UNIVERSIDADE DE LISBOA

INSTITUTO SUPERIOR TÉCNICO

Economics-inspired Adaptive Resource Allocation and Scheduling for Cloud Environments

José Manuel de Campos Lages Garcia Simão

Supervisor: Doctor Luís Manuel Antunes Veiga

Thesis approved in public session to obtain the PhD Degree in
Information Systems and Computer Engineering

Jury final classification: Pass with Distinction

Jury:

Chairperson: Chairman of the IST Scientific Board

Members of the Committee:

Doctor Gaël Thomas
Doctor Bruno Miguel Brás Cabral
Doctor David Manuel Martins de Matos
Doctor Luís Manuel Antunes Veiga

2015

UNIVERSIDADE DE LISBOA

INSTITUTO SUPERIOR TÉCNICO

Economics-inspired Adaptive Resource Allocation and Scheduling for Cloud Environments

José Manuel de Campos Lages Garcia Simão

Supervisor: Doctor Luís Manuel Antunes Veiga

Thesis approved in public session to obtain the PhD Degree in
Information Systems and Computer Engineering

Jury final classification: Pass with Distinction

Jury:

Chairperson: Chairman of the IST Scientific Board

Members of the Committee:

Doctor Gaël Thomas
Professor
Télécom SudParis, France

Doctor Bruno Miguel Brás Cabral
Professor Auxiliar
Faculdade de Ciências e Tecnologia da Universidade de Coimbra

Doctor David Manuel Martins de Matos
Professor Auxiliar
Instituto Superior Técnico da Universidade de Lisboa

Doctor Luís Manuel Antunes Veiga
Professor Auxiliar
Instituto Superior Técnico da Universidade de Lisboa

Funding Institutions:
Fundação para a Ciência e Tecnologia
INESC-ID Lisboa
Instituto Superior de Engenharia de Lisboa
Instituto Politécnico de Lisboa

2015

Para a Margarida e para o Miguel

Publications

The work and results presented in this thesis were partially supported by the Portuguese Science and Technology Foundation (Fundação para a Ciência e Tecnologia), projects "Synergy", "RepComp" and "Prosopon", and by a PROTEC grant from the Polytecnic Institute of Lisbon (Instituto Politécnico de Lisboa, Ministério da Ciência e do Ensino Superior). They are partially described in the following peer-reviewed scientific publications:

- International Journals

  1. José Simão and Luís Veiga, *Partial Utility-driven Scheduling for Flexible SLA and Pricing Arbitration in Clouds*. IEEE Transactions on Cloud Computing, *online first*.

  2. José Simão and Luís Veiga, *Adaptability Driven by Quality Of Execution in High-Level Virtual Machines for Shared Environments*. International Journal of Computer Systems Science and Engineering, 28(6), pp. 59-72, November 2013, CRL Publishing. *Q2 in SCImago*

  3. José Simão and Tiago Garrochinho and Luís Veiga, *A Checkpointing-enabled and Resource-Aware Java VM for Efficient and Robust e-Science Applications in Grid Environments*, Concurrency and Computation: Practice and Experience, 24(13), pp. 1421-1442, September 2012, Wiley. *Q2 in SCImago*.

- International Conferences and Workshops

  1. José Simão and Luís Veiga, *Flexible SLAs in the Cloud with Partial Utility-driven Scheduling Architecture*, IEEE 5th International Conference on Cloud Computing Technology and Science (CloudCom 2013), December 2013, IEEE, *Acceptance ratio* $\approx 17.8\%$, **(Best-Paper Runner-up)**.

  2. José Simão and Navaneeth Rameshan and Luís Veiga, *Resource-Aware Scaling of Multi-threaded Java Applications in Multi-tenancy Scenarios*, IEEE CloudCom 2013, December 2013, IEEE *[short paper]*.

3. José Simão and Jeremy Singer and Luís Veiga, *A Comparative Look at Adaptive Memory Management in Virtual Machines*, IEEE CloudCom 2013, December 2013, IEEE *[short paper]*.

4. João Marques Silva and José Simão and Luís Veiga, *Ditto - Deterministic Execution Replayability-as-a-Service for Java VM on Multiprocessors*, ACM/IFIP/Usenix International Middleware Conference (Middleware 2013), December 2013, Springer. *Core A, Acceptance ratio* $\approx 18.7\%$, *RADIST A*.

5. José Simão and Luís Veiga, *A Progress and Profile-driven Cloud-VM for Improved Resource-Efficiency and Fairness in e-Science Environments*, 28th ACM Symposium On Applied Computing (SAC 2013), March 2013, ACM.

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

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

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

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

- Posters and Talks

1. José Simão and Axel Domingues and Luís Veiga, *Flexible SLAs in the Cloud With Partial-Utility Scheduling*, Poster Session of EuroSys 2013, April 2013.

2. José Simão and Luís Veiga, Invited talk in the Middleware 2012 Doctoral Symposium, December 2012.

3. José Simão and Luís Veiga, *Towards an Adaptive and Resource-Aware Java Runtime for Cloud Computing with Quality-of-Execution*, Poster session of the 17th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), March 2012.

During the PhD program I have also published further extensions to the work presented in my master dissertation, describing how a location service can disclose information conditioned by the enforcement of history-based and discretionary security policies. The policies are described in a extended version of the Security Policy Language [Ribeiro et al., 1999].

- International Journals
  - José Simão and Carlos Nuno da Cruz Ribeiro and Paulo Ferreira and Luís Veiga, *Jano: Location-Privacy Enforcement in Mobile and Pervasive Environments through Declarative Policies*, Journal of Internet Services and Applications (JISA), 3(3), pp. 291-310, December 2012, Springer.
- International Workshop
  - José Simão and Paulo Ferreira and Carlos Nuno da Cruz Ribeiro and Luís Veiga, *Jano - Specification and Enforcement of Location Privacy in Mobile and Pervasive Environments*, Workshop on Middleware for Pervasive Mobile and Embedded Computing (M-MPAC 2010), in Middleware 2010, December 2010, ACM.

Abstract

Cloud infrastructures host different kinds of applications belonging to clients with different levels of service agreements. Their execution is supported by high language virtual machines and system-level virtual machines (VMs). Aiming to maximize revenue, by minimizing the *operational expenditure*, cloud providers often consolidate several VMs in a single server. This is particularly useful also for the emergent distributed clouds where physical resources, at each node, are not so abundant. However, this technique can lead to overcommitment of resources, and to undesirable performance degradation, if carried out in a non-informed way.

This thesis proposes new allocation mechanisms for the VMs used by Platform-as-a-Service (PaaS) and Infrastructure-as-a-Service (IaaS) clouds, and how these mechanisms are to be controlled. They are driven by classic economic notions such as *yield*, which expresses the return the provider has in applying a given resource allocation to the tenants' workload, and *utility functions*, a relation of clients' perceived usefulness to a given allocation.

For PaaS providers, a Java VM was extended with an integrated resource management API, heap resizing policies for yield maximization and concurrent checkpoint for migration of the execution state. Overall, these new mechanisms impose small penalties, measured in the execution of typical benchmarks, while allowing the use of application-tailored policies. At the IaaS level, we present a novel cost model and new scheduling algorithms for system-level VMs, along with their implementation in a state-of-the-art cloud simulation framework. Simulations with synthetic and real-world traces, show that the utility-based scheduling allows more VMs to be allocated, thus allowing extra revenue per resource allocated, and shorter waiting times for clients, when comparing with a utility-oblivious redistribution of resources.

Keywords

Cloud computing, Platform-as-a-Service, Infrastructure-as-a-Service, Virtual Machines, Multi-tenancy, Resources Scheduling, Garbage Collection, Cost Model, Economic Yield, Utility Functions

Resumo

As plataformas e infra-estruturas de computação em nuvem executam diferentes tipos de aplicações, as quais pertencem a clientes com contractos de serviço distintos. A execução destas aplicações é suportada por máquinas virtuais (MVs). Com o objectivo de minimizar despesas operacionais, os fornecedores de serviços recorrem a técnicas de consolidação, juntando várias MVs num único servidor. Esta técnica é ainda mais útil em cenários emergentes como é o caso das nuvens comunitárias. Contudo, tem também o potencial de afectar o desempenho das aplicações, se realizado de uma forma não informada.

Esta tese propõe novos mecanismos de atribuição de recursos para MVs usadas nos modelos de Plataforma como um Serviço (PaaS) e Infra-estrutura como um Serviço (IaaS). A alocação é inspirada em noções clássicas de Economia, como o retorno do investimento, que relaciona o proveito do fornecedor de serviço em usar determinada alocação de recursos, e funções de utilidade, que determinam como o cliente valoriza possíveis alocações apenas parciais.

Para os fornecedores no mercado de Plataforma como um serviço (PaaS), foi estendida uma MV Java para incorporar uma interface uniformizada para a gestão de recursos, políticas para gestão de dimensão dinâmica, e um mecanismo para salvaguarda concorrente do estado de execução das aplicações. No seu conjunto, a implementação destes mecanismos revelou ter um impacto pequeno no tempo de execução de aplicações de referência, dando a possibilidade de usar políticas de gestão de recursos específicas para cada aplicação. Para os fornecedores no mercado de Infra-estrutura como um Serviço (IaaS), é apresentado um novo modelo de custos e de alocação de MVs, bem como a sua implementação numa infra-estrutura para simulação de computação em nuvem. Quando comparado com uma redistribuição de recursos que não tem em conta a utilidade parcial, os resultados de simulações mostram que a estratégia proposta permite que mais MVs sejam alocadas, proporcionando ao fornecedor de serviço maior rentabilidade, e menor tempo de espera para os clientes.

Palavras-chave

Computação em nuvem, Plataforma como um Serviço, Infraestrutura como um Serviço, Máquinas Virtuais, Multi inquilino, Modelo de Custos, Retorno Económico, Funções de Utilidade, Alocação de Recursos, Recolha de Objectos não Alcançáveis

Agradecimentos

Quando em 2009 conheci o Prof. Luís Veiga e discuti com ele a hipótese de trabalhar na área da distribuição de recursos e máquinas virtuais, depressa me apercebi do entuasiasmo e energia que coloca em cada conversa. A sua vontade de fazer progresso manteve-se ao longo dos trabalhos de doutoramento. Havia sempre disponibilidade para uma opinião de como ir mais longe. A sua personalidade, conhecimento científico e estilo de orientação, foram essenciais para os sucessos que conseguimos ao longo deste percurso. A ele o meu muito obrigado.

Chegado aqui, ao olhar para trás vejo mais de duas décadas de estudo, prática e ensino, que se enquadram no que é habitualmente designado como engenharia informática e de computadores. Foram várias as pessoas que ao longo da minha vida académica me mostraram, através do exemplo, as melhoras práticas em cada uma destas actividades. Estou particularmente agradecido a diversos membros do Centro de Cálculo do ISEL os quais, primeiro como meus professores, e depois como meus colegas, contribuiram para a minha formação, e por isso, para o trabalho que agora apresento.

Agradeço ao IPL e ao ISEL, em particular à ADEETC e aos seus docentes, as condições criadas para eu conseguir usufruir da dispensa parcial de serviço docente, no contexto da bolsa PROTEC.

Faço também um balanço muito positivo destes últimos quatro anos de trabalho no INESC-ID Lisboa. Agradeço aos alunos cujos trabalhos tiveram um papel relevante na elaboração de artigos em que fui co-autor. Destes destaco o Tiago Garrochinho e o João Lemos. Agradeço também à direcção do INESC-ID pelas condições que directa e indirectamente me proporcionaram.

Agradeço aos meus pais, Rosália e José, por acreditarem em mim e no meu sucesso, por se terem entusiasmado com as minhas conquistas e serem solidários nas minhas frustações.

Agradeço à Carmen a compreensão pelo tempo que não estive com ela e com os nossos filhotes, a Margarida e o Miguel. Aos três, agradeço o carinho que sempre me deram.

Lisboa, 16 de Outubro de 2014

Contents

List of Figures

List of Tables

Part I

Thesis Motivation and Artifacts

1 Introduction

Contents

1.1 Computing in the Cloud

In today's scenarios of large scale computing and service providing, the deployment of software workloads in distributed infrastructures, namely computer clusters, is a very active research area. In recent years, the use of Grids, Utility and Cloud Computing

shows that these are approaches with growing interest and applicability, as well as scientific [Hiden et al., 2013; Ishakian et al., 2012; Beloglazov and Buyya, 2012; Silva et al., 2011; Buyya et al., 2009] and commercial impact.[1]

After some efforts of the research community to reach a consensual explanation of what is Cloud Computing, from which a minimum set of features would be scalability, pay-per-use utility model, and virtualization [Vaquero et al., 2008], institutional organizations have recently promoted a common ground definition. A report from the European Commission [European Commission, 2012] describes, in simplified terms, that cloud computing is the storing, processing, and use of data on remotely located computers accessed over the internet. Although the report recognizes the wide range of defining features, it focused on some key features such as: i) the dynamically and optimized use of hardware across a network of computers, ii) the ability for the user to pay by usage and, iii) the easiness to change the hardware requirements. The European Commission report also mentions that a cloud set-up consists of several layers: hardware, middleware or platform, and application software.

A similar previous definition, which also emphasizes the remote nature of cloud computing, is made in a report from the U.S. National Institute of Standards and Technology (NIST) [Peter Mell and Tim Grance, 2011], where cloud computing is presented as a computing model for enabling ubiquitous, convenient, on-demand networked access to a shared pool of configurable computing resources. Its main requirements are: i) on-demand self-service, ii) resource pooling, iii) rapid elasticity, iv) Broad network access, and v) Measured service.

Both definitions agree on the same central issue: cloud computing is computation capacity offered as a commodity. This notion is captured in what is today commonly known as Utility Computing [Armbrust et al., 2009], a vision initially presented by professor Jonh MacCarthy in 1961 [Garfinkel, 1999].

1.1.1 Service Models

Widely adopted cloud service models are known as: i) Infrastructure-as-a-Service (IaaS), ii) Platform-as-a-Service (PaaS), and iii) Software-as-a-Service (SaaS). In IaaS, assets are

[1] Cloud Computing Will Become the Bulk of New IT Spend by 2016, `http://www.gartner.com/newsroom/id/2613015`, visited July 3, 2014

generic processing and storage resources where clients can remotely run their own software stack using system-wide virtual Machines (VMs) [Smith and Nair, 2005]. In PaaS, the asset is a remote execution platform or framework supported by the provider, freeing the client from the configuration of the underlying network, servers and storage systems. Finally, the SaaS asset is a complete application running on the provider's hardware, accessed using different types of users-agents, typically browsers.

All these service models are usually multi-tenant, which means that the same underlying infrastructure will serve different clients (or tenants), each using the service in different ways. They offer a computational asset for a given price, while promising to free the client from the hardships of assembling and maintaining the necessary hardware resources. While Software-as-a-Service may have commercial interest to almost any Internet user, the other two service models are usually appealing to startups, small and medium enterprises, and research labs and universities.

NIST's definition also covers several deployment models [Peter Mell and Tim Grance, 2011], namely, Private, Community, Public, and Hybrid Clouds. Private deployments are for exclusive use of a single organization (e.g. Companies). Community clouds are shared among organizations (more or less formal) with similar interests, and may be owned by one or more members of the community. Public deployments exist on the premise of a cloud provider which can be a business, a government department or a university. Finally, hybrid clouds are a combination of the former types with support for assets portability using open source or proprietary technologies.

The IaaS service model rents system virtual machines, lower-level storage and network capabilities. The manager of virtual machines, known as virtual machine monitor or hypervisor [Smith and Nair, 2005], enables the hosting of complete execution stacks, from the operating system to the application server. Amazon EC2[2] is a popular provider for this kind of service. Other major technology players, such as Microsoft, also have solutions for IaaS integrated in their cloud offer.[3] Other providers such as RacketSpace[4] or Apache CloudStack[5] support their business model in open-source solutions, e.g. OpenStack[6], promising to make an easy migration between a private and their public cloud.

Current PaaS providers offer a service model that is mostly tailored to the develop-

[2] `http://aws.amazon.com/ec2/`, visited July 2, 2014
[3] `http://azure.microsoft.com/en-us/services/virtual-machines/`, visited July 2, 2014
[4] `http://www.rackspace.com/`, visited July 2, 2014
[5] `http://cloudstack.apache.org/`, visited October 14, 2014
[6] `http://www.openstack.org/`, visited July 2 2014

ment of web applications, usually supported by the use of different managed runtimes. Google App Engine and Microsoft Azure are two representative examples of PaaS [Armbrust et al., 2009]. Google App Engine is a multi-language development environment with several language extensions to support web application scaling and to enforce their service-level restrictions. Microsoft Azure offers a similar environment but is focused on the .NET platform. Heroku [Heroku, 2014] is a more general purpose platform where applications do not have to bind to vendor-specific programming interfaces. In all these cases, workloads execute inside modern managed runtimes which are known in the literature as high-level language virtual machines (HLL-VM) [Smith and Nair, 2005].

1.1.2 Fundamentals and Innovations in Cloud Technology

From a scientific point of view, Cloud Computing is the natural evolution of research efforts related to computer grids, clusters and data center hosting in general. For example, Grid infrastructures have long dealt with the need to devise market-driven strategies to manage resources, while cluster-related technologies, such as distributed shared memory, are used in many scenarios to simplify crossing the boundary of a single machine (e.g., application scaling [Ferreira et al., 2003] or distributed key-value storage [Dragojević et al., 2014]). Although all these technologies contribute to the success of Cloud Computing, virtualization in general, and virtual machines at the system and language-level, in particular, made a decisive contribution to provide seamless resource pooling, consolidation, on-demand self-service and elasticity. An idea that IBM started in the 1960s was recovered and later extended with hardware support and is now the enabler of many Cloud Computing solutions [Armbrust et al., 2009].

But now that the today's Cloud is well established, and adopted across all the information technology market, how will it evolve? Will current datacenters continue to grow larger and larger, struggling to keep up with energy, cooling, and operational costs, or will the Cloud of the future be made of a mesh of Clouds, where dispersed, heterogeneous computational resources can be used to support either public, private or hybrid cloud solutions? The second option seems to be the most promising one and is currently being embraced by the research and engineering community. For example, the Institute of Electrical and Electronics Engineers (IEEE) is currently assembling, in the

context of the IEEE Cloud Computing Initiative,[7] a testbed – The Intercloud Testbed[8] – to promote the definition of standards for the federation of clouds. Each cloud is supported by different organizations, both companies and universities.

1.2 Thesis Motivation and Challenges

Recently, some researchers have proposed to bring the cloud closer to the clients [Liu et al., 2011; Khan et al., 2013b; Lèbre et al., 2013; Sharifi et al., 2014], distributing the datacenter across a network of nodes. These deployments can be more energy-efficient and bring speed-ups because they reduce network latency.

Although any Internet node can potentially host these new deployments, they will be of most interest to organizations where, although resources may not be abundant, they are used sparsely and with different peaks during the day or periods of longer duration. Examples include universities and research labs where resources already exist and can be organized with advantage into a cloud. For example, Rutgers – The State University of New Jersey – assembled a solar powered datacenter comprised of small containers, solar panels and batteries, hosting two racks of energy-efficient servers.[9] Other public places, such as condominiums and malls, could also have economic advantages by renting resources from their publicly accessed clouds. This is actually what Amazon did with their idle capacity and that started the widespread adoption of cloud computing. Another example is the *data furnace*, proposed by Liu et al. [Liu et al., 2011], which would place small datacenters across condominiums and use the wasted heat produced by the servers for heating residential homes.

Multi-tenancy and resource pooling are essential characteristics of Cloud Computing, which means that the allocation of resources must be done according to each tenant's requests and demand. By distributing the cloud, hosting nodes will have more resource restrictions than the large, energy-hungry and remote datacenters. This makes the problem of resource allocation not just a problem of how to scale (how to dynamically give more or less resources), but to determine which are the tenants that can benefit the most from the new allocation.

Providers must employ dynamic resource allocation strategies to take into account

[7] `http://cloudcomputing.ieee.org/`, visited July 2, 2014
[8] `http://www.intercloudtestbed.org/`, visited July 2, 2014
[9] `http://parasol.cs.rutgers.edu/`, visited July 2, 2014

how effectively each tenant uses their allocated resources, and how their progress will be affected by handing the available resources to higher priority/paying tenants. Such resource allocation considerations are relevant to be taken into account regarding the assets involved both in the PaaS and IaaS service models.

1.2.1 Plataform-as-a-Service

In the PaaS environment, the resource allocation should be tailored in a application-driven way, taking into account the effective progress of the workload. Although currently commercial PaaS are dominated by web hosting infrastructures, there is an increasing interest in using managed languages for physics simulation, economics/statistics, network simulation, chemistry, computational biology and bio-informatics [Hiden et al., 2013; Krampis et al., 2012; Pierre and Stratan, 2012].

We target applications that have little or no user interactions. They are setup to run in the remote cluster with the necessary information at the execution site (which can be in each machine's file system or in a network storage). These applications can also use widely available distributed shared objects middlewares to create the illusion of a distributed shared heap, subject of previous research work [Zhu et al., 2002; Ferreira et al., 2003; Bonér and Kuleshov, 2007] and currently popularized by industry.[10,11,12]

In general, this type of computation is referred to as *e-science*, which constitutes the act of using large scale computational resources to carry out research in biologic, economic, social and other sciences. Available tools for these applications often also belong to a class of computing known as High Performance Computing (HPC). Although some facilities of modern languages, such as just-in-time compilation and automatic garbage collection, have traditionally hindered the adoption of these languages to HPC environments due to their dynamic compilation and pause overheads, they are now more appealing to these communities [Fries, 2012; Chen et al., 2014].

Special-purpose runtimes to take advantage of the growing number of available cores in each node and, by extension, across the cluster, are also being implemented and migrated to managed runtimes. X10 [Charles et al., 2005] is a programming language with constructs for splitting the application's data across a partitioned global address space

[10] `http://hazelcast.org/`, visited July 2, 2014
[11] `http://infinispan.org/`, visited July 2, 2014
[12] `http://terracotta.org/products/bigmemorymax`, visited July 2, 2014

and for explicitly coordinate parallel execution flows. ClusterSs [Tejedor et al., 2012] adopts a different approach, hiding must of these details from the programmer. The *MapReduce* programming pattern has become popular by being able to process large amounts of data in parallel. This pattern has also been adopted for running in a single process, both for unmanaged [Ranger et al., 2007] and managed environments [Singer et al., 2011].

These applications are executed in the context of a managed runtime which handles memory allocation and the access to system resources, being able to adapt its management policies, or be instructed to adapt. For a more coarse-grained management strategy, checkpointing and migration at the application-level are useful to avoid the burdens of migrating the entire system stack.

The work of Silva et al. [Silva et al., 2010a] classifies users in four different types, in order to apply differentiating policies to the work of these users. In academic institutions, for example, the same cluster of computers can be used to run e-science applications by students in different academic levels. Using the same infrastructure will have fewer costs and will be easier to maintain. Nevertheless, the managers of the infrastructure will want to impose a high level policy and give distinguished execution quality to different types of students. The mechanisms to obtain this may range from restraining resource consumption (e.g. CPU usage, physical memory allocated), to the migration of the application to another node.

The execution environment for these compute-intensive applications, with little user interaction, has to comply with different requirements from each tenant. To do so, the execution runtime has to have elasticity, in the sense that resources are made available proportionally to the effective need, based on the progress of each workload. Measuring progress cannot depend only on externally observed events. CPU usage, for example, may not be that relevant because consuming more or less CPU it is not a direct measure of application progress. An application that depends on I/O to make progress in its work will exhibit, over time, lower CPU utilization than a CPU-intensive workload. Therefore, one cannot decide, just by observing CPU usage, which one is making more relative progress [Hoffmann et al., 2010].

1.2.2 Infrastructure-as-a-Service

In the IaaS service model, commercial clouds are still dominated by allocating/buying slots of resources that are seemingly paid-per-use, but as if they were to be used in full over the renting period, and not actually integrating the actual resources used, even if (highly probable) sometimes lower than those allocated. Currently, clients of the IaaS service model rent different kinds of computation capacity, in the form of a virtual machine, but they cannot express what happens if the service cannot completely fulfill their requests. This can be accomplished with an extension to current service level agreements (SLAs), where providers would be able to transfer resources in a way that maximizes the client's utility and the provider's revenue.

