored and regulated. *Explicit* and *implicit* resources might be constrained as a protection mechanism against ill behaved or misusing application [Geoffray et al., 2009]. 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 5.1 presents *implicit* resources and the throttling properties associated to each one. These proprieties can be either counted values (e.g. x number of cores) or rates (e.g. y KiBytes/seconds). To regulate CPU and memory both types of properties are

applicable. For example, CPU can be throttled either by controlling the number of cores or the *cap* (i.e. the maximum percentage of CPU a VM is able to use, even if there is available CPU time). Memory usage can be regulated either through a fixed limit or by using a factor to shrink or grow this limit. Although the heap size cannot be smaller than the working set, the size of the over committed memory influences the application progress. A similar rationale can be made about resource pools, 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).

Chapter 6

Conclusions

References

[xen, 2012] (2012). http://wiki.xensource.com/xenwiki/creditscheduler, visited at 31-03-2012.

[kaf, 2012] (2012). Kaffe virtual machine, http://www.kaffe.org/, visited 29-05-2012.

[Alpern et al., 2005] Alpern, B., Augart, S., Blackburn, S. M., Butrico, M., Cocchi, A., Cheng, P., Dolby, J., Fink, S., Grove, D., Hind, M., McKinley, K. S., Mergen, M., Moss, J. E. B., Ngo, T., and Sarkar, V. (2005). The jikes research virtual machine project: building an open-source research community. *IBM Syst. J.*, 44:399–417.

[Amdahl et al., 1964] Amdahl, G. M., Blaauw, G. A., and Brooks, F. P. (1964). Architecture of the ibm system/360. *IBM J. Res. Dev.*, 8:87–101.

[Aridor et al., 1999] Aridor, Y., Factor, M., and Teperman, A. (1999). cjvm: a single system image of a jvm on a cluster. In *In Proceedings of the International Conference on Parallel Processing*, pages 4–11.

[Arnold et al., 2004] Arnold, M., Fink, S., Grove, D., Hind, M., and Sweeney, P. F. (2004). Architecture and Policy for Adaptive Optimization in Virtual Machines. Technical Report 23429, IBM Research.

[Arnold et al., 2005] Arnold, M., Fink, S. J., Grove, D., Hind, M., and Sweeney, P. F. (2005). A survey of adaptive optimization in virtual machines. In *Proceedings of the IEEE, 93(2), 2005. Special Issue on Program Generation, Optimization, ans Adaptation.*

[Back and Hsieh, 2005] Back, G. and Hsieh, W. C. (2005). The kaffeos java runtime system. *ACM Trans. Program. Lang. Syst.*, 27:583–630.

[Baker, 1994] Baker, H. G. (1994). Thermodynamics and garbage collection. *SIGPLAN Not.*, 29:58–63.

[Barham et al., 2003] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. (2003). Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37:164–177.

[Binder et al., 2009] Binder, W., Hulaas, J., Moret, P., and Villazón, A. (2009). Platform-independent profiling in a virtual execution environment. *Softw. Pract. Exper.*, 39:47–79.

[Blackburn et al., 2006] Blackburn, S. M., Garner, R., Hoffman, C., Khan, A. M., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J. E. B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., and Wiedermann, B. (2006). 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. ACM Press.

[Blake and Rodrigues, 2003] Blake, C. and Rodrigues, R. (2003). High availability, scalable storage, dynamic peer networks: Pick two. In Jones, M. B., editor, *HotOS*, pages 1–6. USENIX.

[Bonr and Kuleshov, 2007] Bonr, J. and Kuleshov, E. (2007). 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.

[Brewer, 2010] Brewer, E. A. (2010). A certain freedom: thoughts on the cap theorem. In Richa, A. W. and Guerraoui, R., editors, *PODC*, page 335. ACM.

[Bruneton et al., 2002] Bruneton, E., Lenglet, R., and Coupaye, T. (2002). Asm: A code manipulation tool to implement adaptable systems. In *In Adaptable and extensible component systems*.

[Cherkasova et al., 2007] Cherkasova, L., Gupta, D., and Vahdat, A. (2007). Comparison of the three cpu schedulers in xen. *SIGMETRICS Perform. Eval. Rev.*, 35:42–51.

[Click et al., 2005] Click, C., Tene, G., and Wolf, M. (2005). The pauseless gc algorithm. In *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, VEE '05, pages 46–56, New York, NY, USA. ACM.

[Czajkowski et al., 2005] Czajkowski, G., Hahn, S., Skinner, G., Soper, P., and Bryce, C. (2005). A resource management interface for the java platform. *Softw. Pract. Exper.*, 35:123–157.

[Czajkowski and von Eicken, 1998] Czajkowski, G. and von Eicken, T. (1998). 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. ACM.

[Deutsch and Schiffman, 1984] Deutsch, L. P. and Schiffman, A. M. (1984). 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. ACM.