Current Cloud SLAs include some form of compensation for customers (i.e., resource renters) with credits when average availability drops below a certain point. However, this credit scheme is too inflexible because consumers lose a non-measurable quantity of performance and are only compensated later (i.e., in the next charging cycle). The clients of classic cloud deployments, but also the ones that rely on these new and more heterogeneous deployments, can have different requirements regarding the execution of their workloads and are willing to trade performance for a lower (or almost free) usage cost. To that end, clients should be able to negotiate with the provider a service level agreement that covers scenarios where a requested VM can only be allocated if some level of depreciation is applied to the VMs running in the datacenter. This would allow the provider to transfer resources between VMs aiming to maximize its profit and minimize the penalty to each workload, in a principled approach that is clear and transparent to the client.

1.2.3 Overall scheduling

In summary, typical IaaS scheduling solutions, which sell general purpose computation capability, need a negotiated and economically sound way to determine how to move resources from a tenant to others. Such a resource exchange can be made in favour of higher priority clients or to the ones that are willing to pay more for the same kind of resources.

Regarding the execution of applications, focusing on the ones running on managed runtimes, a similar problem exists. Although several resources have elasticity, that is, re-

sources can be removed or assigned without breaking the application's execution, memory is the one with more impact. To take advantage of the resources available in the cluster supporting the cloud, a managed runtime for these environments needs a synergy of mechanisms and allocation strategies. These mechanisms should include local adaptations to the consumption of resources (where memory must have an prominent attention), ways to determine progress, along with coarse-grained checkpoint and migration capabilities (e.g., for consolidation, failure recovery).

1.3 Current shortcomings

This thesis proposes new approaches to resource allocation and scheduling at the PaaS and IaaS service level. This is a very active research area with a significant amount of contributions. In this section, we briefly identify some of the limitations of the systems proposed in the literature regarding the managed runtimes in multi-tenant environments and the allocation of system-wide virtual machines.

1.3.1 PaaS

The use of workload-aware policies to distribute resources among different tenants needs specific allocation mechanisms and strategies that can act upon the assets controlled by the managed runtimes, most notably memory. To complement this local management, checkpoint and migration of execution state is a relevant resource management technique that today is mostly done at lower-levels of the system stack, namely by the hypervisor, with all the corresponding overhead of keeping and migrating the execution state of several running applications and kernel services.

Recently, the research community has shown the importance of also incorporating resource-aware mechanisms into managed runtimes, giving them the capabilities to adapt resource usage driven by the the workload characteristics [Singer and Jones, 2011; Salomie et al., 2013]. Two major lines of resource-awareness can be identified: i) avoidind application's bad behavior, ii) performance tunning using adaptive memory structures.

Regarding the first line of research, if the only goal is to restrain resource usage, there are currently lightweight virtualization solutions, such as the Linux containers or the Spoon Virtual Machine, where the main focus is to have an environment similar to a system-wide VM but without a complete separated kernel and emulation of all

the hardware, which is a source of overhead in these VMs.[13,14] These solutions can impose restrictions at several types of resources (e.g., network, memory, cores). However, these solutions operate in an application-oblivious way and are mostly statically ruled. The second line of research is more relevant to our scenario. Nevertheless, current approaches have no notion of *multi-tenancy resource effectiveness* in the sense that when there are scarce resources, there is no attempt to determine where to take resources from applications in such a way that they hurt performance the least.

Finally, regarding mechanisms for checkpointing and migration, they are mainly supported at process-level or at system-level 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 the solution.

1.3.2 IaaS

With the advent of Cloud Computing, resource scheduling in virtualized environments received prominent attention from the research community [Dawoud et al., 2012; Buyya et al., 2011; Hertz et al., 2011; Singer et al., 2010]. But, as we migrate to the small, geo-distributed near-the-client datacenter model, we need an architectural extension to the current relation between cloud users and providers, particularly useful for private and hybrid cloud deployments.

One of the main motivations for this distributed cloud movement is energy efficiency. This problem is also of utmost relevance to classic datacenters. Current solutions use well-known techniques such as consolidation [Beloglazov and Buyya, 2010] and dynamic voltage frequency scaling (DVFS) [Chase et al., 2001; Hagimont et al., 2013]. All these techniques may need to violate to some degree the service level agreement established with the client, but they see that only as a secondary effect, missing the opportunity to establish an economic cost model for the client and the provider to negotiate what happens in such events.

Few works consider that some customers indeed are willing to accept a negotiable performance degradation during the workload execution. This type of flexibility usually requires the adoption of an economic or cost theoretical model. Besides the work in

[13] `https://linuxcontainers.org/`, visited July 3, 2014
[14] `https://spoon.net/docs`, visited August 12, 2014

[Li et al., 2012], Cloudpack [Ishakian et al., 2012] provides support for users to specify workloads in such a way so that they can declare their quantitative resource requirements and temporal flexibilities.

In market-based approaches, the users bid for a VM with a certain amount of resources. To guarantee a steady amount of resources, some systems [Costache et al., 2013] migrate VMs between different nodes. Still, this approach has been found to have the potential to impose a significant performance penalty [Xu et al., 2014].

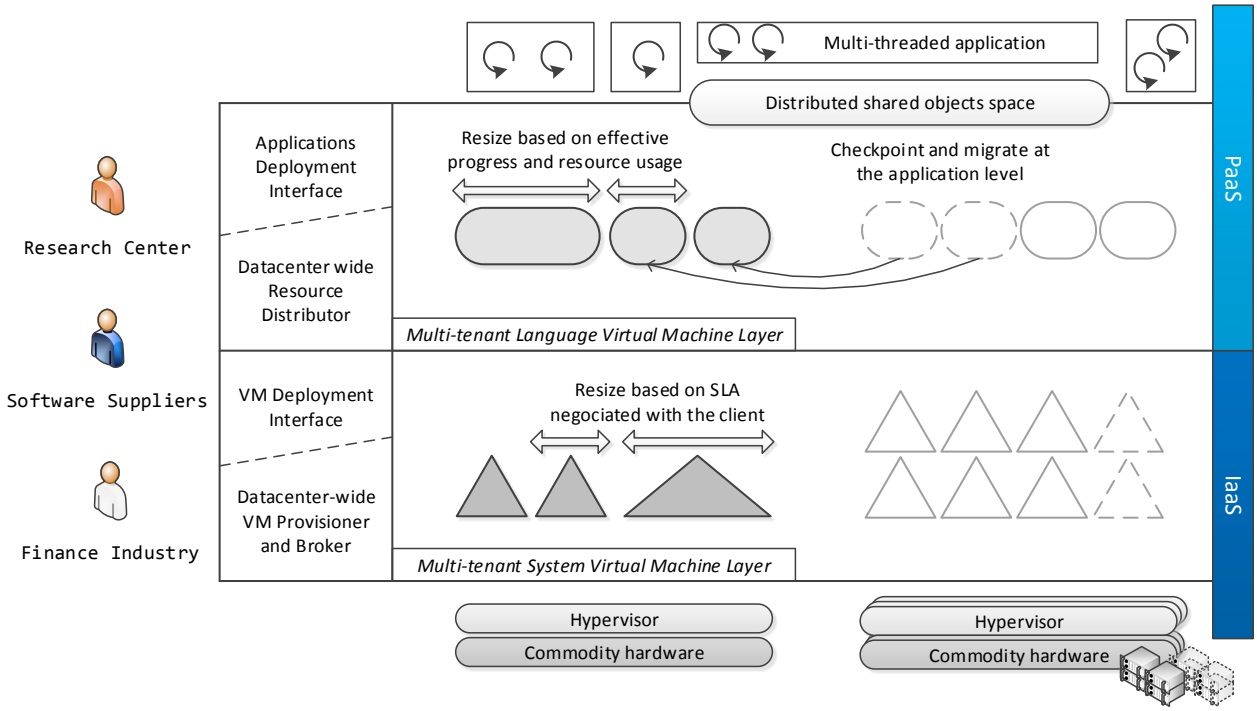


Figure 1.1: Overall system view

1.4 Contributions

Our main contributions are new allocation and scheduling mechanisms driven by strategies that are inspired by classic economic notions such as *utility functions*, a relation of usefulness to a given resource, and *return on investment*, which expresses the advantage the provider has in applying a given resource allocation to the tenant's workload. Figure 1.1 presents the overall view of the systems researched and developed during the

PhD work. The figure focuses on the resource management issues at the PaaS and IaaS levels, targeting scenarios of resource overcommitment.

At the PaaS level we have focused on integrating resource management mechanisms which are currently not available in language runtimes widely used by cloud providers, such as the Java Virtual Machine. These mechanisms enable the use of application-driven policies, which can be used by the cloud provider to redistribute resources more effectively.

At the IaaS level, because resource distribution mechanisms are already commonly available, we have focused on the definition of a new resource distribution model were clients and providers can negotiate how VMs will be provisioned, particularly useful when hosts are overcommited. In the following sections, we briefly describe the most relevant contributions presented in this thesis.

1.4.1 VM's adaptability framework

Researchers have developed a large body of work using different mechanisms and dynamic allocation decisions to tailor how memory, CPU, and I/O management can be adapted to specific workloads. In general, such systems are designed as a *control loop* where sensors are monitored, decisions are made, and actions are performed by actuators. Nevertheless, as is common in systems research, improvement in one property may only be accomplished at the expense of some other property.

The first contribution of this thesis is a framework to classify adaptability in virtual machines [Simão and Veiga, 2012a; Simão et al., 2013]. It describes the adaptation loop of virtual machines discussing their principles, algorithms, mechanisms and techniques. It then proposes a way to qualitative classify each of those according to their responsiveness, i.e., how fast it can react to changes; their comprehensiveness, i.e., the scope of the mechanisms involved; and their intricateness, i.e., the complexity of the modifications to the code base or to the underlying systems. In contrast with other virtual machine surveys [Goldberg, 1974; Arnold et al., 2005; Cherkasova et al., 2007], we present an integrated view of all the steps in the adaptation loop and use a classification approach applied to system-wide and language virtual machines.

1.4.2 Scheduling of PaaS resources

We have designed an Economics-inspired adaptability model and framework to drive the allocation of resources to applications running on high-level language virtual machines, deployed in cloud-like environments, where resources are shared by several tenants [Simão and Veiga, 2012b]. Managed runtimes, executing the workloads of multiple tenants, must adapt to the execution of applications, with different (and sometimes dynamically changing) requirements concerning 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.

The goal of our Economics-inspired model is to incrementally obtain gains in QoE for HLL-VMs running applications requiring more resources (or for more privileged tenants). This, must be accomplished while balancing the relative resource savings drawn from other tenants with the perceived performance degradation. To achieve this goal, we must positively discriminate certain applications, changing their allocation of elastic resources, such as memory. For other applications, resources must be restricted, imposing limits to their consumption, regardless of some performance penalties (that should also be mitigated). In any case, these changes are made transparently to the developer (e.g., by changing internal memory allocation strategies of the runtime) or with minimal cooperation (e.g., through the identification of which operations are the most sensitive to measure progress). Several metrics can be used to infer how applications are making progress, given the resources they are using.

We have developed and evaluated a yield-based model, which has similarities with the economic notion of *return-on-investment*. It determines the benefits that different strategies regarding the heap size (based on the relation between the ratio of live objects and the time spent in GC) and CPU allocation, have to the applications effective progress [Simão and Veiga, 2012, 2013a]. Progress assessment was made based on total execution time but also using workload-aware progress indicators using a simple annotations framework. The framework uses a progress agent that can be attached to any JVM so that the execution of progress-relevant identified methods can be observed with low overhead [Simão and Veiga, 2013c], providing the input variables to our economic model. Results show there are significant benefits for the provider from applying a given policy (e.g. heap size, CPU allocation) in a workload-aware manner, so that resources can be better distributed while performance is not significantly hindered.

JVM-level resource management

As the fundamental building block of our execution environment, we have incorporated, into a Java VM widely-used in research [Alpern et al., 2005; White et al., 2013], the ability to monitor base mechanisms (e.g. garbage collection performance, memory or network consumptions) in order to assess an application's performance and resource usage, and reconfigure these mechanisms at run-time, according to previously defined resource allocation policies (or *quality-of-execution* specifications) [Simão et al., 2011].

We have implemented a Java Specification Request (JSR) not available at current HLL-VMs, the JSR-284 Resource Management API [Grzegorz Czajkowski, 2009]. Our implementation allows the enforcement of JVM-wide policies that act either when resources are being consumed (*constraint policies*) or in response to a given event (*notification policies*) [Simão et al., 2011]. High-level policies are evaluated by consumption points inside the HLL-VM but can be defined externally to the execution environment itself, that is, by the application developer or the provider.

Concurrent checkpointing

We propose a novel solution to Java applications with long execution times, by incorporating checkpoint and migration mechanisms in a Java VM [Simão et al., 2012]. It is able to checkpoint multithreaded applications, ensuring the checkpoint is a consistent snapshot of the execution, taking into account thread concurrency and synchronization, while avoiding application pause by performing the checkpoint concurrently (or incrementally) alongside with application execution.

Our techniques rely on three base mechanisms: on-stack-replacement (OSR), safe yield points (used by the garbage collector) and copy-on-write (COW). The first two mechanisms are available in many other HLL-VM implementations (e.g. Sun HotSpot) while the last one, COW, is generally supported by any modern OS. Therefore, our techniques could be readily applied to other VMs. The main objectives are focused on the problems of transparency and completeness, and how these mechanisms can be activated according to resource management policies.

1.4.3 Scheduling of IaaS resources

Cloud users (i.e., clients) can rent virtual machines with a given memory, CPU, storage, and networking capacity. Clients typically take into account the estimated peak usage of their workloads. To accommodate this simplistic interface, cloud providers have to deal with massive hardware deployments, and all the management and environmental costs that are inherent to such a solution.

In smaller, distributed, and near-the-client cloud deployments, either public, private or hybrid, overcommitment of resources, hardware failures and consolidation may happen more frequently. To help an IaaS provider determine how to transfer resources among tenants, we employ an economic rational based on utility functions. In summary, our main contributions in this area are [Simão and Veiga, 2013b, 2014]:

- An architectural extension to the current relation between cloud users and providers, particularly useful for private and hybrid cloud deployments;
- A cost model which takes into account the clients' partial utility of having their VMs depreciated when in overcommit;
- Strategies to determine, in scenarios where overcommittment can occur, the best distribution of workloads (from different classes of users) among VMs with different execution capacities, aiming to maximize the overall utility of the allocation;
- Extension of a state-of-the-art cloud simulation framework. Implementation and evaluation of the cost model in the extended simulator.

Our work incorporates in the economic model the notions of partial utility degradation in the context of VM scheduling in virtualized infrastructures. It clearly demonstrates that it can render benefits for the providers, as well as reduce user dissatisfaction in a principled-based way. Using our algorithms for overcommited scenarios, the provider's revenue increases with more allocated VMs, even if some are depreciated. The execution time of the workloads decreases as more VMs can be allocated. We demonstrate that this approach has benefits for the providers, users satisfaction in a structured and principled-based way, instead of the typical all-or-nothing approach of queuing, delaying, or rejecting requests.

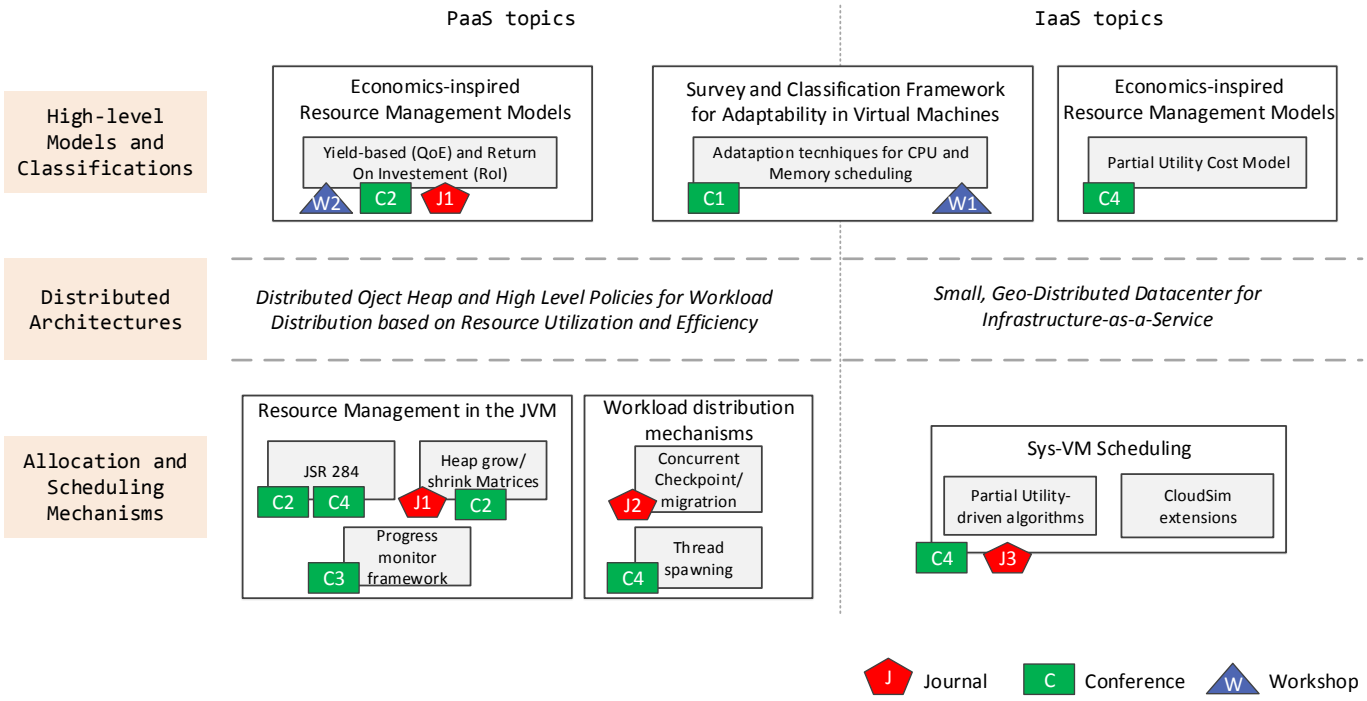


Figure 1.2: Layered view of the researched topics and references to specific contributions

1.4.4 Summary of major publications

Figure 1.2 shows a layered view of the several topics that were researched throughout the PhD work. From top to bottom, it starts by a higher abstraction and systematization layer representing the Economics-inspired resource management models along with a classification framework for adaptability in virtual machines. Then, at the middle layer, it depicts the distributed architectures where we focused our work. Finally, the bottom layer shows the allocation, management and scheduling mechanisms that were developed to support our resource management models. The figure also identifies each topic as related to the PaaS and IaaS service model. Each specific contribution is associated with the corresponding paper, either journal (**J**), conference (**C**) or workshop (**W**), using the references presented next, in the following list of major publications.

The papers that present our survey and classification framework were accepted in a international workshop (co-located with Middleware 2012) and in a international conference (CloudCom 2013) as a short paper. The first describes the idea from the IaaS perspective and the second specializes on memory-only adaptations both at system and

language level virtual machines. They are being extended for an ulterior journal submission.

(W1) José Simão and Luís Veiga, *A Classification of Middleware to Support Virtual Machines Adaptability in IaaS*, 11th International Workshop on Adaptive and Reflective Middleware (ARM 2012), In conjuntion with Middleware 2012, December 2012, ACM.

(C1) José Simão and Jeremy Singer and Luís Veiga, *A Comparative Look at Adaptive Memory Management in Virtual Machines*, IEEE CloudCom 2013, December 2013, IEEE *[short paper]. Ranked C in CORE.*

Regarding the research at the PaaS level, the most relevant works are two international journal articles, one regarding memory and CPU yield-based adaptability and the other describing the concurrent checkpoint mechanism inside a JVM. The companion of these two last articles is the paper accepted in Cooperative Information Systems (CoopIS 2011), where details of the resource monitoring and accounting inside a JVM are presented, along with a thread distribution mechanism based on a middleware for distributed shared objects.

- Resource management mechanisms and Economics-inspired allocation strategies in the JVM:

  (W2) José Simão and Luís Veiga, *VM Economics for Java Cloud Computing - An Adaptive and Resource-Aware Java Runtime with Quality-of-Execution*, The 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2012) - Doctoral Symposium: Cloud Scheduling, Clusters and Data Centers, May 2012, IEEE.

  (C2) José Simão and Luís Veiga, *QoE-JVM: An Adaptive and Resource-Aware Java Runtime for Cloud Computing*, 2nd International Symposium on Secure Virtual Infrastructures (DOA-SVI 2012), OTM Conferences 2012, September 2012, Springer, LNCS.

  (C3) José Simão and Luís Veiga, *A Progress and Profile-driven Cloud-VM for Improved Resource-Efficiency and Fairness in e-Science Environments*, 28th ACM Symposium On Applied Computing (SAC 2013), March 2013, ACM. *Ranked B in CORE.*

(J1) José Simão and Luís Veiga, *Adaptability Driven by Quality Of Execution in High Level Virtual Machines for Shared Environments*. International Journal of Computer Systems Science and Engineering, 28(6), pp. 59-72, November 2013, CRL Publishing. *Q2 in SCImago*.

- Workload distribution mechanisms:

(C4) José Simão and João Lemos and Luís Veiga, $A^2 - VM$*: A Cooperative Java VM with Support for Resource-Awareness and Cluster-Wide Thread Scheduling*, 19th International Conference on Cooperative Information Systems (CoopIS 2011), September 2011, LNCS, Springer. *Core A, Acceptance ratio* $\approx 20\%$*, RADIST B.*

(J2) José Simão and Tiago Garrochinho and Luís Veiga, *A Checkpointing-enabled and Resource-Aware Java VM for Efficient and Robust e-Science Applications in Grid Environments*, Concurrency and Computation: Practice and Experience, 24(13), pp. 1421-1442, September 2012, Wiley. *Q2 in SCImago*.

The work regarding scheduling and cost models for IaaS providers was first presented as a poster in a top-level conference (EuroSys 2013). The definition and usage of our economic models to IaaS was then a distinguished paper in a IEEE conference, CloudCom 2013, which is one of the most cited conferences in the topics of cloud computing [Heilig and Vob, 2014]. This paper was selected for submission to the IEEE Transactions on Cloud Computing (TCC) journal and is now accepted to appear in a future issue. The original CloudCom's paper was extended by i) enhancing and detailing the cost model, ii) discussing how different utility matrices can be compared, iii) comparing the proposed strategy with a more comprehensive list of utility-oblivious algorithms, iv) presenting the results of a larger set of datacenter configurations.

(C5) José Simão and Luís Veiga, *Flexible SLAs in the Cloud with Partial Utility-driven Scheduling Architecture*, IEEE 5th International Conference on Cloud Computing Technology and Science (CloudCom 2013), December 2013, IEEE. *Acceptance ratio* $\approx 17.8\%$, **(Best-Paper Runner-up)**, *ranked C in CORE*.

(J3) José Simão and Luís Veiga, *Partial Utility-driven Scheduling for Flexible SLA and Pricing Arbitration in Clouds*, IEEE Transactions on Cloud Computing, *(to appear)*.

1.5 Outline

This document is organized in four parts as described next:

- **Part I - Thesis Motivation and Artifacts**. This part is divided in three chapters. Chapter 1 presents the thesis motivation and scope along with current shortcomings of the work found in the literature and in commercial systems. Chapter 2 present several systems, mechanisms and algorithms that are related to the work presented in this thesis. In particular, it describes the fundamental building blocks of virtual machines, which are used and extended in the literature to built adaptable systems where resource allocation changes as the workloads are executed. Next, a novel and systematic approach for the classification of adaptability at system-level and high-level language virtual machines is presented. Several state-of-the-art systems are evaluated according to this novel classification framework.

- **Part II - Allocation and Scheduling in Platform-as-a-Service**. This part is divided in three chapters. Chapter 3 starts by presenting the building blocks of the adaptive runtime environment. It describes each building block requirement in order to support adaptation in an application-centric way. It follows with a discussion of our yield-based rationale to drive adaptability, so that resources can be transferred from applications that use them poorly to the ones that can use them more efficiently. Next, Chapter 4 delves into some relevant implementation details of the mechanisms incorporated in a JVM to support progress monitoring, resource allocation, and concurrent checkpoint. This part ends with Chapter 5 which is dedicated to the evaluation of our JVM-level mechanisms and policies. It starts by evaluating how effective the adaptability model is, demonstrating that the tailored allocation of resources to each application has benefits. It concludes with the evaluation of overheads and improvements over the baseline execution of the three adaptation mechanisms.

- **Part III - Allocation and Scheduling in Infrastructure-as-a-Service**. This part discusses the contributions towards the establishment of a new economic relation between clients and providers of the IaaS service model. It is also divided in three chapters. Chapter 6 presents the motivation and overall architecture of the datacenters we are considering, along with a new cost model to be adopted by this

kind of service providers. It shows how the current cost model can be extended to incorporate a partial-utility specification when scheduling the execution of virtual machines. Chapter 7 presents algorithms to determine the best distribution of workloads and maximize the overall utility in a scenario where overcommitment is necessary to accept new requests, instead of delaying or rejecting them. It then discusses the extensions made to a state-of-the-art simulation environment in order to validate and assess the algorithms proposed. Chapter 8 closes this part with the evaluation of the proposed algorithms using simulations of different sizes (i.e., hosts and requested VMs with increasing computing capacity), showing that the approach scales.

- **Part IV - Overall analysis and Future Work**. This chapter concludes the document discussing the overall work and some future directions which we consider as important improvements or extensions, either to the resource management models or the inner mechanisms.

2 Adaptive Mechanisms and Techniques in Virtual Machines

Contents

Chapter overview

Data centers make extensive use of virtualization to achieve workload isolation and efficient resource management, providing the IaaS service model. In general, this is done using virtual machines. In this chapter, we review the main approaches for adaptation and monitoring in virtual machines deployments, their tradeoffs, and their main mechanisms for resource management. We frame them into the control loop [Salehi et al., 2013] where sensors are monitored (e.g. page utilization), decisions are made (e.g. if-else rule, proportional-integral-derivative controller), and actions are performed using actuators (e.g. share page, change heap size). As is common in systems research, improvement in one property is accomplished at the expense of some other property. So, we also propose a classification framework that, when applied to a group of systems, can help visually in determining their similarities and differences. We propose to analyze adaptability techniques in virtual machines using three orthogonal characteristics: responsiveness (R), comprehensiveness (C), and intricateness (I). We then present the details of an extensible classification framework which emphasizes the tradeoffs of different approaches. Using this framework, some representative state-of-the-art systems are evaluated.

2.1 Introduction

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

System-level VMs (Sys-VM), or hypervisors, are strongly motivated by the sharing of low-level resources. As a result, much 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. It ranges from strategies aiming to maximize fairness in the distribution of resources, to those that deliberately favor a given guest based on past resource consumption and prediction on future resource demand. Among all resources, CPU [Zhang et al., 2005; Gong et al., 2010; Hagimont et al., 2013] and memory [Waldspurger, 2002; Mian et al., 2012; Agmon Ben-Yehuda et al., 2014a] are the two for which a larger body of work can be found. Nevertheless, other resources, such as the access to I/O operations, have also been analyzed [Ongaro et al., 2008; Kesavan et al., 2010; Gordon et al., 2012].

High-level language VMs have also been designed as a way to isolate and abstract away 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 et al., 2005a; Binder et al., 2009; Singer and Jones, 2011; Bobroff et al., 2014]). They use different approaches: from making modifications to a standard VM, or even proposing a new implementation from scratch, to modifications in the byte codes and hybrid solutions.

In each work, different compromises are made, putting more emphasis either on the portability of the solution (i.e., not requiring changes to the VM) or on the portability of the guests (i.e., not requiring changes to the application source code).

Virtual machines are not only an isomorphism between the guest system and a host [Smith and Nair, 2005], but a powerful system software layer that can adapt its behavior, or be instructed to adapt, in order to transparently control their guests' performance, in order to comply with a local or global policy. In order to do so, VMs, or systems augmenting their services, can be framed into the well-known adaptation loop [Salehie and Tahvildari, 2009], where systems monitor themselves and their context, detect significant changes, decide how to react, and act to execute such decisions. In this chapter, and similarly to Maggio et al. [Maggio et al., 2012], we consider a control loop with three phases: i) monitoring, ii) decision, iii) actuation. Monitoring determines which components of the system (e.g. hardware, VM, application) are observed. Control and decision take these observations and use them in some decision strategy to decide what has to be changed. Enforcement deals with applying the decision to a given

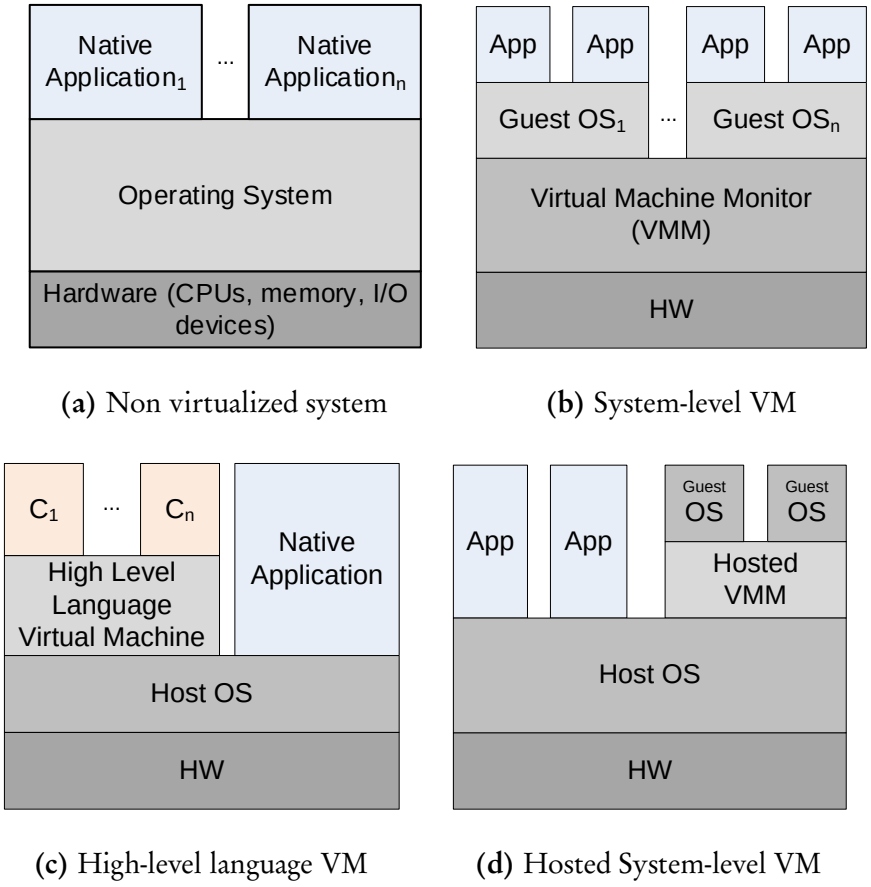
component/mechanism of the VM.

Existing surveys of virtualization technologies (e.g. [Arnold et al., 2005]) tend to focus on a wide variety of approaches which sometimes results only in an extensive catalog. One of the first published surveys of research in virtual machines was presented in 1974 [Goldberg, 1974]. Goldberg's work was focused on the principles, performance and practical issues regarding the design and development of system-level virtual machines that, at the time, where developed by IBM, the Massachusetts Institute of Technology (MIT), and few others. Arnold et al. [Arnold et al., 2005] focus only on HLL-VMs and particularly on the techniques that are used to control the optimizations employed by the just-in-time (JIT) compiler, taking advantage of runtime profiling information.

This chapter surveys several techniques used by virtual machines, and systems that depend on them, to make an adaptive resource management. We present a novel framework to classify resource monitoring and adaptation techniques in virtualized environments, both in Sys-VMs and HLL-VMs. It is grounded on the fundamental techniques used in virtualization and takes an alternative approach to existing survey work. Our framework's goal is to compare different systems regarding three metrics: responsiveness, comprehensiveness, and intricateness. These metrics are used to classify the mechanisms and scheduling policies. This analysis does not try to find the best system, as this depends on the scenario where the system is going to be used, but instead it aims to identity the tradeoffs underpinning each system.

We argue that there are potential overlaps and unexploited synergies between system VM and language VM adaptation processes. It might be the case that techniques from one domain could be transferred profitably to the other. Alternatively, in a system stack, it might be possible for cross-layer exchange of information between these two VM levels to enable co-operative adaptation.

Section 2.2 presents the architecture of high-level and system-level VMs, depicting the building blocks that are used in research concerning resource usage. Section 2.3 presents several adaptation techniques found in the literature and frames them into the adaptation loop. In Section 2.4, the classification framework is presented. For each of the resource management components of VMs, and for each of the three steps of the adaptation loop, we propose the use of a quantitative classification regarding the impact of the mechanisms used by each system. We use this framework to classify state-of-the-



(a) Non virtualized system **(b)** System-level VM

(c) High-level language VM **(d)** Hosted System-level VM

Figure 2.1: Virtualization layers

art systems in Section 2.5, aiming to compare and better understand the benefits and limitations of each one.

2.2 Virtual Machines Fundamentals

Virtual machines have their roots in the 1960's [Rosenblum, 2004] 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, in a time where the notion of *personal computer* did not even exist. IBM's 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 these re-

sources. 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's 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, which are 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 instruction set architecture (V-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 adaptation [Arnold et al., 2005]. 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 a high impact on the use of memory and CPU. After more than three decades of research work focusing on tunning garbage collection algorithms [Jones et al., 2011], recent research work is made towards the selection of application-specific algorithms and parameters, in particular, heap size and the moment of triggering memory collection [Mao et al., 2009; Hertz et al., 2009; Soman et al., 2004].

This section presents fundamental building blocks of virtual machines. These blocks are adapted during the execution of guests in order to reach different execution performance goals. Figure 2.1 depicts four types of deployments. The first is a traditional configuration where an operating system (OS) regulates the access of native applications (i.e., the ones that use the services of the OS) to the hardware. The second, Figure 2.1.b, represents a configuration where an hypervisor, known as virtual machine monitor, takes control of the hardware, making it possible to host several system-level virtual machine on top of the same physical resources. Each virtual machine runs a possibly different operating systems instance. The third depicts the position of high-level language VMs. They are at the level of native applications but support the hosting of *managed* components which rely (almost exclusively) on the services provided by these VMs. Underlying these, finally, and for the sake of completeness, Figure 2.1.d, shows a hosted system-level VM. This chapter focus on deployments b) and c).

Most of the techniques analyzed in this chapter are equally valid for the deployment presented in Figures 2.1.b and 2.1.d and so they are treated as equivalent. However, in practice, and for efficiency reasons, the deployment depicted in Figure 2.1.d usually relies on kernel drivers installed in the hosting OS.

The next three sections will briefly describe how fundamental resources, CPU, memory, and I/O, are virtualized by the two types of VMs. The systems presented in Section 2.5 are based on the building blocks presented here, extending them aiming to implement different adaptive resource management strategies.

2.2.1 Computation as a resource

In a VM, virtualization of computation concerns two distinct aspects: *i)* the translation of instructions, especially if the guest and host use different ISA, *ii)* the scheduling of virtual CPUs to a physical CPU (or CPU core on the now common multicore hardware). 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-level VMs, emulation is necessary to adapt different ISAs or in response to the execution of a privileged instruction (or behavior-sensitive instruction, even if not privileged) in the guest OS. Dynamic binary and byte code translation is achieved by changing the translation technique (between interpretation and compilation) and by replacing code previously translated with a more optimized one. These adaptations are driven by profiling information gathered during program execution.

Regarding CPU scheduling, 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 reason about the program behavior. Regarding System VMs, because they operate directly above the hardware, the VMM must decide the mapping between the real CPUs and each running VM [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, the operating system virtualizes the CPU, scheduling the

runnable tasks to a physical CPU. The distribution of these tasks should be as *fair* as possible. Modern versions of Linux use the Completely Fair Scheduler (CFS) which models an ideal, precise and multitasking CPU, that is, each process has an equal share of the CPU.[1] On a VMM running above the hardware, each guest VM is assigned one or more virtual CPUs (VCPU), whose total number can be larger than the available physical CPUs. When ready, the VCPU needs to be scheduled to a physical CPU. This results in a system with two layers of scheduling: inside the VMM and inside each guest OS.

A VMM scheduler has additional requirements when compared to the OS scheduler, namely the need 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 priority-based approach which is unable to enforce this kind of requirement.

Cherkasova et al. [Cherkasova et al., 2007] classify schedulers as: *i)* work conservative or non-work conservative, and *ii)* preemptive or non-preemptive. Work conservative schedulers take the *share* as a minimum allocation of CPU to the VM. If there are available CPUs, VCPUs will be assigned to them, regardless the VM's share. In non-work conservative, even if there are available CPUs, VCPUs will not be assigned above a given previously defined value (known as *cap* or *cpu limit*). A preemptive scheduler can interrupt running VCPUs if a ready to run VCPU has a higher priority. Section 2.5 presents systems that dynamically change the scheduler's parameters to give guest VMs the capacity that best fits their needs.

2.2.2 Memory as a resource

Regardless of the target environment, designing memory management strategies is a demanding task. Virtual machines (VMs), either virtual machine monitors (Sys-VMs) or high-level language runtimes (HLL-VMs), add an extra level of complexity.

Sys-VMs (e.g. Xen) host multiple isolated guest instances of an operating system (OS) on multi-core architectures, sharing computational resources in a secure way. From an abstract perspective, managing memory in a Sys-VM is a generalization of operations

[1] http://www.linuxjournal.com/magazine/completely-fair-scheduler, visited August 19, 2014

performed by a classical OS [Smith and Nair, 2005]. However, in practice, many Sys-VMs gain performance advantages by dynamically adapting their memory management strategies.

HLL-VMs (e.g. JVM) have a single guest application, even when there is more than one address space, i.e., *application domains* in the Common Language Runtime (CLR). Again, HLL-VMs adapt memory management decisions based on the dynamics of a given workload. Actions include heap resizing or, in more extreme scenarios, changing the garbage collection algorithm to one that saves memory at the expense of some performance.

Conceptually, Sys-VMs and HLL-VMs perform the same memory management task, i.e., they mediate hosted application access to an underlying, potentially scarce, address space. Memory is virtualized to give the illusion to their guests of a virtually unbounded address space. Because memory is effectively limited, it will eventually become full and the guest (operating systems or application) will have to deal with memory shortage.

In system-level VMs, an extra level of indirection is added to the already virtualized environment of the guest operating systems, which give to their guests (i.e., processes) a dedicated virtual address space, usually larger than the real available hardware. As pointed out by Smith et al. [Smith and Nair, 2005], the VMM extra level of indirection generalizes the virtual memory mechanisms of operating systems.

In HLL-VMs, memory is requested on demand by the guest application, without the need to be explicit freed by it. When a given threshold is reached, a garbage collection activity is started to detect unreachable objects and reclaim their memory. There is no "one size fits all" garbage collector algorithm.

Memory management in System VMs

The VMM can be managing multiple VMs, each with his guest OS instance and type. Therefore, the mapping between physical and real addresses must be extended because what is seen by a guest 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.

When an operating system kernel, running on an active VM, uses a *real* address to perform an operation (e.g., I/O), the VMM must intercept this address and change it

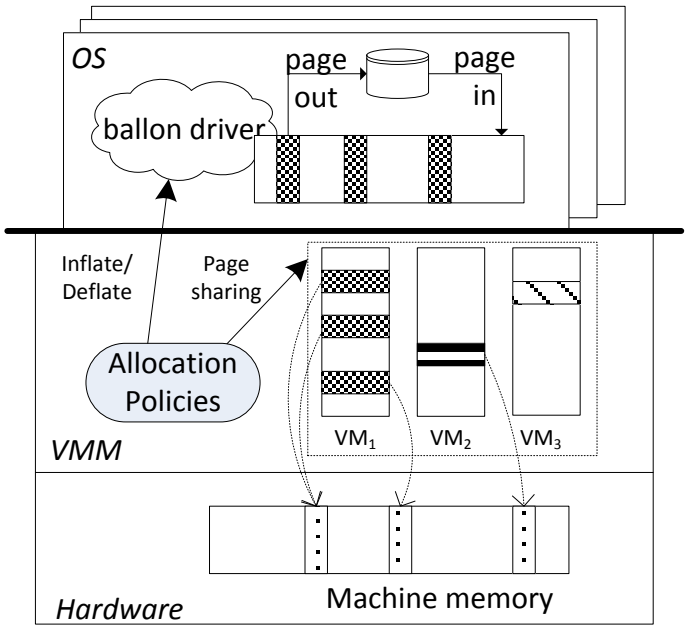


Figure 2.2: The control loop of memory management in hypervisor based deployments

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

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. The decision of which specific pages should be removed from memory is usually left to the guest OS, using a kernel driver known as the *balloon* driver [Barham et al., 2003; Waldspurger, 2002].

Figure 2.2 depicts the balloon driver available at each guest. The balloon driver is controlled by memory management policies which will be introduced in Section 2.3. When the balloon is instructed to inflate it will make the guest OS swap memory to secondary storage. When the balloon is instructed to deflate, the guest OS can use more physical pages, reducing the need to swap memory. Another issue related to memory

management in the VMM is the sharing of machine pages between different VMs. If these pages have code or read-only data they can be shared avoiding redundant copies.

Automatic memory management in High Level 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 references to data structures (i.e. objects), promoting easier extensibility of functionalities because the bookkeeping code that must be written in non-virtualized environment is no longer needed [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, containing thread stacks and globals). All these approaches strive for a balance between the time the program needs to stop and the frequency the collecting process needs to execute. This is mostly influenced by the heap dimension and, in practice, some kind of nursery space is used to avoid searching all the heap.

New objects are created in a small space (e.g. 512 KBytes). When this space fills-up, live objects are promoted to a bigger space, leaving the nursery empty and ready for new allocations. These collectors are called *generational collectors*. The nursery space can be generalized and the heap organized in more than two generations. As parallel hardware becomes ubiquitous and GC pause time reduction becomes essential, the stop-the-world has been questioned, resulting in the design of concurrent and incremental collectors [Click et al., 2005; Tene et al., 2011]. However, recent studies show that the base approach has no fundamental scalability problem [Gidra et al., 2013] and that the GC impact can be diminished with parallel techniques, which still need to stop the program, but that explore the root set in parallel.

Researchers have analyzed garbage collection performance and found it to be application-dependent [Soman and Krintz, 2007] and even input-dependent [Mao et al., 2009; Tay et al., 2013]. Based on these observations, several adaptation strategies have been proposed [Arnold et al., 2005], ranging from parameters adjustments (e.g. the current size of the managed heap [Guan et al., 2009; Singer and Jones, 2011]) to changing the algorithm itself before the first execution [Singer et al., 2011] or at runtime [Soman and

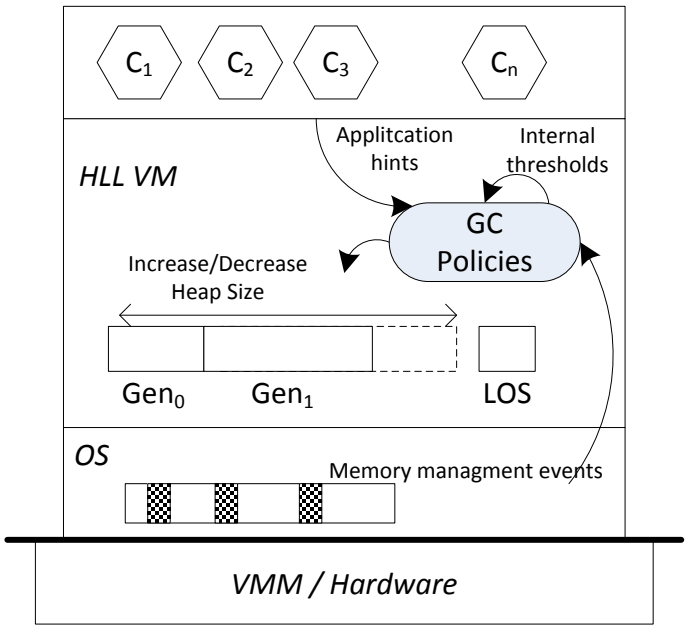


Figure 2.3: The control loop of memory management in HLL VM based deployments

Krintz, 2007].

Figure 2.3 depicts different events that have been researched to guide the moment of garbage collection and the resizing of the heap size. These events, which are either collected from the application key performance indicators, or from the operating system, are taken into account by the memory management policies of the garbage collection sub-system. These policies determine when to act on selected actuators (e.g. heap size, GC algorithm). Section 2.3 discusses these different approaches.

2.2.3 Input/Output as a resource

In both types of VMs, virtualization of input/output deals with the emulation, accounting and constraining of using available physical devices. In spite of these similar goals, virtualization occurs with different impacts. In a VMM, the access to device drivers can be para-virtualized or full virtualized. In the first scenario, a cooperative guest OS is expected to call a virtual API in the VMM [Barham et al., 2003]. In the second scenario (a full virtualized environment) the VMM can either intercept the I/O operation, at the device driver or system call level [Smith and Nair, 2005]. The typical option is to virtualize at the device driver level, installing virtual device drivers at each guest, which, from the guest operating system standpoint, are regular drivers.

The main challenge in I/O virtualization for fully virtualized systems, such as the ESX [Waldspurger, 2002] or the KVM [Lublin et al., 2007] hypervisors, is to avoid the extra context switches between the guest and the host to handle interrupts generated by I/O devices [Adams and Agesen, 2006; Gordon et al., 2012]. The interrupts are, by nature, asynchronous and sent to the CPU to signal the completion of I/O operations. So, the overhead comes from the extra CPU cycles necessary to exit the guest, run the host interrupt handler and inject the virtual interrupt in the guest.

The performance of I/O-intensive applications in a virtualized environment is also affected by the CPU scheduling and memory sharing mechanisms [Cheng and Wang, 2012; Ram et al., 2010; Ongaro et al., 2008; Cherkasova et al., 2007]. The CPU scheduling strategy of each physical cores to the virtual cores has impact in the I/O performance of the applications running on top of virtual machines. A detailed analysis of the scheduler's impact on VM's performance is available in the literature [Ongaro et al., 2008; Cherkasova et al., 2007]. The main observations were related to the domain driver's preemption during the dispatch of multiple network events and the order of VMs in the run queue.

High-level language VMs rely on the operating system API to accomplish input/output operations as disk and network read and writes. Depending on the address space isolation supported by the VM, accounting and regulation have different levels of granularity. In a classic JVM implementation, accountability can be done globally at the VM or on a per-thread basis [Suri et al., 2001]. In HLL-VMs supporting the abstraction of different address spaces (e.g. isolates in Multi-task VM [Czajkowski et al., 2005a], application domains in the Common Language Runtime) accounting is made independently for each of these spaces.

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

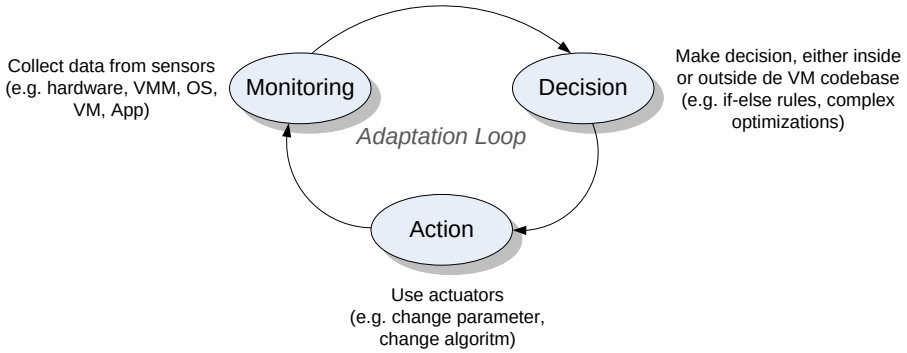


Figure 2.4: Adaptibility loop

2.3 Adaptation techniques

In a software system, adaptation is regulated by monitoring, analyzing, deciding and acting [Salehie and Tahvildari, 2009]. Monitoring is fed by sensors and actions are accomplished by actuators, forming a process known as the *adaptation loop*, as depicted in Figure 2.4. Virtual machines, regardless of their type, are no exception.