[Duran-Limon et al., 2011] Duran-Limon, H., Siller, M., Blair, G., Lopez, A., and Lombera-Landa, J. (2011). Using lightweight virtual machines to achieve resource adaptation in middleware. *IET Software*, 5(2):229–237.

[et al., 2009] et al., G. C. (2009). Java specification request 284 - resource consumption management api.

[Geoffray et al., 2009] Geoffray, N., Thomas, G., Muller, G., Parrend, P., Frenot, S., and Folliot, B. (2009). I-JVM: a Java Virtual Machine for component isolation in OSGi. In *IEEE/IFIP International Conference on Dependable Systems & Networks*.

[Gong et al., 2010] Gong, Z., Gu, X., and Wilkes, J. (2010). Press: Predictive elastic resource scaling for cloud systems. In *Network and Service Management (CNSM), 2010 International Conference on*, pages 9 –16.

[Govil et al., 1999] Govil, K., Teodosiu, D., Huang, Y., and Rosenblum, M. (1999). Cellular disco: resource management using virtual clusters on shared-memory multiprocessors. *SIGOPS Oper. Syst. Rev.*, 33:154–169.

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

[Grzegorczyk et al., 2007] Grzegorczyk, C., Soman, S., Krintz, C., and Wolski, R. (2007). 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. IEEE Computer Society.

[Guan et al., 2009] Guan, X., Srisa-an, W., and Jia, C. (2009). 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. ACM.

[Gulati et al., 2007] Gulati, A., Merchant, A., Uysal, M., and Varman, P. J. (2007). Efficient and adaptive proportional share i/o scheduling. Technical report, HP Laboratories Palo Alto.

[Hertz et al., 2009] Hertz, M., Bard, J., Kane, S., Keudel, E., Bai, T., Kelsey, K., and Ding, C. (2009). Waste not,want not: resource-based garbage collection in a shared environment. Technical Report TR-2006-908, University of Rochester.

[Hertz et al., 2005] Hertz, M., Feng, Y., and Berger, E. D. (2005). Garbage collection without paging. *SIGPLAN Not.*, 40:143–153.

[Hertz et al., 2011] Hertz, M., Kane, S., Keudel, E., Bai, T., Ding, C., Gu, X., and Bard, J. E. (2011). 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. ACM.

[Hines et al., 2011] Hines, M., Gordon, A., Silva, M., Silva, D. D., Ryu, K. D., and Ben-Yehuda, M. (2011). Applications know best: Performance-driven memory overcommit with ginkgo. In *CloudCom '11: 3rd IEEE International Conference on Cloud Computing Technology and Science*.

[Holland et al., 2008] Holland, R. C. G., Down, T. A., Pocock, M. R., Prlic, A., Huen, D., James, K., Foisy, S., Dräger, A., Yates, A., Heuer, M., and Schreiber, M. J. (2008). Biojava: an open-source framework for bioinformatics. *Bioinformatics*, 24(18):2096–2097.

[Hulaas and Binder, 2008] Hulaas, J. and Binder, W. (2008). Program transformations for light-weight cpu accounting and control in the java virtual machine. *Higher Order Symbol. Comput.*, 21:119–146.

[Iyer et al., 2009] Iyer, R., Illikkal, R., Tickoo, O., Zhao, L., Apparao, P., and Newell, D. (2009). Vm3: Measuring, modeling and managing vm shared resources. *Computer Networks*, 53(17):2873–2887.

[López-Arévalo et al., 2007] López-Arévalo, I., Bañares-Alcántara, R., Aldea, A., and Rodríguez-Martínez, A. (2007). A hierarchical approach for the redesign of chemical processes. *Knowl. Inf. Syst.*, 12(2):169–201.

[Maggio et al., 2012] Maggio, M., Hoffmann, H., Papadopoulos, A. V., Panerati, J., Santambrogio, M. D., Agarwal, A., and Leva, A. (2012). Comparison of decision-making

strategies for self-optimization in autonomic computing systems. *ACM Trans. Auton. Adapt. Syst.*, 7(4):36:1–36:32.

[Manson et al., 2005] Manson, J., Pugh, W., and Adve, S. V. (2005). The java memory model. *SIGPLAN Not.*, 40:378–391.

[Mao et al., 2009] Mao, F., Zhang, E. Z., and Shen, X. (2009). 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. ACM.

[Michael Hines et al., 2011] Michael Hines, A. G., Silva, M., Silva, D. D., Ryu, K. D., and Ben-Yehuda, M. (2011). Applications know best: Performance-driven memory overcommit with ginkgo. In *CloudCom '11: 3rd IEEE International Conference on Cloud Computing Technology and Science*.

[Microsoft,] Microsoft. Clr profiler for the .net framework 2.0.

[Nagpurkar et al., 2006] Nagpurkar, P., Krintz, C., Hind, M., Sweeney, P. F., and Rajan, V. T. (2006). Online phase detection algorithms. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 111–123, Washington, DC, USA. IEEE Computer Society.