In a broad sense, virtual machines have an important property of autonomic systems which is self-optimization [IBM, 2005]. An example are the adaptive JIT compilation techniques of HLL-VMs [Arnold et al., 2005] or GC algorithms that use feedback-directed online techniques to avoid page faults [Grzegorczyk et al., 2007]. Furthermore, virtual machines export adaptability mechanisms that are used by outside decision systems to reconfigure VM's parameters or algorithms. Examples include the work of [Padala et al., 2009] which periodically imposes a new CPU limits (i.e. setting the *cap* parameter) to each VM controlled by their system. Gingko [Hinesa et al., 2011] also places the decision step outside the hypervisor, acting upon the balloon controller to automatically transfer memory between VMs.

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

Next we will present and discuss the state of the art regarding the three major steps

of the adaptation loop for each type of VM and their internal resource management mechanisms.

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 - monitoring, decision, and acting.

CPU Management

CPU management relates to activities that can be done exclusively inside the hypervisor or both inside and outside. An example of an exclusively inside activity is the CPU scheduling algorithm. To enforce the weight assigned to each VM, the hypervisor has to monitor the time of CPU assigned to each VCPUs of a VM, decide which VCPU will run next, and assign it to a CPU [Shao et al., 2009; 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) but then use its internal actuators to adjust parameters. For example, monitoring the waiting time inside the spin lock synchronization primitive (in the kernel of the guest operating system) may be necessary to inform the hypervisor's scheduler about the best co-scheduling [Ousterhout, 1982] decisions of VCPUs [Weng et al., 2011].

Decision strategies can be simple, like the proportional share-based that enforces predefined *shares* defined by high level policies in a multi-tenant environment. More complex decisions, made outside the hypervisor, include: i) control theory using a PID controller [Zhang et al., 2005; Park and Humphrey, 2009], ii) linear optimization [Padala et al., 2009], iii) signal processing and statistical learning algorithms [Gong et al., 2010].

The actions taken by the CPU scheduler inside the hypervisor include: i) number of VCPUs [Shao et al., 2009], ii) co-scheduling [VMware, 2009; Weng et al., 2011, 2009], iii) VCPU migration [VMware, 2009], iv) number of threads and sleep time [Zhang

et al., 2005]. Systems where decisions are made outside the hypervisor use the available actuators, namely: i) VCPUs share, ii) VCPUs *cap* [Gong et al., 2010; Padala et al., 2009; Heo et al., 2009].

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 utilization using either page or sub-page scope. In this step of the control loop, the VMM needs to determine how pages are being used by each VM. To do so, it must collect information regarding: i) page utilization [Waldspurger, 2002; Weiming and Zhenlin, 2009], ii) page (and sub-page) contents equality or similarity [Waldspurger, 2002; Gupta et al., 2008]. Some systems also propose to monitor application performance, either by instrumentation or external monitoring, in order to collect information closer to the application's semantics [Hinesa et al., 2011; Salomie et al., 2013].

Because operating systems do not support dynamic changes to physical memory, the maximum amount of memory that can be allocated is statically configured for each VM. Nevertheless, the VMM supports overcommit, that is, the total memory configured to the overall VMs can be higher than the one that is physically available. When in overcommit, pages of memory need to be transfered between VMs. To determine which guest OS must relinquish pages in favor of other guests, decisions are made using i) shares [Waldspurger, 2002] ii) feedback control [Heo et al., 2009], iii) LRU histogram [Weiming and Zhenlin, 2009], iv) linear programming [Hinesa 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], 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, using the balloon driver described in Section 2.2.

2.3.2 High-Level Language Virtual Machine

In this section, the three major language VM subsystems, JIT compiler, GC and Resource manager, will be framed into the adaptation processes. HLL-VMs monitor events inside their runtime services or in the underlying platform. As always, there is a trade

off between deciding fast but poorly, or deciding well (or even optimally), but spending too much resources and time in the process of doing so. Most systems base their decision on an heuristic, that is, some kind of adjustment function or criterion that, although it cannot be fully formally reasoned about, it still gives good results when properly used. Nevertheless, some do have a mathematical model guiding their behavior [Tay et al., 2013]. 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.[2] Hot methods information is acquired using: i) sampling, ii) instrumentation. In the first case, the execution stacks are periodically observed to determine hot methods. In the second case, method code is instrumented so that its execution will fill the appropriate runtime profiling structures. Sampling is known to be more efficient [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., 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; Kulkarni and Cavazos, 2012].

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

[2]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 'bloat', one of DaCapo's benchmarks [Blackburn et al., 2006].

Garbage collection

Traditional 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 commercial 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. heap in used) [Singer et al., 2010, 2011], ii) GC statistics (e.g. GC load, GC frequency) [Soman and Krintz, 2007], iii) relevant events in the operating systems (e.g. page faults, allocation stalls) [Grzegorczyk et al., 2007; Hertz et al., 2011], iv) working set size [Yang et al., 2006].

Decision regarding the adaptation of heap-related structures are taken either *i)* offline, or *ii)* inline with execution. Offline analysis takes into consideration the result of executing different programs to see which parameter or algorithm has the best performance for a given application. Inline decisions must be taken either based on a mathematic model or on some kind of heuristic. Some authors have elaborated mathematical models of objects' lifetimes. These models are mostly used to give a rationale of the GC behavior, rather than being used in a decision process [Baker, 1994]. Thus, most systems have a decision process based on some kind of heuristics. The decision process includes: i) machine learning, ii) PID controller, iii) microeconomic theories such as the elasticity of demand curves.

Similarly to the JIT compiler, adaptability regarding memory management aims to improve overall system performance. Classic GC algorithms provide base memory virtualization. Recent works have been focused on optimizing memory usage and execution time, taking in consideration not only the program dynamics and but also the state of the execution environment [Hertz et al., 2009]. Some work also adapts GC to avoid memory exhaustion in environments where memory is constrained [Soman and Krintz, 2007]. 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., 2005a], or *ii)* byte code instrumentation [Hulaas and Binder, 2008]. In the former, it is possible to collect more information, both from a quantitative as a qualitative perspective. A well-known example is the Java Virtual Machine Tool Interface, which is mainly used by development environments to display debug information.[3] Because HLL-VMs do not necessarily expose this kind of service, instrumentation allows some accounting in a portable way. Accounted resources usually include CPU usage, allocated memory and relevant system objects such as threads or files.

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

This subsystem is particularly important in VMs that support several independent processes running in a single instance of the runtime. Research and commercial systems apply resource management actions to: i) limit resource usage, ii) resource reservation. Limiting resource usage aims to avoid denial of service, or to ensure that the (possibly payed) resource quota is not overused. The last scenario is less explored in the literature [Czajkowski et al., 2005a]. Resource reservation ensures that, when multiple processes are running in the same runtime, it is possible to ensure a minimum amount of resources to a given process.

2.3.3 Summary of techniques

Figure 2.5 presents the techniques used in the adaptation loop. They are grouped by the two major adaptation targets, CPU and memory, and then into the three major phases of the adaptation loop. The CPU management sub-tree is the one that has more elements (i.e., more adaptation techniques). This reflects the emphasis given by researchers to this component of Sys-VMs. Regarding memory, early work of Waldspurger [Waldspurger, 2002] and Barham et al. in [Barham et al., 2003] laid solid techniques for virtualizing and managing this resource. Recent work shows that, to improve perform of workloads

[3] `http://docs.oracle.com/javase/7/docs/technotes/guides/jvmti/`, visited July 1, 2014

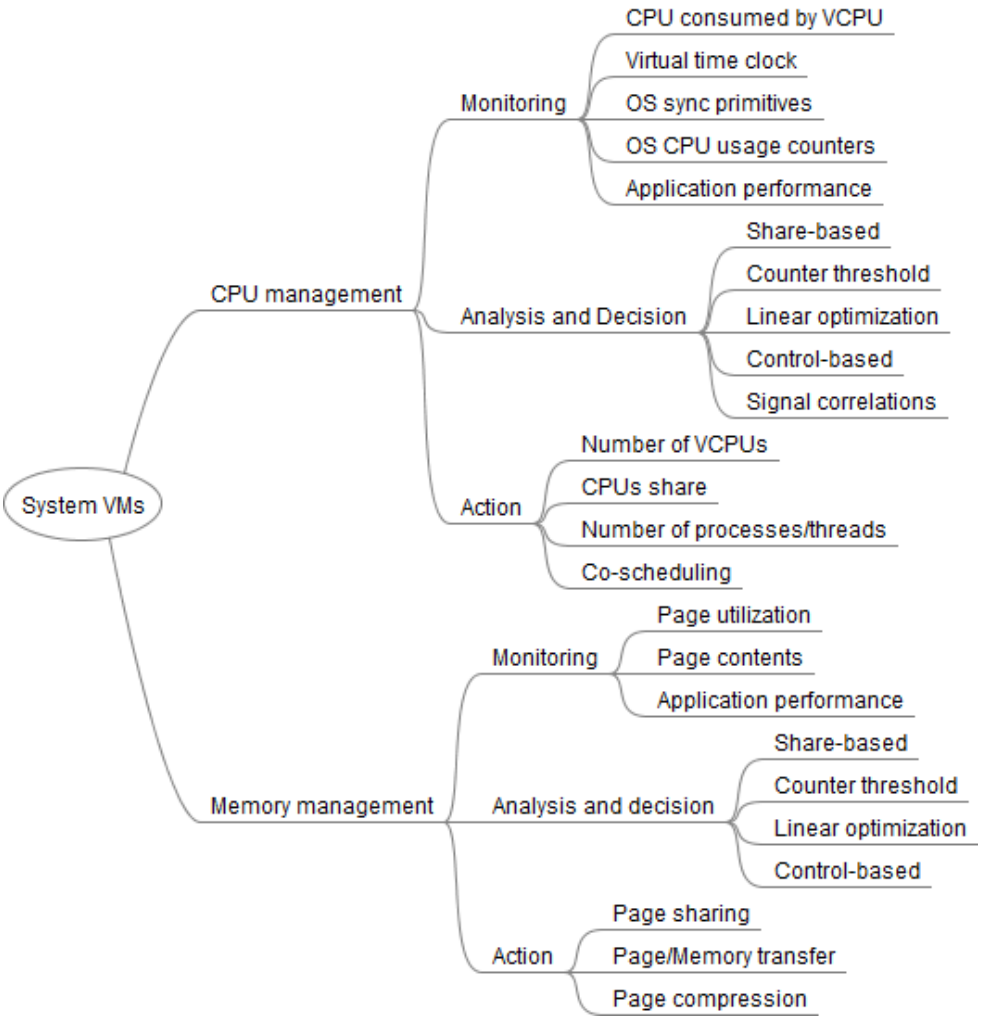


Figure 2.5: Techniques used by System VMs in the monitoring, decision and action phases

regarding their use of memory, it is crucial to have more application-level information [Weiming and Zhenlin, 2009; Hinesa et al., 2011].

Figure 2.6 presents the techniques used in the adaptation loop of systems using HLL-VMs. They are grouped into the three major adaptation targets i) JIT compiler, ii) garbage collection, iii) resource management. Each adaptation target is then divided into the three phases of the adaptation loop. The garbage collection sub-tree has a higher number of elements when compared with any of the other two. This reflects different research paths, but also a higher dependency of the garbage collection process on the workloads and on the context of execution (i.e., shared environment, limited memory,

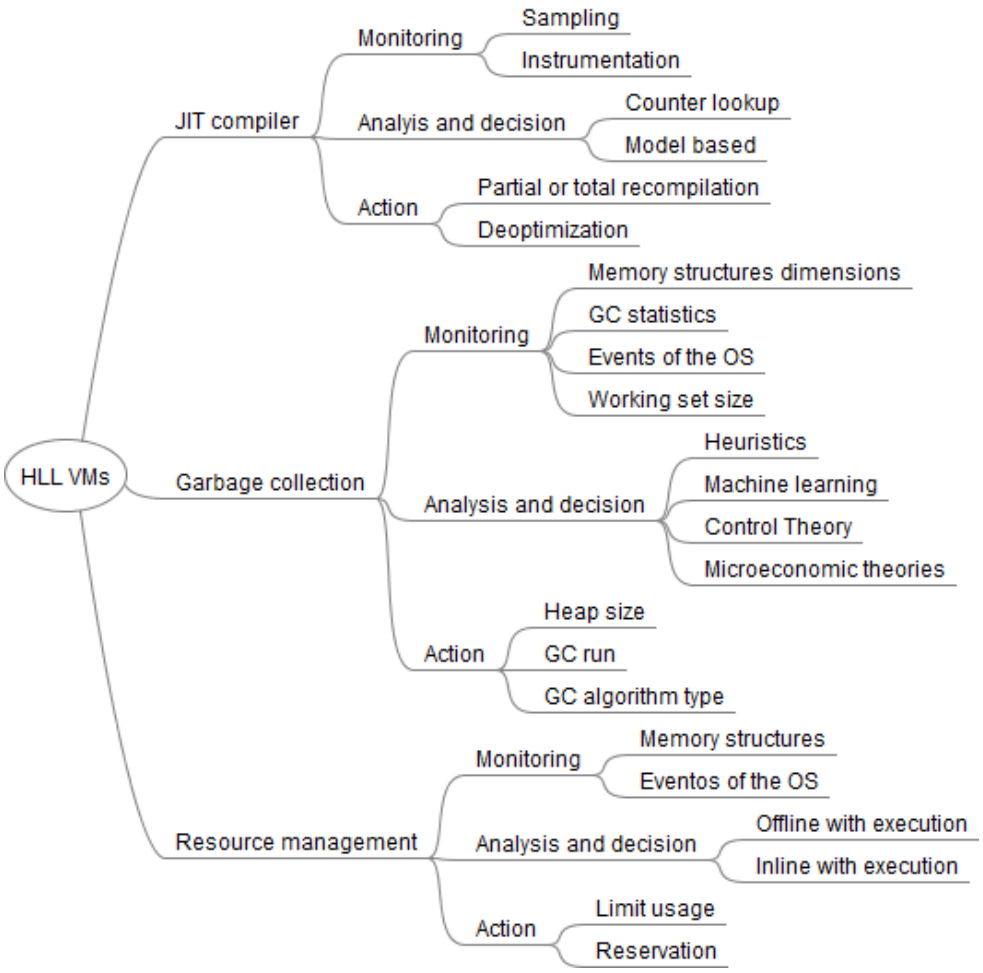


Figure 2.6: Techniques used by HLL-VMs in the monitoring, decision and action phases

etc.).

The techniques used in the monitoring and action phase are domain-specific. For example, there are sensors related to the utilization of memory pages or actuators that change a parameter in the garbage collection algorithm. On the contrary, the strategies used in the decision phase can be found in other adaptability works and, in general, in autonomic computing systems [Salehie and Tahvildari, 2009; Maggio et al., 2012].

Maggio et al. [Maggio et al., 2012] have focused attention on the characterization of decision techniques. They divide them into three broad categories: heuristics, control-based, and machine learning. In fact, we can also see these categories when we look to the techniques identified in this section. Figure 2.5 and Figure 2.6 show that the decision strategies are either heuristic (e.g. microeconomics, share-based), control-based

(e.g. PID controller), based on signal processing techniques (e.g. correlation of different windows of samples), and machine learning (e.g. reinforcement learning). Regarding strategies that use linear programming, they are used only to make a general model of the scheduling variables. In practice, these approaches use integer linear programming which is known to be NP-hard. Thus, they use some kind of greedy approach to solve it.

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

2.4 The RCI Framework for classification of VM adaptation techniques

To understand and compare different adaptation processes we now introduce a framework for classification of VM adaptation techniques. The classification is based on the different techniques described earlier and depicted in Figure 2.5 and Figure 2.6. The analysis and classification of the techniques and the way they are used in each of the adaptation loops revolves around three fundamental criteria: ***R**esponsiveness*, ***C**omprehensiveness* and ***I**ntricateness*. We call it RCI framework. Our goal is to put each system in perspective and compare them regarding three criteria. The final RCI values of a given system depend on the techniques the system uses for monitoring, decision and acting.

These aspects were chosen, not only because they encompass many of the relevant goals and challenges in VM adaptability research, but mainly because they also seem to 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 expense of the remaining one*.

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

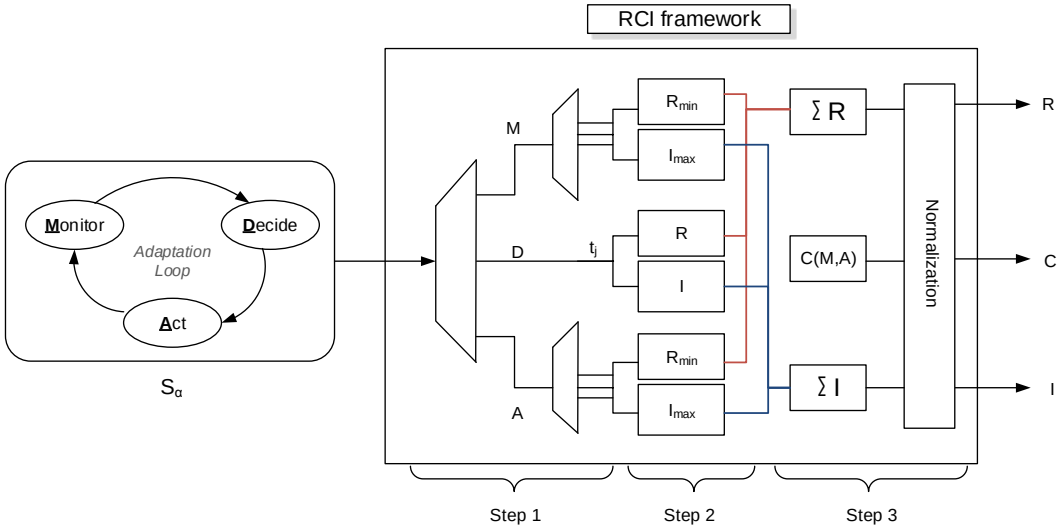


Figure 2.7: A step-by-step classification process

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 is the tension, in the domain of peer-to-peer systems, among high availability, scalability, and support for dynamic populations [Blake and Rodrigues, 2003].

The framework starts by taking the input system and decomposes it into the adaptation techniques used in the monitoring, decision, and acting phase. This is represented in step 1 of Figure 2.7. Then, for each technique, a value for R, and I is determined (step 2). The metric C is determined in step 3 by taking into account the order of magnitude of the number of sensors and actuators. Also in step 3, the previous values are aggregated and normalized, determining the final RCI tuple for the system.

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

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

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

These two steps are detailed in the following sections. First, Section 2.4.1 discusses a quantitative criteria, where design options, representing groups/classes of techniques,

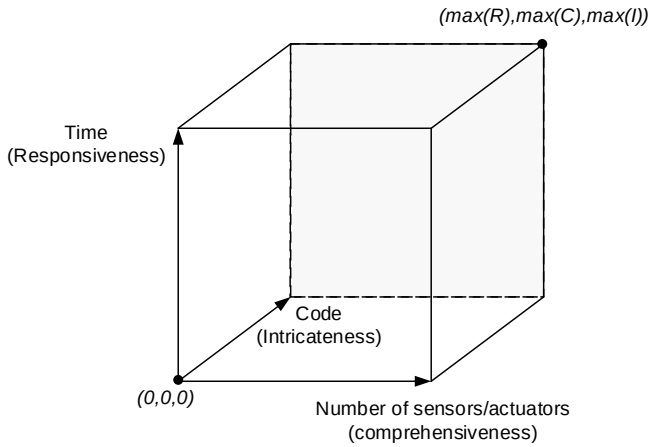


Figure 2.8: Systems design interval

are assigned a single value. Next, Section 2.4.2 maps the set of specific techniques presented in Section 2.3 to these classes, so that each technique is assigned a unique value of R and I. This completes step 2 of the classification process. Finally, Section 2.4.3 explains the rationale of step 3, showing how the previous values are aggregated with the C metric to determine a system's RCI.

2.4.1 Quantitative Criteria of the RCI framework

We think the three metrics are able to capture a *design interval* as presented in Figure 2.8. They capture time, space and complexity-related characteristics. Our conjecture is that we will see systems that are away from the minimum and the maximum of the cube, that is, neither too simple (e.g., near the base of the coordinates) nor excelling in the three metrics (e.g., near or coincident with the maximum point in the design space). The following list points the exact meaning of the three criteria, regarding each of the adaptation phases. Next, we will detail how they are mapped to a numeric scale, in each phase, which will be used to determine the RCI of systems.

- **Responsiveness**. It captures time-related characteristics of the techniques. Regarding the monitoring phase, it depends on the latency of reading a value. Higher values are assigned to sensors immediately available on the VM code base, where higher values represent external sensors (operating system or application-specific). For the decision phase, responsiveness is lower in those techniques that take longer

to reach a given adaptation target. Regarding the action phase, high values indicates that the effect is (almost) immediate, while a low value represents actuators that will take some time to produce effects.

- **Comprehensiveness**. It captures quantity-related characteristics of the techniques. Regarding the monitoring and deciding phases, it gets higher as the quantity of the monitored sensors increases. Regarding the acting phase, the comprehensiveness value grows with the quantity of actuators that the system can engage.

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

Figure 2.9 represents each of these criteria (R, C, I) for the three adaptation phases (M, D, A). For each criteria, in each adaptation phase, the figure shows the several options there are for the classification of a given technique, used in step 2 of the classification process. It does so by showing the mapping between a design option (e.g. use a sensor that is an extension inside the VM) and a quantitative value. These values establish an order among different options.

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

The mapping between classes and specific techniques will be presented next, in Section 2.4.2. Note also that the scale of the values is not important (they typically represent different orders of magnitude) as long as the values are positive and monotonically

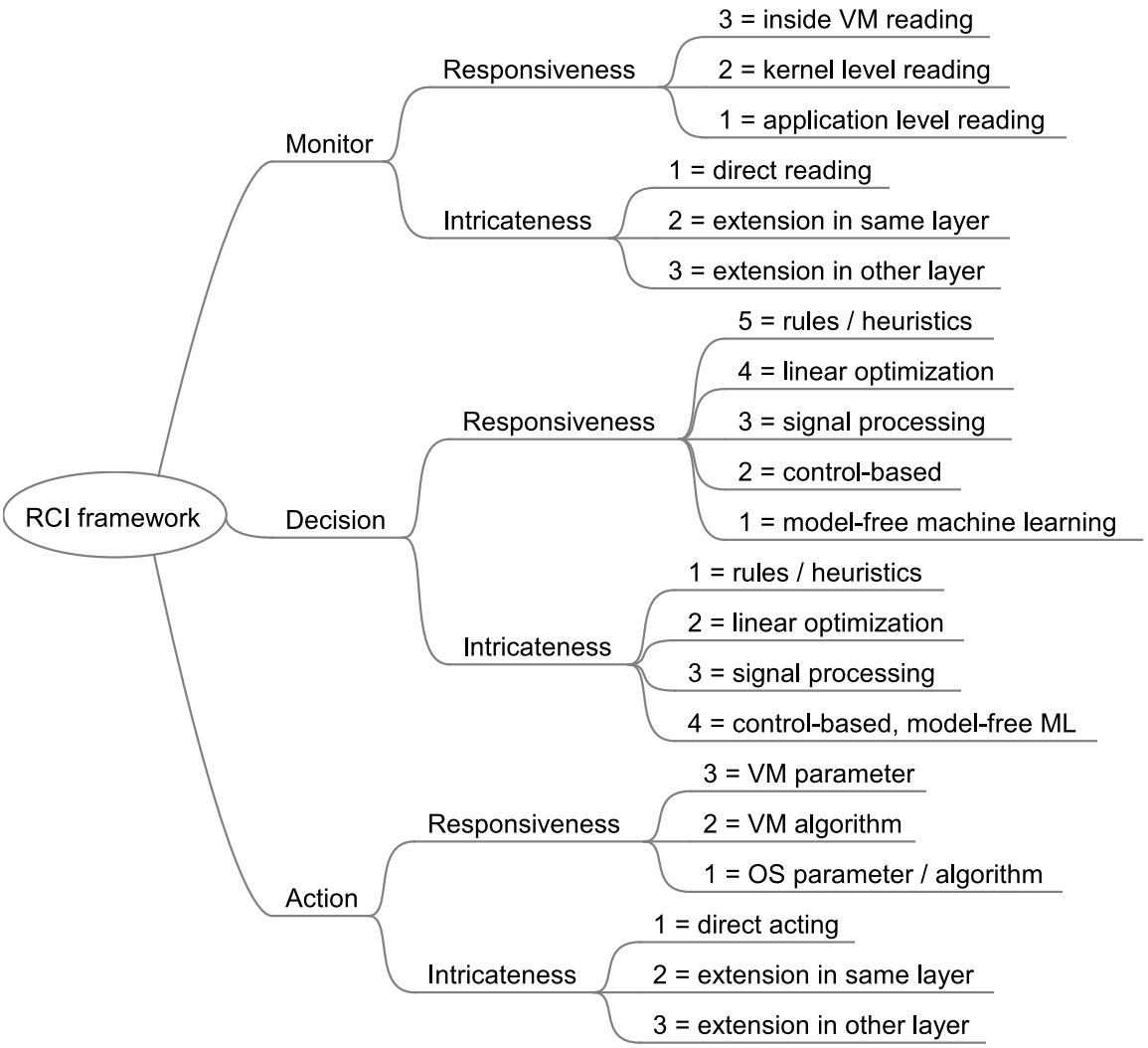


Figure 2.9: Quantitative values for the design options of the RCI framework

increasing or decreasing, in accordance with the corresponding criteria.

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

- **Monitoring**. The responsiveness of the monitoring phase depends on the cost of reading. The cost of reading relates to the time spent in reading a single value, that is, how fast can a single value be collected. This depends on the layer where the sensor is in relation to where the decision is made. For example, some systems use application-level monitors which require inter-process communication to read them (e.g. number of completed SQL transactions [Hinesa et al., 2011]). Others depend only on values collected inside the virtual machine monitor or the HLL-VM context. A middle ground approach is that of systems that depend on sensors from other layers, such as the OS, but, reading them has a low cost (e.g. the `/proc` virtual file system).

  The intricateness of the monitoring phase is a measurement of how complex is the code for reading sensors. Value 1 is assigned to systems that use preexisting sensors of the virtual machine or in the execution environment, which have a direct access. Value 2 is for extensions made inside the virtual machine and value 3 is assigned when extensions were made in the underlying system and/or hardware (e.g. operating system, in the case of HLL-VMs).

- **Deciding**. The responsiveness and intricateness of the deciding phase is in a large part inspired by the study of Maggio et al. [Maggio et al., 2012]. They discuss how feedback control mechanisms compare to each other in the context of a benchmark suite composed of multi-threaded programs, instrumented with the Application Heartbeat framework [Hoffmann et al., 2010]. Taking into account the analyzed techniques, our classification framework is based on five decision types i) rules/heuristics; ii) linear optimization; iii) control-based solutions; iv) signal processing techniques; v) model-free machine learning solutions.

  We have classified these five types of decision strategies as decreasingly responsive, because they take an increasing amount of time to reach a certain target

point. They are increasingly intricate with the exception of control-based solutions which we consider more intricate than signal processing. This is so because of the panoply of parameters that usually have to be tunned. A model-free solution has also the highest intricateness value because the tunning of assigning credits to each possible action and the balance between exploitation versus exploration (i.e. balancing between making the best decision given current information or explore more system states) [Maggio et al., 2012].

- **Acting**. In this phase, responsiveness reflects the capacity of the actuator to produce an observable and measurable consequence. Any throttle to the processing capacity will have almost immediate effect and so a value of 1 is assigned to this type of actuators. Regarding memory, tweaking the set of pages assigned to a VM will have a quicker impact than simply changing its memory share. Changing heap parameters is, in comparison with the other techniques, the least responsive one, and so it gets a value of 3. Intricateness has, in this phase, a similar characterization to the one made in the monitoring phase.

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

2.4.2 Classification of techniques

Tables 2.1-2.3 refer to system-level virtual machines and map a specific sensor, actuator or decision technique to a particular value. This is done according to the classes and values defined in Figure 2.9, which was discussed in the previous section. For each line, the first column identifies a technique (as presented in Figures 2.5 and 2.6) while the second and third columns contain a design class and the corresponding value, for responsiveness (second column) and intricateness (third column). Tables 2.4-2.6 are the ones corresponding to the high-level language virtual machines and follow the same logic.

Looking at the techniques used in the monitor phase, Tables 2.1 and 2.4 show us that only two techniques have the minimum responsiveness. This is so because most of the sensors are *near* the VM execution space (either in a sub-system of the VM or in the operating system). Low intricateness also is dominant as most sensors are already available. Regarding the decision phase, analyzed in Tables 2.2 and 2.5, a majority of

Table 2.1: System VMs: Sensors

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

Table 2.2: System VMs: Control techniques

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

techniques have high responsiveness values. As a consequence, they are less intricate. In HLL-VMs, techniques are usually either very simple or have maximum complexity. Finally, regarding the action phase, we note that all actuators are either already available in the VM code base or are extensions to the VM code base. Contrary to sensors, no new actuators are proposed for other layers of the execution stack. This leads to not having, in practice, actuators with the maximum intricateness.

Table 2.3: System VMs: Actuators

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

Table 2.4: HLL VMs: Sensors

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

Table 2.5: HLL VMs: Control techniques

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

Table 2.6: HLL VMs: Actuators

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

2.4.3 Aggregation of quantities

In this section, we give the details about the implementation of the final stage of step 2 and how step 3 operates, as depicted in Figure 2.7.

Regarding the final stage of step 2, because a given system may use more than one sensor, in the monitoring phase, and more than one actuator, in the acting phase, the framework must determine how a single R and I value is determine for these two phases (i.e. R_M, R_A, I_M, I_A). Regarding responsiveness, we consider the technique with the lowest responsiveness, as presented in Equation 2.1. This was so because the monitor or the action phase will be as responsive as the least responsive technique the system uses. Regarding the intricateness metric, we use the technique with the highest value as a representative of the phase's intricateness. Finally, note that this is not an issue for the decision phase because specific systems only use one strategy.

$$R_\pi = minimum\ of\ techniques'\ responsiveness,\ where\ \pi\ \in\ \{M, A\} \tag{2.1}$$

For a given system, S_α, the three metrics of the framework, responsiveness, comprehensiveness and intricateness, are represented by $R(S_\alpha)$, $C(S_\alpha)$, $I(S_\alpha)$, respectively. Each of these metrics depends on the specific values of the techniques used by the system. So, to determine $R(S_\alpha)$, the framework adds the responsiveness of each phase of the adaptation loop (**M**onitor, **D**ecision, **A**ction), as presented in Equation 2.2. A similar operation is done to determine the intricateness metric.

$$R(S_\alpha) = \sum_{\pi\ \in\ \{M,D,A\}} R_\pi(S_\alpha) \tag{2.2}$$

To determine comprehensiveness, $C(S_\alpha)$, the framework takes into account the number of sensors used in the monitoring phase, the number of actuators used in the acting phase, and adds them to reach a single value. This is the operation identified as $C(M, A)$ in step 3 of Figure 2.7.

As an example, consider system S_α, which uses several hypothetical techniques for each phase of the adaptation loop. The last line of Table 2.7 shows the result of the *aggregation* operations used to determine, for each of the three phases, the **R** and **I** values. The aggregate function $minimum$ is used for responsiveness, while the aggregate

Table 2.7: Example of the aggregations made in step 2 for system S_α

System	Monitor	R	I	Decision	R	I	Action	R	I
S_α	T_a	2	3	T_d	2	3	T_e	1	2
	T_b	3	2				T_f	2	1
	T_c	1	2						
		1	**3**		**2**	**3**		**1**	**2**

function $maximum$ is used for intricateness. Table 2.8 completes the example, showing the *arithmetic* operations necessary to determine the overall **R**, **C**, **I** values of system S_α. The values from the last line of Table 2.7 are the ones used to determine **R** and **I** in Table 2.8, following the equation 2.2.

Table 2.8: Example of the arithmetic operations in step 2 for system S_α

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

2.4.4 Critical analysis of the framework

The RCI framework is the first work that tries to make this kind of classification, aiming to show trade-offs in the design of adaptive systems in the context of virtual machines. Its critical point is the design options tree, presented in Figure 2.9, and the corresponding quantitative values. It can be the case that either the design options do not represent the entire design space or that the quantitative values are not correctly assigned. We tried to minimize this by designing the framework after examining several systems to better understand the scope of the design space. However, we are still missing to collect the opinion of other researchers in the area while using the framework, and possibly improve it based on theirs feedback.

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.

2.5 VM systems and their classification

In this section we start by surveying several state of the art systems, regarding system-level VMs, Section 2.5.1, and high-level language VMs, Section 2.5.2. In each case we frame the analyzed systems into the classification framework presented in Section 2.4, describing each of the techniques used, resulting in the classification and comparison of complete systems.

2.5.1 System Virtual Machine

The following are succinct descriptions of system-level VMs and systems that extend them. We start by presenting a well known open-source hypervisor. A list of systems that extend this or other similar hypervisors follows. Most of them are centered either on CPU or memory. At the end of the section, Table 2.9 summarizes the techniques used in each system. This process was identified as step 1 in Figure 2.7. This is the base for determining each system's RCI.

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.

Xen includes three scheduling algorithms: Borrow Virtual Time (BVT), Simple Earliest Deadline First (SEDF) and Credit [Cherkasova et al., 2007]. [4] The former two are deprecated and at the time of this writing the documentation mentions that they will be removed from the available schedulers. Credit is a proportional fair scheduler. This means that the interval of time allocated for each VCPU is proportional to its *weight*, excluding small allocation errors. Additionally to *weight*, each *domain* has a *cap* value representing the percentage of extra CPU it can consume if its quantum has elapsed and there are idle CPUs. At each clock tick, the running VPCUs are charged and eventually some will loose all their *credit* and tagged be as *over*, while the others are tagged *under*.

[4] `http://wiki.xenproject.org/wiki/Credit_Scheduler`, visited at June 16, 2014

VCPUs tagged as *under* have priority in scheduling decisions. Picking the next VCPU to run on a given CPU, Credit looks, in this order, for an *under* VCPU from the local running queue, an *over* VCPU from the local running queue, or an *under* VCPU from the running queue of a remote CPU, in a work-stealing inspired fashion.

Friendly Virtual Machines (FVM). 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 neither overused nor underused. Fairness is in the sense that each VM gets a proportional share of the bottleneck resource. Each VM is responsible for adjusting its demand of the underlying resources, resulting in a distributed adaptation system.

The adaptation strategy is done using feedback control rules such as Additive-Increase/Multiplicative-Decrease (AIMD), typically used in network congestion avoidance [Chiu and Jain, 1989], driven by a single control signal - the *Virtual Clock Time* (VCT), to detect overload situations. 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 bottleneck 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 thus, 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.

ASMan. The Adaptive Scheduling Manager (ASMan) [Weng et al., 2011] is an extension to Xen's scheduler. It adds the capacity to co-schedule virtual CPUs (VCPU) of VMs where there are threads holding a blocking synchronization mechanisms, such as spin locks. In non-virtualized systems, threads holding spin locks are not preempted. In a virtualized system, the VCPU continues to be hold by the thread but, because the hypervisor sees the VCPU as being idle, the VCPU is taken from execution and placed

on the waiting queue. Using the concept of VCPU related degree (VCRD), the ASMan system determines the degree of relationship between the VCPUs in a VM. The system dynamically determines this metric by monitoring, in each guest OS, the time spent in spin locks. The VM is then classified with a low or high VCRD if it is bellow or above a certain threshold. When the VCRD is high, the VCPUs of that VM are co-scheduled.

HPC computing. Shao et al. [Shao et al., 2009] use runtime information collected by a monitor running inside each guest's operating system to adapt the VCPU mapping of Xen [Barham et al., 2003]. They target high performance computing applications, and adjust the number of VCPUs to meet the real needs of each guest running these types of workloads. Decisions are made based on two metrics: the average VCPU utilization rate and the *parallel level*. The *parallel level* mainly depends on the length of each VCPU's run queue. The adaptation process uses an addictive increase and subtractive decrease (AISD) strategy. Shao et al. focus their work on benchmarks used to represent the common operations of high performance computing applications.

Auto Control. The Auto Control system [Padala et al., 2009] 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 caps for CPU and disk I/O. For each application, there is an application controller which collects the application's performance metrics (e.g. application throughput or average response time) and, based on the application's performance target, determines the new requested allocation. Because computational systems are non-linear, the model is adjusted automatically, aiming to adapt to different operating points and workloads.

Based on each application's 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. PRedictive Elastic ReSource Scaling for cloud systems (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 similar pattern (i.e., a signature) in the resource usage history. If a signature is not found, PRESS uses a discrete-time Markov chain. This technique allows PRESS to calculate how the system should change the resource allocation policy, by transitioning to the highest probability state, given the current state. The authors focus on CPU usage [Gong et al., 2010]. Thus, the prediction scheme is used to set the CPU cap of the target VM. The evaluation was made based on a synthetic workload applied to the RUBiS benchmark, built from observations of two real world workloads.[5]

Overbooking and Consolidation. In [Heo et al., 2009], Heo et al. use a feedback control mechanisms to dynamically allocate memory in an environment where multiple virtual machines share the same host. They show that allocating memory in such an overcommitted environment without taking also into account the CPU, results in significant service level violations. Their sensors measure memory allocation and usage and also application performance (i.e., response time of an Apache Web server). Memory allocations are collected from the balloon driver along with page fault rates from the `/proc` file system.

Ginko. Ginko [Hinesa 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 the 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's 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).

[5] `http://rubis.ow2.org/`, visited July 2, 2014

VMMB. In [Min et al., 2012], Min et al. presents VMMB, a Virtual Machine Memory Balancer for Unmodified Operating Systems. It uses the LRU histogram to estimate memory demand to periodically re-balance the memory allocated to each VM. Their algorithm determines the memory allocation size of each VM while it strives to globally minimize the page miss ratio. Their sensors are looking at nested page faults and to guest swapping, using a pseudo swap device for monitoring. They act of the balloon driver to enforce each VM's new memory size. When the balloon cannot collect enough memory, VMMB uses a VMM-level swapping to select a set of victim pages and immediately allocate memory to a beneficiary VM.

Difference engine. Gupta et al. [Gupta et al., 2008] is an extension to the Xen VMM which supports sub-page sharing using a novel approach that builds patches to pages by using the difference relative to a reference page. Similar pages are identified by comparing hash values of randomly selected parts of different pages. In-memory compression of infrequently accessed pages is made using multiple algorithms.

2.5.1.A Overall systems analysis

Table 2.9 summarizes the systems analyzed in this section. After the system name, the second column identifies the dominant resource, that is, the resource over which the system is monitoring but also acting. From the third to the fourth column, we present the techniques used in each of the adaptation phases. The last column allows us to quickly determine if the system proposes extensions to the code base of the VM or not.

Figure 2.10 depicts the overall RCI of each system that uses or augments a system-level VM. It presents a visual, quantitative and comparative analysis, which completes Table 2.9. Overall, systems tend to favor responsiveness design options (as this metric prevails in every system).

When looking for memory-dominant systems (Difference engine, VMMB, Overbooking, Ginko) we see that Overbooking, by trying to embrace a large number of sensors and actuators, is less responsive. In the CPU-dominated systems, HPC is the one classified as the most responsiveness but uses simply techniques (low intricateness) and a minimum number of sensors and actuators. ASMan is more intricate, basically because it needs extensions for the monitoring and action phase, but it had to give up on some responsiveness. Overall, these observations are in line with our initial *RCI*

Table 2.9: Sys-VM Systems

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

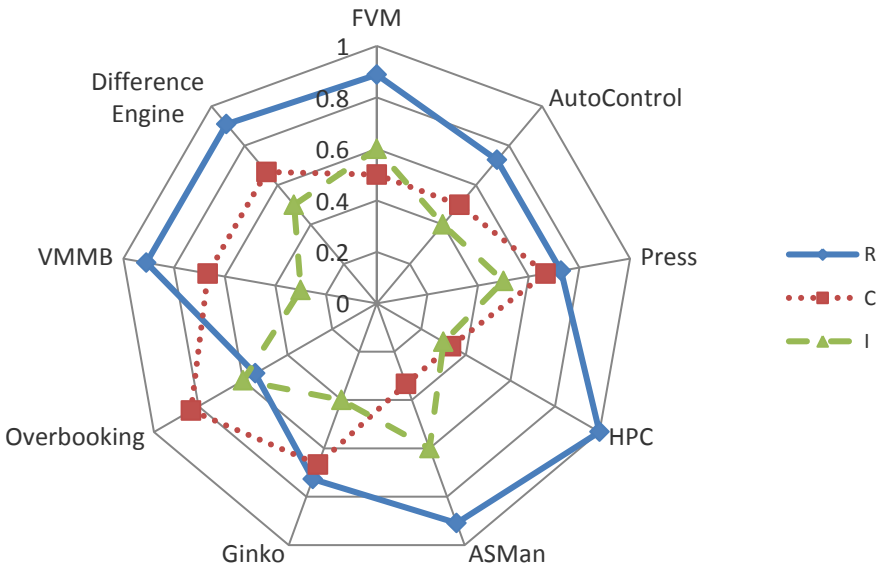


Figure 2.10: RCI of Sys-VMs

conjecture.

2.5.2 High-Level Language Virtual Machines

This section will present and discuss different systems that monitor resource usage, resulting in either imposing limitations or changing the policies of the JIT, GC, or resource manager sub-systems. Adaptation in high-language virtual machines is made by changing their building block parameters (e.g. JIT level of optimization, GC heap size) or the actual algorithm used to perform certain operations. This section starts by presenting classic work on Java Virtual Machines (JVMs) whose goal was to incorporate resource usage constraints on regular VMs. It then surveys more recent systems where the focus was to diminish the impact of GC in program execution. At the end of the section, Table 2.10 summarizes the techniques used in each system. As in the case of system-level VMs, this process is the implementation of step 1 in Figure 2.7, which is the base for determining each system RCI.

KaffeOS. Built on top of Kaffe virtual machine [Back and Hsieh, 2005], 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 for memory, KaffeOS uses a hierarchical structure where each process is assigned a hard and a soft limit. Hard limits relate to reserved memory. Soft limits acts as guard limit not assuring that the process can effectively use that memory. Children tasks can have, globally, a soft limit bigger than their parent but only some of them will be able to reach that limit.

JRES. The work of Czajkowski et al. [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 some limits and when new threads are created. The only supported resources 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 thread creation process, placing the native thread identification in a global registry. Then, at regular intervals, the registry is traversed and native calls are used to ask the operating system for the time spent in each thread. Byte code rewriting is also used to know how much memory is used by objects allocated by each thread.

Multitask Virtual Machine (MVM). The MVM [Czajkowski et al., 2005a] extends the Sun Hotspot JVM to support *isolates* and resource management. *Isolates* are similar to processes in KaffeOS. The distinguishing difference of MVM is in its generic Resource Management (RM) API, which uses three abstractions: resource attributes, resource domain, and dispenser. Each resource is characterized by a set of attributes (e.g. memory granularity of consumption, reservable, disposable). In [Czajkowski et al., 2005a] the MVM is able to manage the number of open sockets, the amount of data sent over the network, the CPU usage and heap memory size. When the code running on an isolate wants to consume a resource, it will use a library (e.g. send data to the network) or runtime service (e.g. memory allocation). In these places, the resource domain to which the isolate is bound will be retrieved. Then, a call to the dispenser of the resource is made, which will interrogate all registered user-defined policies to know if

the operation can continue. A dispenser controls the quantity of a resource available to resource domains. 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.

Isla Vista. 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. This heap sizing behavior is inspired by the exponential backoff model for TCP congestion control, where transmission rate relates to heap size, and packet loss relates to page faults.

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 its heap size, for a given amount of physical memory. Their approach allows for the heaps of multiple applications to remain small enough to avoid the negative impacts of paging, while still taking advantage of memory that is available within the system. They have modified the slow path of the GC (i.e., the code path that can result from tracing alive objects) to also take into account two conditions: *i)* if the resident set has decreased or, *ii)* if the number of page faults has increased. If any of these conditions is true, a GC will be triggered. They call this situation a *resource-driven* garbage collection.

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 microeconomics - 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 application-specific behavior.

CRAMM. In CRAMM [Yang et al., 2006] the heap size is dynamically adjusted to improve application performance. The resizing process is driven by the effective working set size (WSS) of the application. This is defined as the smallest main memory allocation for which page faulting degrades process throughput by less than $t\%$. To determine this, CRAMM proposes to extend the virtual memory manager of the operating system so that the WSS is dynamically built as the application progresses, monitoring minor and major page faults. After each heap collection, the GC requests a WSS estimative to the virtual memory manager. It then considers this value to resize the heap. After each GC run, the histogram is also reset since the new heap size will produce a new reference histogram pattern.

Control Theory. Heap sizing was also researched as a control theory problem[White et al., 2013]. In White's et al. work, a PID controller is used where the control variable is the heap resize ratio and the measurement variable is the GC overhead. To determine the new heap size, the controller, after each collection cycle, measures the error between the current GC overhead and the target GC overhead, specified by the user. The goal is to achieve and maintain the user-defined target GC overhead. The controller's parameters, such as the gain and the oscillatory period, were manually fine-tuned for a set of benchmarks. They have only tested their system under a full-heap collector.

Machine Learning for Memory Management. Machine learning techniques have been used to dynamically learn which is the best moment to garbage collect [Andreasson et al., 2002] and to choose, a-priori, the best GC configuration (algorithm, serial, parallel) [Singer et al., 2007, 2010] given an profile run of the application. In the first case, a reinforcement learning algorithm is used. A binary action is to be taken in each step leading to the decision to run the GC or not. The reinforcement learning algorithm accumulates penalties based on its decisions and, as time passes, it *learns* which are the best situations to run the GC. In the second group of papers, an offline machine learning algorithm, based on decision trees, is used to generate a classifier that, given a profile run

of a *new* program (i.e., not used to build the model), can predict a GC algorithm that minimizes the execution time.

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, in order to decide when to dynamically switch, and which GC it should switch to. The modified runtime incorporates all the available GC algorithms into a single VM image. At load time, all possible virtual memory resources are reserved. The layout of each space (i.e., nursery, Mark-Sweep, High Semispace, Low Semispace) is designed to avoid a full garbage collection for as many different switches as possible. For example, a switch from Semi-Space to Generational Semi-Space determines that the allocation site will be done at a nursery space, but the two half-spaces are shared. Switching can be triggered by points statically determined by previous profiling application execution, or by dynamically evaluating the GC load versus the application's threads. If the load is high, they switch from a Semi-Space (which performs better when more memory is available) to a Generational Mark-Sweep collector (which performs better when memory is more constrained).

2.5.2.A Overall systems analysis

Table 2.10 summarizes the system analyzed in this section. The majority of them are focused on the management of the heap size and use simple heuristics to guide this process. Exception are the ones using a PID controller [White et al., 2013] and a machine learning algorithm. However, these two systems either have to be fine-tuned manually or impose limitations on the type of garbage collector. Only one work takes into account the collocation of VMs and the need to transfer memory between them [Hertz et al., 2011]. Even so, it is focused on the individual performance of each instance and not the distribution of memory based on the progress of each workload.