[Oracle,] Oracle. Java virtual machine tool interface (JVMTI), http://download.oracle.com/javase/6/docs/technotes/guides/jvmti/.

[Padala et al., 2009] Padala, P., Hou, K.-Y., Shin, K. G., Zhu, X., Uysal, M., Wang, Z., Singhal, S., and Merchant, A. (2009). 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. ACM.

[Salehie and Tahvildari, 2009] Salehie, M. and Tahvildari, L. (2009). Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4:14:1–14:42.

[Shao et al., 2009] Shao, Z., Jin, H., and Li, Y. (2009). Virtual machine resource management for high performance computing applications. *Parallel and Distributed Processing with Applications, International Symposium on*, 0:137–144.

[Simão et al., 2010] Simão, J., Ribeiro, C., Ferreira, P., and Veiga, L. (2010). Jano: specification and enforcement of location privacy in mobile and pervasive environments. In

Proceedings of the 2nd International Workshop on Middleware for Pervasive Mobile and Embedded Computing, M-MPAC '10, pages 2:1–2:8.

[Simão et al., 2011] Simão, J., de Lemos, J. N. P. A. S., and Veiga, L. (2011). A2-VM: A Cooperative Java VM with Support for Resource-Awareness and Cluster-Wide Thread Scheduling. In *19th International Conference on Cooperative Information Systems (CoopIS 2011)*. LNCS, Springer.

[Simão et al., 2012] Simão, J., Garrochinho, T., and Veiga, L. (2012). A Checkpointing-enabled and Resource-Aware Java VM for Efficient and Robust e-Science Applications in Grid Environments. *Concurrency and Computation: Practice and Experience*.

[Simão and Veiga, 2012a] Simão, J. and Veiga, L. (2012a). Towards an Adaptive and Resource-Aware Java Runtime for Cloud Computing with Quality-of-Execution. ASPLOS/VEE 2012 Poster Session.

[Simão and Veiga, 2012b] Simão, J. and Veiga, L. (2012b). VM Economics for Java Cloud Computing: An Adaptive and Resource-aware Java Runtime with Quality-of-Execution. In *12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*.

[Singer et al., 2007] Singer, J., Brown, G., Watson, I., and Cavazos, J. (2007). Intelligent selection of application-specific garbage collectors. In *Proceedings of the 6th international symposium on Memory management*, ISMM '07, pages 91–102, New York, NY, USA. ACM.

[Singer et al., 2010] Singer, J., Jones, R. E., Brown, G., and Luján, M. (2010). The economics of garbage collection. *SIGPLAN Not.*, 45:103–112.

[Singer et al., 2011] Singer, J., Kovoor, G., Brown, G., and Luján, M. (2011). 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. ACM.

[Smith and Nair, 2005] Smith, J. and Nair, R. (2005). *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann.

[Soman and Krintz, 2007] Soman, S. and Krintz, C. (2007). Application-specific garbage collection. *J. Syst. Softw.*, 80:1037–1056.

[Soman et al., 2004] Soman, S., Krintz, C., and Bacon, D. F. (2004). 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. ACM.

[Stoica et al., 1996] Stoica, I., Abdel-Wahab, H., and Jeffay, K. (1996). On the duality between resource reservation and proportional share resource allocation. Technical report, Norfolk, VA, USA.

[Tanenbaum, 2007] Tanenbaum, A. S. (2007). *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition.

[VMware,] VMware. Vmware vspher 4: The cpu scheduler in vmware esx 4.

[Waldspurger, 1995] Waldspurger, C. A. (1995). *Lottery and stride scheduling: flexible proportional-share resource management*. PhD thesis. AAI0576752.

[Waldspurger, 2002] Waldspurger, C. A. (2002). Memory resource management in vmware esx server. *SIGOPS Oper. Syst. Rev.*, 36:181–194.

[Weiming and Zhenlin, 2009] Weiming, Z. and Zhenlin, W. (2009). 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.

[Wilson, 1992] Wilson, P. R. (1992). Uniprocessor garbage collection techniques. In *Proceedings of the International Workshop on Memory Management*, IWMM '92, pages 1–42, London, UK. Springer-Verlag.

[Zhang et al., 2008] Zhang, H., Lee, J., and Guha, R. (2008). Vcluster: a thread-based java middleware for smp and heterogeneous clusters with thread migration support. *Softw. Pract. Exper.*, 38:1049–1071.

[Zhang et al., 2005] Zhang, Y., Bestavros, A., Guirguis, M., Matta, I., and West, R. (2005). 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. ACM.

[Zhu et al., 2002] Zhu, W., Wang, C.-L., and Lau, F. C. M. (2002). Jessica2: A distributed java virtual machine with transparent thread migration support. *Cluster Computing, IEEE International Conference on*, 0:381.