Figure 2.11 depicts the overall RCI of each system that augments a high-level language VM, complementing the analysis of Table 2.10. As in the case of system-level VMs, systems have design options that favor responsiveness. The system taking into account the elasticity curve of micro-economics has the highest level of responsiveness perhaps because of its low overall intricateness of sensors, decision process, and actuators. We also see that the extra intricateness of the decision phase in "Control" and

Table 2.10: HLL-VM Systems

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

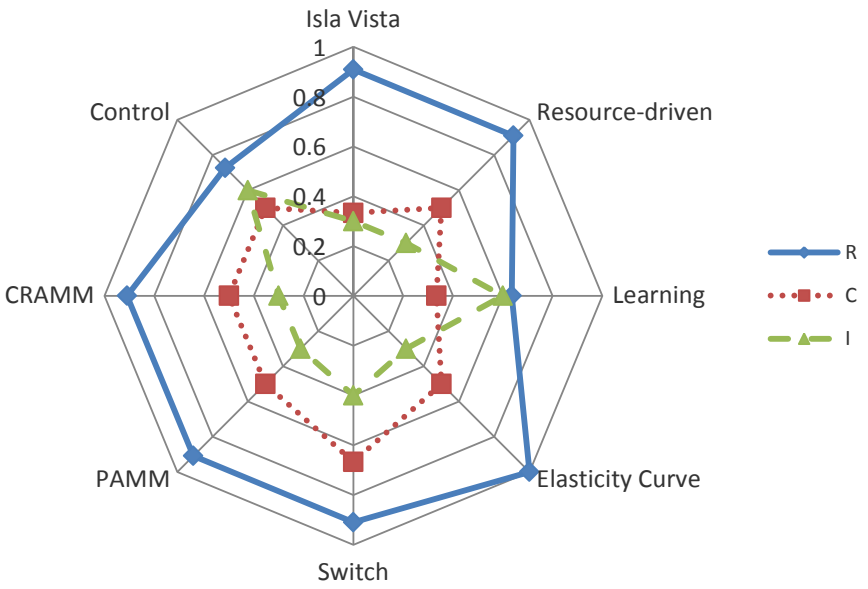


Figure 2.11: RCI of HLL-VMs

"Learning" had a cost. In the first case, it was the overall responsiveness, while in the second the system had to be designed with a smaller number of sensors, reducing comprehensiveness. Further research is needed to determine if other unexplored techniques in these two fields can bring more advantage.

2.6 Summary

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

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

Part II

Allocation and Scheduling in Platform-as-a-Service

3 Architecture of a Cloud-enabled JVM

Contents

Chapter overview

This chapter starts by discussing the overall architecture of QoE-JVM, its requirements, along with the most relevant building blocks that must exist in the execution environment to support them [Simão et al., 2011; Simão and Veiga, 2012, 2013a]. Next, we

present an economics-inspired model to drive adaptability in environments where resources are shared by multiple tenants. Our adaptability model is used to determine from which tenants resource scarcity will hurt performance the least, putting resources where they can do the most good to applications and the cloud infrastructure provider. We then close this chapter with a more detailed view on how this model can be applied, discussing different ways to monitor progress and what are the most relevant resources to be acted upon.

3.1 Introduction

Grid Computing flourished to a great extent due to its widespread adoption in many e-Science domains. Grids ease resource sharing, pooling and are amenable to both simple as well as sophisticated scheduling approaches. Currently, there is an analogous ongoing trend, this time to encompass new and existing Grid infrastructures into private, hybrid and federated clouds for e-Science. Clouds inherit the potential for resource sharing and pooling due to their inherent multi-tenancy support. In Grids, resource allocation and scheduling can be performed online, albeit mostly based on initially predefined and static job requirements. In contrast, in Clouds, resource allocation can also be changed elastically (up or down), at runtime, in order to meet the application effective needs at each time, improving flexibility and resource usage.

Public cloud infrastructures and supporting middleware for private/hybrid clouds (e.g., eucalyptus, open-stack) offer APIs to allow explicit allocation and deallocation of instances. Predominantly, this is done using system VMs that run full-fledged guest OS instances and applications in each instance. Deciding and dealing with this programming is cumbersome for e-scientists that are required to invoke cloud API besides writing their own code. This can be partially mitigated in the relevant, yet specific, case of Bag-of-Tasks applications, where multiple tasks can be successively assigned to a given VM, while the number of active VMs is managed automatically by the middleware [Silva et al., 2011].

When allocation needs to be changed, and resources are scarce, determining from which tenants resources must be taken to impact performance the least is a non-trivial and often deemed intractable problem, when outside the realm of batch scheduling with full prior information on resource requirements for each task, job, or VM instance. Other works have addressed resource allocation in a shared or multi-tenant environ-

ment [Salomie et al., 2013; Chen et al., 2014; Hertz et al., 2011; Hinesa et al., 2011]. Nevertheless they lack the notion of *resource effectiveness* in the sense that when there are scarce resources, there is no attempt to determine when to take such resources from applications (i.e., either isolation domains or the whole VM) so that they hurt performance the least.

Managed languages (e.g., Java, C#) are becoming increasingly relevant in the development of large scale solutions, leveraging the benefits of a virtual execution environment (VEE) to provide secure, manageable and component-oriented solutions. Relevant examples include work done in various areas such as web application hosting, large scale data processing [Castro Fernandez et al., 2013], enterprise services, supply-chain platforms, implementation of functionality in service-oriented architectures. The field of e-Science also shows an increasing interest in Java for physics simulation, economics and statistics, network simulation, chemistry, computational biology and bio-informatics, showing that to some extend, high performance and high throughput computing have also been ported to managed languages [Hiden et al., 2013; Krampis et al., 2012; Pierre and Stratan, 2012; Holland et al., 2008; Gront and Kolinski, 2008].

To extend the benefits of a local VEE and allow a mostly transparent horizontal scaling, several solutions have been proposed to federate Java virtual machines [Kächele and Hauck, 2013; Zhang et al., 2008; Zhu et al., 2002], aiming to provide a single system image where the managed application can benefit from the global resources of the cluster. If this system image has elasticity, in the sense that resources are made available proportionally to the effective need, and if these resources are accounted/charged as they are used, we can provide a high-level language virtual machine (HLL-VM) across the cluster, as an utility. If these changes are made dynamically (instead of explicitly by their users), we will have an adaptive and resource-aware virtual machine, that can be offered as a value-added Platform-as-a-Service (PaaS).

3.2 Related work

3.2.1 Resource accounting in High-Level Virtual Machines

Resource monitoring and management are required in HLL-VMs for Cloud environments due to two major reasons: i) monitoring is required to obtain some kind of measurement of resource usage by an application, and ii) management is required in order to

determine the amount of resources should be awarded to an application, and to enforce those limits somehow. Such mechanisms, regarding low-level aspects of an execution environment, may need to be continuously or at least frequently activated, enforced or inquired. Therefore, their implementation must aim at minimizing the impact to the overall application's performance.

Monitoring low-level aspects of a computer system regarding the execution of a given application must be done with low impact in the overall application's performance. Sweeney et al. [Sweeney et al., 2004] aims to accomplish these goals using hardware performance counters. They extended the Jikes RVM with a performance monitor layer, interacting with a native C library. Although relevant, in fact, they do not support any kind of restriction on resource consumption. Regarding implementation, they rely on a previous version of Jikes RVM and its N-M thread mapping (where there were N VM threads mapped by the runtime to M native threads), while the current version already uses a 1-1 mapping to native threads.

For runtime mechanisms, Price et al. [Price et al., 2003] describe a method for modifying the garbage collector to measure the amount of live memory reachable from each group of threads. Their implementation is based on an old version of Jikes RVM but the algorithms proposed are interesting in the context of our system, and have the potential to be further extended (i.e., the work presented in [Price et al., 2003] does not support tracing collectors). They give some usage scenarios for the information accounted, but leave as an open issue the building of a policy driven framework.

Some high-level virtual machines have been augmented or designed from scratch to integrate resource accounting [Czajkowski and von Eicken, 1998; Suri et al., 2001; Back et al., 2000; Czajkowski et al., 2005a]. The Multitask Virtual Machine (MVM) [Czajkowski et al., 2005a] is based on the Hotspot virtual machine. It supports isolated computations, akin to address spaces, to be made in the same instance of the VM. This abstraction is called *isolate*. Another distinguishing characteristic is the capacity to impose constraints regarding consumption of *isolates*. MVM resource management work is related to the Java Specification Request 284 [Grzegorz Czajkowski, 2009] but MVM uses services only available on Solaris operating system, which runs on top of SPARC's hardware. Porting the open-source code to a new platform is an open issue.

The work in [Suri et al., 2001] and [Back et al., 2000] enables precise memory and CPU accounting. Nevertheless, they do not provide an integrated interface to deter-

mine the resource consumption policy, which may involve VM, system or class library resources. Another approach is to exchange low level precision and additional overhead for the sake of portability. Binder's profiling framework [Binder et al., 2009] statically instruments the core runtime libraries, and dynamically instruments the rest of the code. The instrumented code periodically calls pure Java agents to process and collect profiling information.

In [Czajkowski et al., 2005b], Czajkowski et al. propose to enhance the resource management API of the MVM [Czajkowski et al., 2005a], forming a cluster of this VMs where there are local and global resources that can be monitored and constrained. However, Czajkowski's work lacks the capacity to determine the effectiveness of resource allocation, relying on predefined allocations. In [Janik and Zielinski, 2010], a reconfigurable monitoring system is presented. This system uses the concept of Adaptable Aspect-Oriented Programming (AAOP) in which monitored aspects can be activated and deactivated based on a management strategy. The management strategy is, in practice, a policy which determines the resource management constraints that must be activated or removed during the application's lifetime.

Ginko [Hinesa et al., 2011] is an application-driven memory overcommitment framework which allows cloud providers to run more system VMs with the same memory. The system works on a scenario where the virtualization stack is made of system-level VMs with some running a JVM that supports the execution of an application server. Ginkgo uses a JVM-level memory balloon that can reclaim memory more quickly when necessary. For each VM, Ginkgo collects samples of the application performance, memory usage, and submitted load. Then, in the production phase, instead of assigning the same amount of memory to 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.

In [Singer et al., 2011], the GC is auto-tuned in order to improve the performance of a MapReduce Java implementation for multi-core hardware. For each relevant benchmark, machine learning techniques are used to find the best execution time for each combination of input size, heap size and number of threads in relation to a given GC algorithm (i.e., serial, parallel or concurrent). Their goal is to make a good decision about a GC policy when a new MapReduce application arrives. The decision is made locally to an instance of the JVM. The experiments we presented are also related to memory management, but our definition of QoE (as will be presented in Section 3.4) can go

beyond this resource.

At the middleware level, Coulson et al. [Coulson et al., 2008] present OpenCom, a component model oriented to the design and implementation of reconfigurable low-level systems software. OpenCom's architecture is divided between the kernel and the extensions layers. While the kernel is a static layer, capable of performing basic operations (i.e., component loading and binding), the extensions layer is a dynamic set of components tailored to the target environment. These extensions can be reconfigured at runtime to, for example, adapt the execution environment to the application's resource usage requisites. Our work handles mechanisms at a lower level of abstraction.

Duran et al. [Duran-Limon et al., 2011] use a thin virtual machine to virtualize CPU and network bandwidth. Their goal is to provide an environment for resource management, that is, resource allocation or adaptation. Applications targeting this framework use a special purpose programming interface to specify reservations and adaptation strategies. When compared to more heavyweight approaches like system VMs, this lightweight framework can adapt more efficiently for I/O intensive applications. The approach taken in Duran's work binds the application to a given interface for resource adaptation. Although in our system the application (or the libraries they use) can also impose their own restrictions, the adaptation process is mainly driven by the underlying virtual machine without direct intervention of the applications.

3.2.2 Measuring progress

Because measuring application progress is an important step in any adaptation process, there are several contributions on this topic, ranging from low,level system information, such as performance counters, to information about the progress of specific variables inside applications.

Performance counters have been used to analyze object oriented applications [Sweeney et al., 2004; Hauswirth et al., 2010]. Nevertheless, these works do not attempt to adapt the behavior of the application or the high-level VM, as they focus only on the study of different workloads to better understand how major runtime components behave. The utilization of performance counters in full virtualized systems (using system level VMs) have similar problems because applications running in a guest VM will see information about the hypervisor instructions as their own [Du et al., 2011].

In Chapter 2 we have surveyed several sensors used in high-level language VMs as input of decision control modules. These systems aim at minimizing execution time or memory usage. Most monitor how applications running in high-level virtual machines are using memory and, indirectly, how they are making progress [Andreasson et al., 2002; Bobroff et al., 2014]. Examples include memory allocation rate, which relates the heap in use and the memory freed at two distinct time intervals, or the processor time spent on executing instructions of the running program (and not related to GC tasks). While the last example seems to be useful across a wide range of applications, the first one is less effective for scientific applications that allocate most of their working space before starting the intensive computation phase.

In [Hoffmann et al., 2010], an application programming interface (API) is proposed to enable applications reporting progress through heartbeats. PowerDial [Hoffmann et al., 2011] monitors the performance of applications using the Heartbeat framework. The system can dynamically adapt the application's configuration (e.g. parameters given in the command line) in response to changes of load or power, threatening the ability to deliver results in effective time. In these cases, results will eventually be delivered with less accuracy. In our work, we want the progress information to be used transparently to adapt the application execution runtime, restricting or giving more resources, without depending on the application parameters.

Task-driven workloads, typical in Grid infrastructures, must also be monitored by the execution runtime to adapt the relevant system parameters and achieve the desired goals (e.g. improve performance, save energy). Cushing et al. [Cushing et al., 2011] propose a prediction-based framework to automatically scale the number of tasks running in scientific workflow management systems. The prediction of the number of tasks is based on the size of the input queues of each task and the data processing rate.

At a higher level of abstraction, Silva et al. [Silva et al., 2008, 2011] focus on choosing the best number of hosts to run Bag-of-Tasks workloads, in an attempt to find a trade-off between performance and cost effectiveness (regarding the host renting time). Their heuristics are based on the tasks execution time. These approaches not only require more expertise to organize programs but they are also sensitive to long running workloads, where finishing time among different tasks (or length of the input queue and the size of each element) can have large variations. Unlike Grid infrastructures, Cloud infrastructures depend on virtual machines to provide the two basic service models, either IaaS or PaaS. In [Mc Evoy and Schulze, 2011], Mc Evoy et al. discuss implications of

scheduling work in such environments showing the importance of knowing more about the workloads' profile so that the execution environment can be adapted to provide improved performance.

3.2.3 Checkpointing, restoring and migration mechanisms

Checkpoint and restore mechanisms have been developed at different levels of the execution stack, namely, the hypervisor (to operate on virtual machines), operating system (to operate on native application), and high-level language VMs (to operate on manage application). At each of these levels, the nature of the artifacts considered for checkpointing and restoring is different. These artifacts can either be classified as *internal* or *external* to the target of the checkpoint. Internal state includes all the artifacts that have dependencies with the execution container (which can be the hypervisor, operative system or high-level language VMs). External state represents guest's specific data structures.

At the hypervisor it is necessary to keep the memory pages (and related information) used by the virtual machine supporting the execution of an operating system and its application, along with the registers of the virtual CPUs in execution [Lagar-Cavilla et al., 2011]. At the operating system level a general solution would have to keep the virtual address space of the process (heap, stack, and any mapped region), the signal handlers and registers [Hargrove and Duell, 2006]. Finally, in high-level language VMs, what is necessary to be persisted is the application specific state and the stack frames of the currently executing threads.

Common to these three levels of intervention is the requirement to keep the external state of the application, such as file contents or socket connections. While the former issue is typically solved using a shared file system, the latter is usually not a problem in e-science scenarios [Lagar-Cavilla et al., 2011]. It is, however, a research topic in contexts related to the migration of multimedia applications [Velazquez-Garcia et al., 2013].

Our architecture design, presented in Section 3.3, includes a mechanism which can checkpoint and migrate a given application when running co-located with others in a consolidated environment. To do so, hypervisor-level techniques are too coarse-grained, as they would checkpoint (and eventually migrate) all the applications inside a given virtual machine. Alternatively, it would require us to deploy each application in an individual HLL-VM running on top of a single system-level VM. With the widely adopted

approach of HLL-VMs requesting the services of an operating system, this would demand the provisioning of a large amount of physical resources. However, research efforts to run a HLL-VM directly on top of an hypervisor are being made. [1,2]

Application-level checkpoint support at the operating system level is a viable alternative. Nonetheless, this is not done without the introduction of extra overheads. These overheads exist because of extensions to kernel modules [Hargrove and Duell, 2006] or additional levels of virtualization [Osman et al., 2002] which would lay between the OS and the managed runtime.

As our work targets checkpoint at the HLL-VM level, we focus our discussion of related work on this abstraction level. A possible division, when analyzing these works, is to consider those that require modifications to the HLL-VM and those only rely on source code or bytecode instrumentation. A similar classification was used when discussing the resource accounting frameworks and mechanisms in Section 3.2.1.

Regarding the first approach, an HLL-VM is augmented to be able to export and import the state of individual threads. A significant body of work followed this design: JavaThread [Bouchenak et al., 2004], Jessica2 [Lam et al., 2010; Zhu et al., 2002], MobileJikesRVM [Quitadamo and Leonardi, 2008], Nomads [Suri et al., 2000]. Regarding the second approach (i.e., instrumentation) it takes a method in bytecode an produces a new one that can capture and restore the execution state (local variable and operands) [Ma et al., 2002; Sakamoto et al., 2000; Ferreira et al., 2003]. OBIWAN [Ferreira et al., 2003] is a middleware for the migration of mobile agents, and it can deal with multiple threads. This solution manipulates Java source code in order to add additional instructions to support migration. This has two main disadvantages: it does not support applications whose Java source code is not provided and, needs the assistance of the programmer to annotate where checkpointing can be performed and which data to be included in it.

In summary, few works consider the checkpoint/migration of the complete application and thus, lack the support for obtaining the execution state of blocked threads. They also lack support for the transparent migration of file contents. Finally, none of the existent solutions support the checkpointing of the application's specific state along with the execution of the application.

[1] `https://kenai.com/projects/guestvm`, visited August 27, 2014

[2] `http://www.virtualizationpractice.com/virtualize-java-without-an-operating-system-5639/`, visited August 27, 2014

3.3 Architecture Overview

In a Cloud-like environment, when several instances of a high-level virtual machine (or manage runtime) are running, each will have a set of resources allocated to the execution of a potential distinct application. Target applications of this thesis have typically a long execution time and can dynamically spawn several execution flows to parallelize their work. This is common in the field of science supported by informatics like economics and statistics, computational biology and network protocols simulation [Pierre and Stratan, 2012; Halappanavar et al., 2012; Gront and Kolinski, 2008]. Given this heterogeneity of workloads, each managed runtime will have a different ratio between the resources allocated and the progress the application is effectively doing. In general, we name this ratio the *quality-of-execution* or simply QoE [Simão et al., 2011; Simão and Veiga, 2012]. In our architecture, each instance of a Java VM with extended services, so that the QoE can be adjusted, we call it *QoE-JVM*.

QoE can be inferred coarsely from the application's execution time for medium running applications, request execution times for more service driven ones such as those web-based, or from critical situations such as thrashing or starvation. Also, it can be derived in a more fine-grained way from incremental indicators of application progress, such as the amount of processed input, disk and network output generated, execution phase detection, or memory pages updates.

QoE can be used to drive a VM economics model, where the goal is to incrementally obtain gains in QoE for VMs running applications requiring more resources or for more privileged tenants. This, while balancing the relative resource savings drawn from other tentants' VMs and considering the perceived performance degradation. To achieve this goal, certain applications will be positively discriminated, reconfiguring the mechanisms and algorithms that support their execution environment (or even engaging available alternatives to these mechanisms/algorithms). For other applications, resources must be restricted, imposing limits to their consumption, regardless of some performance penalties (that should also be mitigated). In any case, these changes should be operationally transparent to the developer and especially to the application's user. Section 3.4 will delve into more details.

Figure 3.1 presents the overall architecture of our distributed *platform as a service* for Cloud environments. QoE-JVM is supported by several runtime instances, eventually distributed by several computational nodes, each one cooperating to the sharing of re-

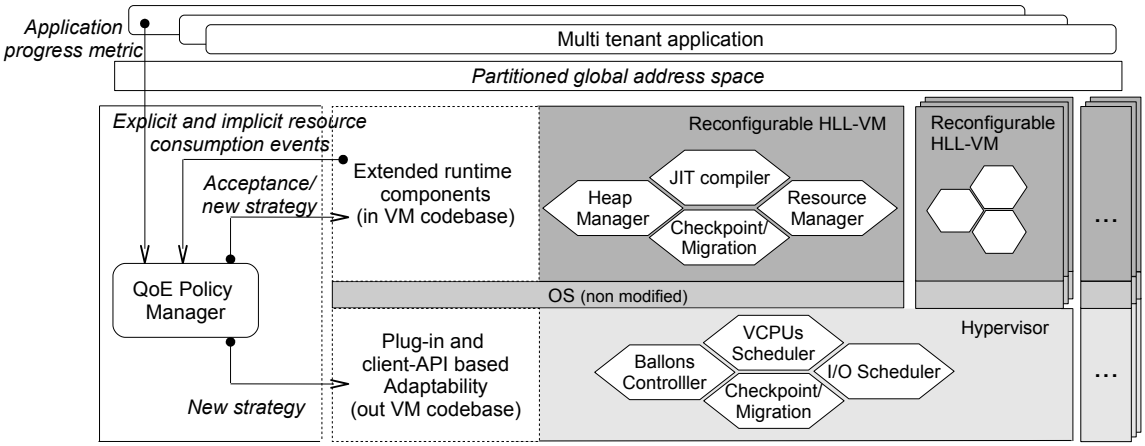


Figure 3.1: Overall architecture

sources. For an effective resource sharing, a coordinated mechanism must be in place to make weak (e.g. change parameters) or strong (e.g. change GC algorithm, migrate running application) adaptations [Salehie and Tahvildari, 2009]. QoE-JVM encompasses a distributed shared objects middleware, a reconfigurable high-level language virtual machine (HLL-VM), and, at the bottom, the available reconfiguration mechanisms of system level virtual machine (Sys-VM). In this architecture, the operating system (OS) services are only used, not extended. The following sections will present details of the these building blocks.

3.3.1 Resource Awareness and Control

The Resource Aware virtual machine is the underlying component of the proposed infrastructure. It has two main characteristics: *i)* resource usage monitoring, and *ii)* resource usage restriction or limitation. Current virtual machines for managed languages can already report about several aspects of their internal components, like used memory, number of threads, classes loaded, such as the JVM Tools Interface. [3] However, they do not enforce limits on the resources consumed by their single node applications. In a cluster of competing 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 existing resources must be done

[3] `http://docs.oracle.com/javase/7/docs/technotes/guides/jvmti/`, visited July 1, 2014

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

Controlling resource usage inside an HLL-VM can be carried out by either *i)* intercepting calls to the classes that constitute the platform's library and virtualize access to system resources or, *ii)* changing parameters or algorithms of internal components of the virtual machine. Examples of the first hook type include classes of the base class library, such as `java.util.Socket` (where the number of bytes sent and received per unit of time can be controlled) and `java.util.concurrent.ThreadPoolExecutor` (where parameters control the minimum and maximum number of threads available at each pool).

Regarding the second hook type, a prominent example is the memory management subsystem (i.e., garbage collection algorithm, heap size). Because we are dealing with applications where memory is automatically managed by the runtime, this second type of hook has the potential for a significant impact on application performance. More than determining the ideal moment to collect [Zhang et al., 2006; Hertz et al., 2011] or using intricate mechanisms to minimize page misses during collection [Hertz et al., 2005], the system should employ heap sizing policies that favor an effective use of resources [Libič et al., 2014; White et al., 2013; Singer and Jones, 2011].

3.3.2 Accurate Progress Monitoring

We believe that resource scheduling should be done in a mostly transparent way. This is so because target developers (i.e., scientists using Java applications or writing their own using existing Java frameworks) will most likely be skeptic about the perspective of learning new programming interfaces to transfer their applications or frameworks. In order to comply with this idea, we need, not only to monitor resource consumption, but also to determine application progress.

There are different ways to measure the progress of an application. Progress metrics can be either collected from sensors outside or inside the application. In the first case, progress metrics are collected from the execution environment, which includes, hard-

ware, operating system and managed runtimes. In the second case the, application is built from the source with the capability to report its progress, or is instrumented at load time and enhanced to do so. To be useful, the instrumentation process must be guided by hints about relevant flows (e.g. annotations or configuration files).

3.3.3 Checkpointing and Migration of the Execution State

The counterpart of resource containment is when an application needs more resources but the node where it is running is highly loaded, or resource-exhausted. If the application is allowed some elasticity in the used resources, they should be made available.

This migration should be done without the need for an application restart. To this end, the resource aware VM includes a mechanism for checkpointing and migration, which enables the whole application to migrate to another node, where another resource-aware VM is running with the necessary amount of resources. The migration is performed without restarting the application, avoiding losing all the work previously done. This is particularly useful for applications with long execution times, as in various fields related with e-Science (mostly in the context of Grid and Cloud computing) where managed languages are becoming dominant, including chemistry, computational biology and bio-informatics [Hiden et al., 2013; Pierre and Stratan, 2012; Gront and Kolinski, 2008], with many available Java-based APIs (e.g., Neobio [4]).

The checkpointing component of the architecture represents the necessary extensions to the HLL-VM, so that fine-grained checkpointing, restoring and migration of applications is possible. Our checkpointing mechanism can also run concurrently with the main program, preventing full pause of the application during checkpointing, thus further reducing the overhead experienced by applications. It addresses checkpoint consistency and excessive resource consumption (i.e., CPU, memory). The activation of these mechanisms is regulated either by local rules, activated to provide a failure-tolerant environment, or by a *Quality of Execution* Controller (QoE Controller). In the last case, based on execution requirements (e.g., CPU, memory, and network usage), the QoE controller can apply two coarse-grained measures: i) checkpoint and suspension of a VM, ii) migration of the application execution state to another node.

[4] `http://neobio.sourceforge.net/`, visited August 6, 2014

3.3.4 Adaptability and the Policy Engine

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

The policy engine operates autonomously or in reaction to a given resource outage in the VMs. Autonomous behavior is governed by maintaining knowledge about the applications' previous execution, and adjusting the VMs and cluster parameters to achieve better performance for that specific application. Reactive operation is driven by declarative policies that determine the response to a resource outage. This response may result in a local adaptation (e.g. restrain the resources of another VM in the same node, or change the GC algorithm to consume less memory but eventually taking more time to execute), or have cluster-wide impact (e.g. migrate the entire application to a VM in another node).

Figure 3.2 presents an example of a declarative policy to be used by VM instances represented in Figure 3.1. Policies are organized in *resource attributes*, which identify adaptation targets, and *rules*, which determine monitoring, decision and actions to be made for each resource considered.

The example presented in Figure 3.2 defines limits for CPU usage, and the number of threads and sockets the application is allowed to use. CPU usage and threads are monitored and managed by specific rules but using a similar, reusable approach: i) CPU usage is monitored with a sliding window in order to filter irrelevant peaks, while ii) the number of active threads is also monitored with a sliding window in order to trigger rescheduling only when the limit is consistently exceeded. When CPU usage limit is reached the execution is suspended. In the case of reaching the maximum number of local threads, a cluster-wide thread distributor would be activated to be used in the creation of new threads.

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

Figure 3.2: Declarative policy example considering two types of resources

3.4 Driving Adaptability with Quality-of-Execution

This section starts by presenting how a simple and generic metric can be used to determine which runtime resource management strategy can be used for each workload in order to maximize its performance when faced with resource degradation. It then describes which kind of performance or progress metrics are relevant to be used. We finish by presenting the kind of resources that are relevant to be controlled, in order to have an elastic behavior, without breaking the intended semantics of application execution.

3.4.1 An economic-inspired model

In Economics, there are typically two major classes of variables that drive business performance or its *processing*: those related with: i) *Value* and its equivalent *output, revenue*, ii) those related with *input* and its associated *cost*. Depending on the specific kind of economic activity, revenue may be the value of sales of a shop or factory, or financial

gains in banking, stock market, etc. Costs may be associated with labour, resources, raw materials, energy, investment, capital expenditures, and so on.

Economists sometimes need to take into account non-direct monetary aspects, such as opportunity cost, risk, trade-offs in capital investment [Mankiw, 2011]. This leads to two inherent notions in Economics (apparent even to those uninitiated) that: i) an activity consumes/costs resources and creates value, and ii) faced with limited resources, these should be geared towards the activities that at a given moment provide more return, and should be taken from activities where they will harm their return the least, i.e., commit and transfer resources in order to achieve a global positive (or maximized) *yield* from the whole process.

In many shared or multi-tenant infrastructures, such as private clouds, there may be no money. However, even when there is a credit-based system, we are left with more impressive notions of *Progress* and *Resource Usage*. These are more easily comparable over time, and sometimes across applications and application classes. Resource usage is more easily established while still open to some debate. Memory, CPU, and storage are mostly obvious and should be accounted for. The notion of progress, while intuitive, is more elusive and dependent on application semantics (we address this in Section 3.4.3). For the moment, let us consider progress as units of work carried out by the application. Additionally, we can also measure how progress and resource usage vary, at what rate, and determine the *effectiveness* by relating both.

Regarding *yield*, we are mostly interested in determining how to identify two specific situations: i) when an application is making reduced progress due to resource shortage and could use more resources effectively, and ii) when an application is not taking full advantage of its resources and could make similar progress with fewer resources. Obviously, we want to transfer resources from applications in ii) (least effective first) to applications in i) (more hurt first).

This should be performed incrementally, based on the derivative of progress and resource consumption over time. Therefore, we are immune to different ways of measuring progress, resource usage, and profiles across applications. This tradeoff is illustrated in Figure 3.3, where we compare percentage variation of progress (%dP) against percentage variation of resource usage (%dR) and establish a ratio between the two , our *yield*. The measurement of this variable will determine the transfer of resources. As decisions also affect the system, we could restrict resources to take slots of 5% (of currently allo-

%dP \ %dR	50%	55%	60%	65%	70%	75%	80%	85%	90%	95%	100%	105%	110%	115%	120%	125%	130%	135%	140%	145%	150%
50%	1.00	1.10	1.20	1.30	1.40	1.50	1.60	1.70	1.80	1.90	2.00	2.10	2.20	2.30	2.40	2.50	2.60	2.70	2.80	2.90	3.00
55%	0.91	1.00	1.09	1.18	1.27	1.36	1.45	1.55	1.64	1.73	1.82	1.91	2.00	2.09	2.18	2.27	2.36	2.45	2.55	2.64	2.73
60%	0.83	0.92	1.00	1.08	1.17	1.25	1.33	1.42	1.50	1.58	1.67	1.75	1.83	1.92	2.00	2.08	2.17	2.25	2.33	2.42	2.50
65%	0.77	0.85	0.92	1.00	1.08	1.15	1.23	1.31	1.38	1.46	1.54	1.62	1.69	1.77	1.85	1.92	2.00	2.08	2.15	2.23	2.31
70%	0.71	0.79	0.86	0.93	1.00	1.07	1.14	1.21	1.29	1.36	1.43	1.50	1.57	1.64	1.71	1.79	1.86	1.93	2.00	2.07	2.14
75%	0.67	0.73	0.80	0.87	0.93	1.00	1.07	1.13	1.20	1.27	1.33	1.40	1.47	1.53	1.60	1.67	1.73	1.80	1.87	1.93	2.00
80%	0.63	0.69	0.75	0.81	0.88	0.94	1.00	1.06	1.13	1.19	1.25	1.31	1.38	1.44	1.50	1.56	1.63	1.69	1.75	1.81	1.88
85%	0.59	0.65	0.71	0.76	0.82	0.88	0.94	1.00	1.06	1.12	1.18	1.24	1.29	1.35	1.41	1.47	1.53	1.59	1.65	1.71	1.76
90%	0.56	0.61	0.67	0.72	0.78	0.83	0.89	0.94	1.00	1.06	1.11	1.17	1.22	1.28	1.33	1.39	1.44	1.50	1.56	1.61	1.67
95%	0.53	0.58	0.63	0.68	0.74	0.79	0.84	0.89	0.95	1.00	1.05	1.11	1.16	1.21	1.26	1.32	1.37	1.42	1.47	1.53	1.58
100%	0.50	0.55	0.60	0.65	0.70	0.75	0.80	0.85	0.90	0.95	1.00	1.05	1.10	1.15	1.20	1.25	1.30	1.35	1.40	1.45	1.50
105%	0.48	0.52	0.57	0.62	0.67	0.71	0.76	0.81	0.86	0.90	0.95	1.00	1.05	1.10	1.14	1.19	1.24	1.29	1.33	1.38	1.43
110%	0.45	0.50	0.55	0.59	0.64	0.68	0.73	0.77	0.82	0.86	0.91	0.95	1.00	1.05	1.09	1.14	1.18	1.23	1.27	1.32	1.36
115%	0.43	0.48	0.52	0.57	0.61	0.65	0.70	0.74	0.78	0.83	0.87	0.91	0.96	1.00	1.04	1.09	1.13	1.17	1.22	1.26	1.30
120%	0.42	0.46	0.50	0.54	0.58	0.63	0.67	0.71	0.75	0.79	0.83	0.88	0.92	0.96	1.00	1.04	1.08	1.13	1.17	1.21	1.25
125%	0.40	0.44	0.48	0.52	0.56	0.60	0.64	0.68	0.72	0.76	0.80	0.84	0.88	0.92	0.96	1.00	1.04	1.08	1.12	1.16	1.20
130%	0.38	0.42	0.46	0.50	0.54	0.58	0.62	0.65	0.69	0.73	0.77	0.81	0.85	0.88	0.92	0.96	1.00	1.04	1.08	1.12	1.15
135%	0.37	0.41	0.44	0.48	0.52	0.56	0.59	0.63	0.67	0.70	0.74	0.78	0.81	0.85	0.89	0.93	0.96	1.00	1.04	1.07	1.11
140%	0.36	0.39	0.43	0.46	0.50	0.54	0.57	0.61	0.64	0.68	0.71	0.75	0.79	0.82	0.86	0.89	0.93	0.96	1.00	1.04	1.07
145%	0.34	0.38	0.41	0.45	0.48	0.52	0.55	0.59	0.62	0.66	0.69	0.72	0.76	0.79	0.83	0.86	0.90	0.93	0.97	1.00	1.03
150%	0.33	0.37	0.40	0.43	0.47	0.50	0.53	0.57	0.60	0.63	0.67	0.70	0.73	0.77	0.80	0.83	0.87	0.90	0.93	0.97	1.00

Figure 3.3: Ratio of Progress versus Resource Allocation variation overview.

cated), possibly repeated while the application yield does not change significantly and is ranked high by the algorithm.

This simple model allows the scheduler to make decisions, over a given time frame, of where resources should primarily be diverted to and where from. It is very adaptable and flexible as it is driven by differential, incremental measurements to detect and determine effectiveness of resource usage and makes quick decisions on resource allocation transfer, restriction (in the extreme of application checkpointing or migration).

The upper triangular zone in Figure 3.3 illustrates applications that are using resources effectively, since the extra resources they allocated have resulted in a comparatively larger progress, while in the down triangular zone, application progress has not improved at the same rate as the influx of resources. Empirically, applications experience several phases where they are more or less eager for resources, and others where they stabilize, or could even drop some with little impact (e.g. caching data elements no longer accessed). The key point is to approximately identify when this happens and transfer resources, when they are scarce, accordingly.

The same happens in Economics, regardless of the type of business. No matter the business is more or less labor- or capital-intensive, yearly profit percentage increase reports make up a uniform measurement that allows decision-makers to know when markets are emergent (booming) or mature (saturated), thus able to reward better or worse the new investments on them, or are simply below opportunity cost. The same happens with stock market prices and their percentage variation (even though they may not represent results accurately but simply expectations). The next section will detail our resource redistribution model.

3.4.2 QoE-JVM Economics

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

We are partially inspired by the Cobb-Douglas [Cobb and Douglas, 1928] production function (henceforth referred to as equation) from Economics to motivate and to help characterize QoE. The Cobb-Douglas equation, presented in Equation 3.1, is used in Economics to represent the *production* of a certain good.

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

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

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

Alternatively, labour can be regarded, not as a variable representing a measure of human work employed, but as a result, representing the efficiency of the capital invested, given the production output achieved, i.e., labour as a multiplier of resources into production output. This is usually expressed by representing Equation 3.1 in terms of L, as in Equation 3.2. For simplicity, we have assumed the three extra elements to be equal

to one. First, the technological and civilization context does not apply, and since the data center economy is simpler, as there is a single kind of activity, computation, and not several, the relative weight of labour and capital is not relevant. Furthermore, we will be more interested in the variations (relative increments) of efficiency than on the efficiency values themselves, hence the simplification does not introduce error.

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

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

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

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

We assume a scenario where, when applications are executed in a constrained (overcommitted) environment, the infrastructure may remove m units of a given resource from a set of resources R (e.g. memory size, CPU cores, bandwidth) and give it to another application that can benefit from this transfer. Examples of transferable units are 50 MiBytes of heap size, 1 core, and 2 MiBytes/sec of bandwidth. This transfer may have a negative impact on the application that offers resources and it is expected to have a positive impact on the receiving application. To assess the effectiveness of the transfer,

the infrastructure must be able to measure the impact on the giver and receiver applications, namely somehow to measure the approximate savings in PCU, that is, the relation between employed resources and effective progress, as described next.

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

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

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

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

The definition of $Savings_r$ represents the savings of a given resource r when two allocation or management strategies are compared, s_a and s_b, as presented in Equation 3.6. Functions $U_r(s_a)$ and $U_r(s_b)$ relate the *usage* of resource r, given two different management strategies or allocation configuration, s_a and s_b. For example, if r represents memory, then U could be total bytes currently allocated. We allow only those reconfigurations which offer savings in resource usage to be considered in order to calculate yields.

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

Regarding *performance degradation*, it represents the impact of the savings, given a specific performance metric, as presented in Equation 3.7. Considering the time taken to

execute an application (or part of it), the performance degradation relates the execution time of the original configuration, $P(s_a)$, and the execution time after the resource allocation strategy has been modified, $P(s_b)$.

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

Each instance of the QoE-JVM continuously monitors 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. A fundamental aspect of this approach is to determine how much progress the application is doing, as explained next.

3.4.3 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. aligning 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 that analyze graphs of objects have a visited/processed objects set, which typically will encompass all objects when the algorithm terminates (or at least, a significant part of it). If the rate of objects processed can be determined, it will indicate how the application is making progress. Other examples would be applications to perform video encoding, where the number of frames processed is a measure of progress [Hoffmann et al., 2011].

There is a balance and trade-off in measuring progress, using a metric that is close to

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

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

progress. A similar indication will be given when the application is writing in new or previously unused (or unmodified) memory pages.

Although QoE-JVM could read low-level indicators as I/O storage and network activity or memory page activity, we have made experiences with two metrics to measure performance degradation as defined in Section 3.4.2, the **completion time** and the **frequency of calls** to annotated methods. Completion time is used to demonstrate the benefits of our system when using well known subsets of core application components, organized as benchmarks [Blackburn et al., 2006] for Java runtimes. These core components are representative of different types of workloads but tend to have a short execution time (more details in Chapter 5). When running the complete version of these applications, we use the frequency of calls during a window observation. Frequency of calls metric allows monitoring a small set of methods that have to be identified by the programmer. We rely on the programmer's knowledge about the domain model to identify a relevant set of methods, using language-specific constructions such as annotations. A similar rationale can be found in other fields of software engineering such as dynamic software updating [Pina et al., 2014; Hayden et al., 2012], or software transactional memory [Carvalho and Cachopo, 2011], where the programmer's help allows to reduce the performance impact of identifying critical points in the program.

3.4.4 Resource types and usage

In the model presented at Section 3.4.2, $Savings_r$ refers to any computational resource (r), a measurable computational asset which applications consume to make progress. Resources can be classified as either *explicit* or *implicit*, regarding the way they are consumed. *Explicit* resources are the ones that applications request during execution, such as, the number of allocated objects, the number of network connections, the number of opened files. *Implicit* resources are consumed as a 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 resource types are relevant to be monitored and regulated. *Explicit* and *implicit* resources might be constrained as a protection mechanism against ill behaved or misusing applications [Geoffray et al., 2009]. For well-behaved applications, limiting 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

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

Table 3.1: Implicit resources and their throttling properties

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 memory space is not a guarantee that the application will benefit from that allocation, while restraining memory space will still allow the application to make some progress.

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

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

Summary

This chapter presented a general resource allocation and adaptation schema which obeys to a VM economic model, which aims to maximize overall quality-of-execution (QoE) through resource efficiency. Essentially, our system wants to put resources where they can do the most good to applications and the cloud infrastructure provider, while taking them from where they can do the least harm to applications (from which they are taken).

4 Resource Management Mechanisms

Contents

Chapter overview

In this chapter we present some implementation details of the main resource management mechanisms that are supported by our managed execution platform. We focus on the necessary extensions to a high-level virtual machine regarding resource accounting, internal adaptability mechanisms and progress framework [Simão et al., 2011; Simão and Veiga, 2013c]. We also discuss how a serial checkpointing mechanism, which allows the migration of applications across nodes in a cluster, was modified to operate concurrently with the main execution of the application [Simão et al., 2012; Silva et al., 2013].

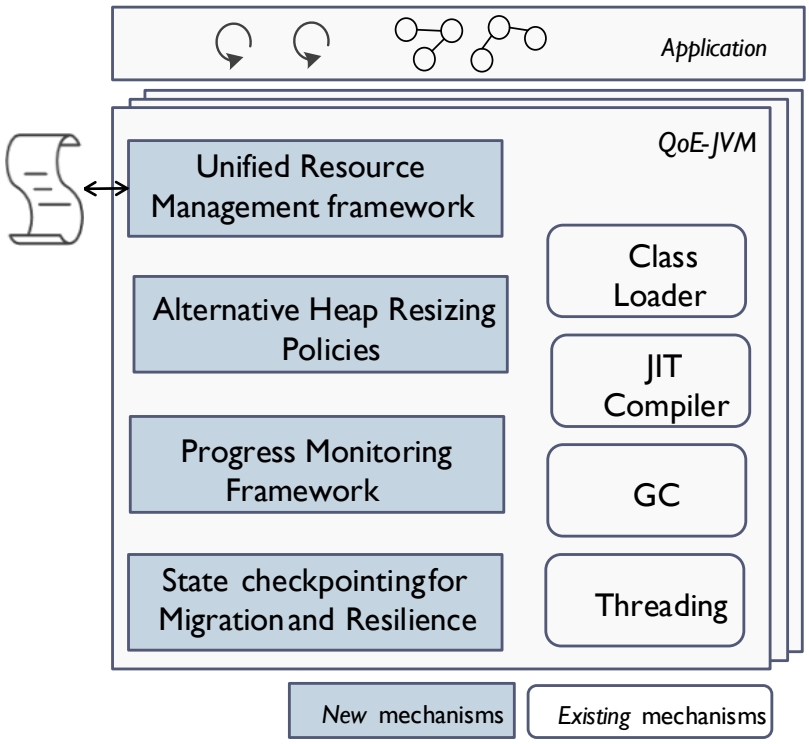


Figure 4.1: Layered view of the new resource management mechanisms

Figure 4.1 presents a layered view of the four extensions discussed in this chapter. The chapter starts by introducing the JVM's sub-systems that are relevant to our work.

4.1 Overview of the Jikes Research Virtual Machine

The Jikes Research Virtual Machine (Jikes RVM) [Alpern et al., 2005] is a Java VM (JVM) which provides a flexible infra-structure to test new virtual machine technologies. It is currently a widely used JVM in several research areas such as garbage collection [Cameron and Singer, 2014; Hertz et al., 2011; Kumar et al., 2012; Click et al., 2005], just-in-time compilation techniques [Brown and Horspool, 2010; Fink and Qian, 2003; Zhang and Krintz, 2005], debugging and reliability [Makarov and Hauswirth, 2013; Li et al., 2011] and cross-layers interactions [Frampton et al., 2009], with more than 188 publications. It is the open source evolution of Jalapeño, an IBM internal project [Alpern et al., 2000].[1]

The original goal of Jalapeño was to build an efficient JVM for servers, with the

[1] `http://jikesrvm.org/`, visited June 30, of 2014

additional property of being primarily written in Java. To solve this meta-circular nature, the compilation process of Jikes RVM includes the writing of an image file with the VM's initial execution state. A Java program called *image writer* runs in a regular JVM (called the *source* JVM) that loads the classes of the basic sub-systems of Jikes RVM, which are also written in Java. This program resembles a cross-compiler, where, in this case, the translation is from the object model of the source JVM to the one used by Jikes RVM. To boot the VM, this image is read by a small C and assembly program. After creating the necessary external dependencies with the underlying operating system (e.g. files for the System.in, System.out), Jikes RVM code will setup its sub-systems (GC and JIT compiler) and call the `Main` method of the class specified in the command line.

In the following sections we give further details of three sub-system related to the extensions presented later in this chapter.

4.1.1 Thread management

Japaleño started with a green threading model where, for one physical processor, the VM would request one operating system thread (called virtual processor) which would be used to spawn N Java threads. Because it was possible to use M physical processors, this model was also known as M-to-N threading. It adds some advantages such as rapid switching between mutators (i.e., regular Java threads) and GC threads, OS-independent locking mechanisms, and fast context switching. Recently, Jikes RVM has abandoned this approach and now (since version 3.1.0) maps each Java thread to an OS-native thread (e.g. a `pthread`), offloading most scheduling decisions to the OS.

All threads running in each Jikes RVM instance (either the ones spawned by the applications or the ones dedicated to garbage collection and compilation services) derive from `org.jikesrvm.scheduler.RVMThread`. This class provides relevant thread management state and operations that must still be present regardless of the OS thread scheduler.

Some runtime mechanisms, notably the garbage collector, need threads to yield their execution, regardless of the application code. A stop-the-world garbage collector needs to ensure that threads are stopped and in a *safe* state when scanning objects starting from each thread's roots. A thread in a *safe* state will not change the state of objects. The obvious way to reach this state is if the thread is blocked in some synchronization mechanism (e.g. a monitor). However, threads can be in an equivalent state even if not blocked.

For example, a thread running native code after a call to the Java Native Interface (JNI) cannot interfere with the garbage collector or other runtime mechanisms, and so it is in what is called an *effectively safe* state.

4.1.2 Memory management

Jikes RVM uses the Memory Management Toolkit (MMTk) [Blackburn et al., 2004; Shahriyar et al., 2013] for garbage collection services. This toolkit provides the implementation of several well-known garbage collection strategies (e.g. copying, mark-sweep, reference counting, generational) and a well-structured framework to develop new strategies. MMTk is organized around high-level concepts, such as the *policy*, which associates a given memory space to an allocation and collection algorithm. Generational garbage collectors, which divide the heap in two or more logical spaces (i.e., generations) can use more than one *policy*.

Underpinning the execution of all memory management strategies is the request of new virtual memory pages. This operation is done lazily by MMTk as more heap space is needed. The total space used by the heap is adjusted during the execution of the application. The size of this space has a complex relation with the application's execution time. A small heap will result in fast but frequent collections while a larger heap will result in longer but less frequent collections. Concurrently with this trade-off, when the heap size goes beyond a certain dimension, the number of page faults will increase.

4.1.3 Extensions to the language and Native Calls

Some services inside Jikes RVM need to execute operations that are not available in the original Java language. An example is pointer arithmetic used in garbage collectors. This is done using Magic [Frampton et al., 2009], a small set of extensions targeting high-level languages.

In some situations the core code of the Jikes RVM has to directly use services of the underlying operating system. Scheduling mechanisms are a common place to find these interactions, when, for example, threads have to block in synchronization mechanisms. Although Java has a mechanism for native invocations, the Java Native Interface (JNI), Jikes RVM does not use such calls for its own internal communications with the operating system interface, or "C" libraries in general, because of the extra overhead they

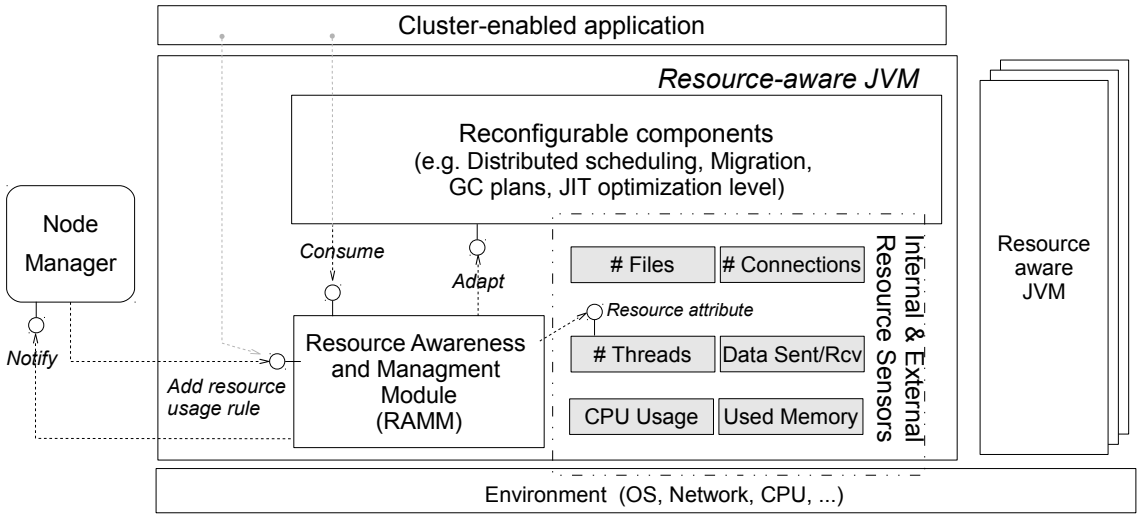


Figure 4.2: Interactions with the Resource Awareness and Management Module

imply. This overhead is the result of, for example, setting up a pointer to the caller object and the JNI interface pointer so that native code can access JNI functions representing Java services (e.g. string manipulation, reflection).

4.2 Resource accounting framework

To implement our architecture, we need to develop a managed language virtual machine with the capacity to monitor and restraint the use of resources based on a dynamic policy, defined declaratively outside the VM. Some work has been done in the past aiming to introduce resource-awareness in such high-level virtual machines (which details were presented in Section 2.5.2). Nevertheless, to the best of our knowledge, none of them is publicly available or currently usable with widely used software, operating systems, and hardware architectures. Based on this observation, we have chosen to extend the Jikes RVM [Alpern et al., 2005] to be resource-aware. Thus, in the next subsection we will describe different aspects of our current work on Jikes RVM.

Figure 4.2 depicts further details on the architecture of the resource-aware VM we developed for *QoE*-JVM . The resource-aware HLL-VM has a specific module for each type of manageable resource (e.g., files, threads, CPU usage, connections, bandwidth, and memory). Each of the module exports to the Resource Awareness and Manage-

ment Module (RAMM) an *attribute* that abstracts the specifics of the resource. This way, when the RAMM decides to limit, reduce or block the usage of a resource by the application, it can instruct the respective attribute without worrying about the details of applying limitation to that specific resource (e.g., disallowing file open, or take a thread out of scheduling). The RAMM consumes profile information from the main VM and QoE-JVM mechanisms (GC and JIT level, and migration, respectively). These mechanisms can be adapted and reconfigured by command of the RAMM.

4.2.1 Resource management policies

The management of a given resource implies the capacity to monitor its current state and to be able to directly or indirectly control its usage. The resources that can be monitored in a virtual machine can be either specific of the runtime (e.g. number of threads, number of objects, amount of memory) or be strongly dependent of the underlying architecture and operating system (e.g. CPU usage). To unify the management of such disparate types of resources, we carried out the implementation of JSR 284 - The Resource Management API [Grzegorz Czajkowski, 2009] in the context of Jikes RVM, previously not implemented in the context of any widely usable virtual machine.

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

A *Consumer* represents an executing entity which can be a thread or the whole VM. Each consumer is bound to a resource through a *Resource Domain*. Although *consumers* can be bound to different *Resource Domains*, they cannot be associated to the same *resource* through different *Domains*. Currently, we consider the whole VM as a single consumer.

Resource domains impose a common resource management policy to all registered *consumers*. This policy is programmable through callback functions, which can be either *constraints* or *notifications*. *Constraints* exist to impose resource consumption limits

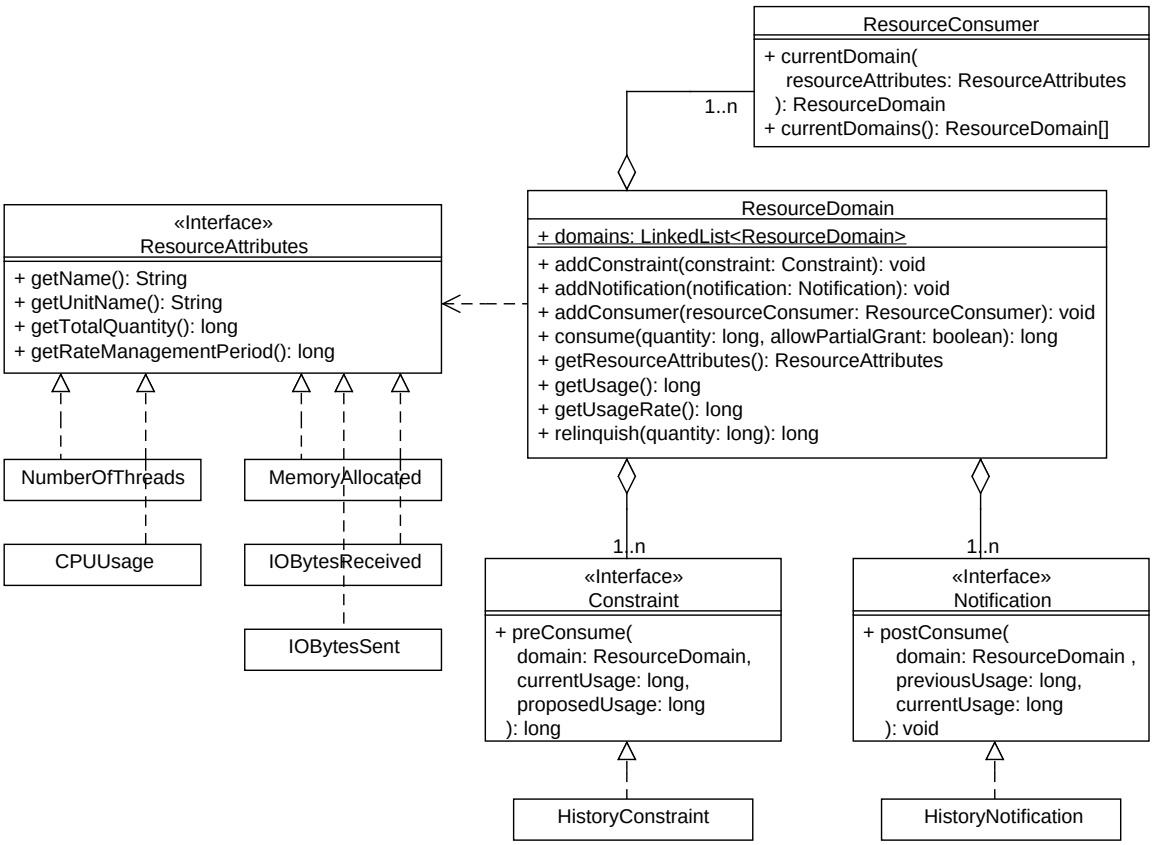


Figure 4.3: Class diagram with the main entities of JSR 284, Resource attributes, Constraints, and Notifications

while *notifications* can be used to monitor the usage of a certain resource and determine more complex actions (e.g. migration of the execution state). The JSR 284 does not specify what happens when the consumption of a certain amount of resources is accepted or denied by the installed constraints. The behavior of our system in this event is determined by the installed rules, namely the elements `<OnAfterComsumption>` or `<OnLimit>`, as exemplified in the policy of Figure 3.2 in Chapter 3.

Figure 4.3 presents the main interfaces specified by the JSR 284. Basically, the code artifacts defined by this JSR are only these interfaces, along with their rationale detailed in the documentation.[2] The figure also includes relevant classes representing resource attributes, constraints and notifications that are currently supported by our execution environment. The `ResourceDomain` resembles the *observable* from the Observer de-

[2]Although there is a link to a reference implementation in `https://home.java.net/`, the SVN repository for the project `jsr284-ri-tck` does not contain any source code. Visited May 11, 2012.

```
1  public class HistoryAverage implements Constraint {
2    ...
3    long[] _samplesHistory;
4    public HistoryAverage(int wndSize, long maxConsumption)
5    { ... }
6    public long preConsume(ResourceDomain domain,
7           long currentUsage, long proposedUsage) {
8      long average = 0;
9      if (_nSamples == _samplesHistory.length) {
10       average = _currentSum / _nSamples;
11       _currentSum -= _samplesHistory[_idx];
12     }
13     else { _nSamples   += 1; }
14     _currentSum += proposedUsage;
15     _samplesHistory[_idx] = proposedUsage;
16     _idx = (_idx + 1) % _samplesHistory.length;
17     return average > _maxConsumption ? 0 : proposedUsage;
18   }
19 }
```

Figure 4.4: Regulate consumption based on past $wndSize$ observations

sign pattern [Gamma et al., 1995] where interested *observers* register for pre- or post-consumption information, as `Constraints` or `Notifications`, respectively.

Figure 4.4 shows an example of a *constraint*, `HistoryAverage`, which can be used to regulate a CPU usage policy. Consider a scenario where the running application cannot use the CPU above a threshold for a given time window, because the remaining CPU available is reserved for another application (e.g., as part of the quality-of-execution awarded to it). In this case, when the CPU usage monitor evaluates this rule, it would suspend all threads (i.e., return `0` for the allowed usage) if the intended usage is above the average of the last `wndSize` observations. A practical case would be to suspend the application if the CPU usage is above 75% for more than 5 observations.

4.2.2 Resource management hooks in the VM and classpath

A new package of classes was integrated in the GNU classpath in order for applications to be able to specify their policies.[3] With this infrastructure, all consumable resources

[3] `http://www.gnu.org/software/classpath/`, visited July 4, 2014.

monitored, or directly controlled by the VM and class library, can be constrained by high-level policies defined externally to the VM runtime.

These classes interact with the resource-aware underlying VM so that applications can add their own resource consumption policies, if needed. This is useful if there are third party libraries, used by the application, which want to impose their own restrictions or need to receive notifications of resource consumptions. Regarding the particular case of constraints, application-defined policies cannot give more resources than the ones allowed for each application by the provider. This is enforced by taking the minimum values of each constraint for a particular resource domain.

We have considered two kinds of resource consumption, as defined in Section 3.4.4: explicit and implicit. The control of explicitly consumed resources has a similar implementation. In general, the resources that can be consumed explicitly (e.g. file, sockets, threads) are already represented in the platform as Java classes which the application uses directly (e.g. `File`, `Socket`, `Thread`). During the creation of these objects, to account the number of instances, and during the use of these services, to account usage rates, the installed rules are evaluated.

Examples of controlled explicit resources include the number of threads, which is a common source of CPU contention and performance degradation when multiple applications are running, and the amount of live objects.

Regarding the first example, 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 RVM class responsible for the creation and representation of all threads was also extended to use the callbacks of this *resource domain*, so that the number of new threads is determined by a policy defined declaratively outside the runtime. VM-internal threads (e.g. GC, finalizer) are not accounted.

Regarding the second example, the call to module RAMM, passing an instance of `UsedMemoryResourceAttributes`, is made in method `RuntimeEntrypoints.resolvedNewScalar(RVMClass cls)`, as demonstrated in Figure 4.5. The final stage of the garbage collection process was modified to relinquish the amount of memory freed.

When a resource is about to be consumed, the decision on whether the operation is allowed or not is delegated to the RAMM, giving as parameter an instance of a `ResourceAttribute`, which describes the resource that is about to be consumed. The RAMM, based on the type of `ResourceAttribute`, then obtains the corresponding

```
import org.jikesrvm.ramm.RAMM;
import org.jikesrvm.ramm.resourceattributes.UsedMemoryResourceAttributes;
import org.jikesrvm.mm.mminterface.MemoryManager;
public class class RuntimeEntrypoints
  implements Constants,
             ArchitectureSpecific.StackframeLayoutConstants {
  public static Object resolvedNewScalar(RVMClass cls) {
    int size = cls.getInstanceSize();
    if (VM.fullyBooted)
      if (RAMM.initialized && RAMM.enableMem) {
        if (RAMM.consume(
                UsedMemoryResourceAttributes.getInstance(),
                size, false) == 0) {
          VM.sysWriteln("[A2RVM@MemoryUsageMonitor]:
                        Memory policy threshold excedeed.");
          MemoryManager.isMemoryExhausted = true;
        }
      }
    // remaining unchanged code
  }
}
```

Figure 4.5: Modification to the code that resolves the bytecode 0xbb (“new”)

```
public final class ResourceDomain
  private long usage;
  // ...
  public long consume(long quantity, boolean allowPartialGrant) {
    long proposedUsage = usage + quantity,
         min = Long.MAX_VALUE,
         result = 0;
    for (Constraint constraint : constraints) {
      result = constraint.preConsume(this, usage, proposedUsage);
      if (result < min)
        min = result;
    }
    if (result < quantity)
      return 0;
    usage = result;
    return result;
  }
}
```

Figure 4.6: Delegation of resource consumption decision to the installed constraints

```
1 package org.jikesrvm.runtime;

3 public abstract class SysCall {
4   // ...
5   @SysCallTemplate
6   public abstract int sysCPUUsage();
7 }
```

Figure 4.7: Java stub to generate call to native code

resource domain and requests that a certain amount of units be consumed. This will result in either allowing, delaying, or denying the request (thrown and exception). Figure 4.6 shows the code in the `ResourceDomain` class which is responsible for querying the installed *constraints*.

The information regarding the consumption of implicit resources is currently obtained using the operating system services, namely the `/proc` filesystem. Others have adopted a solution that relies on the instrumentation of bytecode. These approaches have the potential for a significant overhead, in both code size and execution performance. Hulaas et al. [Hulaas and Binder, 2008] reports a framework, J-RAF2, for the demanding task of per-thread accurate CPU accounting. J-RAF2 increases code size in approximately 14% and has an average execution overhead of 30%. Given these observations, our approach depends on a small portion of native code that can be easily tailored to a specific operating system.

Regarding CPU usage, we have modified the booting process of the Jikes RVM so that a new internal thread is created to monitor this activity. The interaction with OS system calls is efficiently supported by the available JIT compilers. When a properly annotated method is called, the JIT compiler will generate a call to a "C" language *stub*, using the platform's underlying calling convention. An example of the stub for reading CPU usage, by the above mentioned thread, is presented in Figure 4.7. The code invoked by the stub then reads from `/proc`, and returns the results, as presented in Figure 4.8.

4.2.3 Yield-driven heap management

The process of garbage collection (GC) relates to execution time but also to allocated memory. On the CPU side, a GC algorithm must strive to minimize the pause times (more so if of stop-the-world type). On the memory side, because memory management

```
1  extern "C" long unsigned sysCPUUsage(void)
2  {
3    char buf[LIMIT];
4    long unsigned cpuUsageUser, cpuUsageKernel;
5    int i, paramCount;
6
7    fp = fopen("/proc/self/stat", "r");
8    if (fgets (buf, LIMIT, fp) == NULL) { /* error */ }
9    fclose(fp);
10
11   cpuUsageUser=0; cpuUsageKernel=0; i=0; paramCount=0;
12   while (i<LIMIT) {
13     if (buf[i++] == ' ') {
14       ++paramCount;
15       if (paramCount == PARAM_IDX) break;
16     }
17   }
18   if (i == LIMIT) { /* error */}
19   ++i;
20   while (i < LIMIT && buf[i]!=' ')
21     cpuUsageUser = cpuUsageUser*10 + (buf[i++]-'0');
22   if (i == LIMIT) { /* error */}
23   i++;
24   while (i < LIMIT && buf[i]!=' ')
25     cpuUsageKernel = cpuUsageKernel*10 + (buf[i++]-'0');
26   return (cpuUsageUser+cpuUsageKernel);
27 }
```

Figure 4.8: Native code for reading /proc cpu usage

is virtualized, the allocated memory of a managed language runtime is typically bigger than the actual working set. With many runtimes executing in a shared environment, it is important to keep them using the least amount of memory to accomplish their work.

Independently of the GC algorithm, runtimes can manage the maximum heap size, mainly determining the maximum used memory. In the memory management system of the research runtime Jikes RVM, after a full heap collection, the runtime considers whether the heap should change size. The algorithm to change the heap size takes into account the percentage of live objects and the ratio of time spent in GC. The live objects ratio measures the relation between the total memory reserved (which includes large, immortal and non-movable objects spaces) and the space reserved for regular objects allocation, as presented in Equation 4.1.

$$liveObjectsRatio = \frac{reservedMemory}{currentHeapSize} \tag{4.1}$$

On the other hand, the ratio of time spent in GC measures the relation between the accumulated time spent in GC related activities and total application time, as presented in Equation 4.2.

$$gcTimeRatio = \frac{\sum_{collection=0}^{n} GCDuration_{collection}}{totalAppTime} \tag{4.2}$$

This heap size change policy is represented by a function that takes as input the $liveObjectsRatio$ and the $gcTimeRatio$, returning the heap size growth/shrink percentage. In this document, we refer to this function as a *matrix* because it maps two parameters to one. The default policy determines that the heap shrinks about 10% when the time spent in GC is low (less than 7%) when compared to regular program execution, and the ratio of live objects is also low (less than 50%). This allows for savings in memory used. On the other hand, the heap will grow by about 50%, when the execution environment spends more time in GC activities, and the number of live objects still remains high. This growth in heap size will lead to an increase in memory used by the runtime, aiming to use less CPU time because the GC will run less frequently.

Considering this heap management strategy, the $heapsize$ is the resource which contributes to the $yield$ measurement. To determine how the workloads executed by each tenant react to different heap management strategies, e.g., more targeted at heap expansion or more at heap saving, we apply Equation 3.6. The memory savings ($Savings_{hsize}$)

are then found by comparing the results of applying two different allocation policies, that is, two different allocation matrices, M_α and M_β, as presented in Equation 4.3. In this equation, U_{hsize} represents the maximum number of bytes assigned to the heap.

$$Savings_{hsize} = \frac{U_{hsize}(M_\alpha) - U_{hsize}(M_\beta)}{U_{hsize}(M_\alpha)} \tag{4.3}$$

4.2.4 Yield-driven CPU ballooning

A similar approach can be extended to CPU management, employing a strategy inspired by the *ballooning* mechanism available to virtual machine monitors. In system VMs, ballooning works by having a customized kernel driver installed at the guess OS allocating memory pages excessively, in order to drive the guest OS into swapping, and reduce the amount of useful pages in the guest's physical memory. This allows the core working set of the guest VM to be obtained with a grey-box approach. We have conducted experiments in *ballooning* the CPU by using OS processes' priority levels and capping mechanisms of the Linux containers.[4]

In the first case, each instance of the JikesRVM bounds to a single core (calling the Linux CPU scheduler API using the extensions described in Section 4.1.3) while an auxiliary process exists in each machine, that is executed with the highest priority. This last process, when active, can be regulated to consume more or less CPU by varying its passive waiting period. Because of its high priority, this behavior will have impact on the regular JikesRVM processes. This approach is effective to assess the relative efficiency of the remaining CPU usage awarded to a JVM but is only useful and justified when there are other JVMs to award the CPU time to.

When experimenting with the Linux containers, we executed each JikesRVM process inside a container, and used the parameter `lxc.cgroup.cpuset.cpus`, to determine core affinity, and `lxc.cgroup.cpu.shares`, to determine the proportional shares. This second approach proved to be more useful since the one relying only on priorities made it very difficult to impose a given percentage of CPU restriction. When using the `shares` parameter, if the initial share is, for example, 1000, and then changed to 750, this means the affected JikesRVM will get 25% less CPU time.

[4] `https://linuxcontainers.org/`, visited July 3, 2014

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD,
         ElementType.FIELD,
         ElementType.PARAMETER})
public @interface ProgressMeter {
  double relevance() default 1.0;
}
```

Figure 4.9: The *progress* annotation

```
public class AClass {
  @ProgressMeter(relevance=0.8)
  public void m1() { ... }
  public void m2(@ProgressMeter(relevance=0.2) p) {
    for (int i=p; i<limit; ++i) ...
  }
}
```

Figure 4.10: Example of usage of the *progress* annotation

Akin approaches would include specifically designed user-level mechanisms for CPU scheduling [dos Reis and Cerqueira, 2010], or with the hypervisor CPU scheduler, lowering CPU caps or reassigning the VCPUs to CPUs mapping [Barham et al., 2003].[5]

Thus, regarding CPU as the resource, the savings in computational capability (that can be transferred to other tenants) can be measured in MFLOPS or against a known benchmark as Linpack. The savings are found by comparing two different CPU shares or priorities, CPU_α and CPU_β, as presented in Equation 4.4. In this case, U_{Mflops} gives us the total CPU saved, e.g., relative to the number of MFLOPS or Linpack benchmarks that can be run with the CPU 'saved'.

$$Savings_{Mflops} = \frac{U_{Mflops}(CPU_\alpha) - U_{Mflops}(CPU_\beta)}{U_{Mflops}(CPU_\alpha)} \tag{4.4}$$

4.3 Progress monitoring library

In this section, we discuss some of the implementation details regarding the progress library used by QoE-JVM to capture the frequency of calls to methods identified by the

[5] `http://wiki.xenproject.org/wiki/Credit_Scheduler`, visited at June 16, 2014

```
1 public class ProgressAgent {
2   public static void premain(String args, Instrumentation inst) {
3     inst.addTransformer(
4       new MethodLookupTransformer(args), false
5     );
6   }
7 }
```

Figure 4.11: Entrypoint of the Java agent

programmer as most relevant to the progress of the application. The identification is made using annotations like the one presented in Figure 4.9. These are placed by the programmer into strategic places of the code, where the application is known to make effective progress.

By their definition, these annotations can be, in principle, applied to methods, fields and method parameters. If applied to methods what is relevant is to know what is the call rate to the method. We currently keep track of the *overall call rate* (OCR) which represents the call frequency since the application started. Periodically, QoE-JVM also calculates a *window call rate* (WCR) which is the call frequency in the last observation window (e.g. number of calls in the last 5 seconds). This helps to determine approximate derivatives of these values and to promptly detect possible phase changes in application execution. If applied to fields or parameters, what is relevant to know is the frequency of writes.

The annotation is characterized by the relative contribution of the method to the overall application progress. Figure 4.10 presents a code snippet with two usage examples, one associated with a method and the other associated with a parameter. All annotations are used to insert progress measurement code at load time which will either count method calls or updates regarding fields and local variables dependent on parameters. The current implementation only looks for annotations associated with methods.

To process annotations at start time and instrument the program with the measuring code, QoE-JVM augmented the load process of a JVM, using an instrumentation Java agent of the `java.lang.instrument` package, to look for annotations in the classes metadata as they are loaded into the VM. The base structure of the Java instrumentation agent used is represented in Figure 4.11. The agent does not have to obey to a given interface but it has to define the a method called `premain`. Here, the agent's

arguments are processed (e.g. observation window size) and the instrumentation process can be setup using the services provided by a JVM specific implementation of the `Instrumentation` interface. These services include the hooking of a class-level byte code transformer, represented by the interface `ClassFileTransformer`.

The agent transverses each class' metadata and inserts progress measuring code where necessary, as explained in the steps of Figure 4.12. This was done with the help of Javassist, a framework which allows injection of source code and bytecode.[6] The inserted code, in step 3, is a call to the method that updates the per-method OCR. Besides this instrumentation code during class loading, the agent spawns a thread which, in the predefined period, looks at the current and previous OCR to determine the WCR.

4.4 Checkpointing and migration of the execution state

This section presents the extensions made to a mechanism to checkpointing and restore the application's execution state, so that the checkpoint operation can be made concurrently with the application, instead of having to stop the execution during the process. The base checkpointing mechanism [Garrochinho, 2010] also extends previous work [Quitadamo and Leonardi, 2008] by i) being able to autonomously checkpoint and restore the whole JVM execution, ii) providing support for the modern version of Jikes RVM, where the green threading model was abandoned, and iii) supporting persistence of objects representing files. We start by highlighting the main challenges regarding obtaining a consistent checkpoint of the execution state and the restoring of that state. We support our discussion on the overview of the Jikes RVM, particularly Section 4.1.1, where we have introduced the life cycle of threads managed by the Jikes RVM.

4.4.1 Consistent extraction of the execution state

To obtain a consistent snapshot of the application's execution context, the checkpointing mechanism must extract the execution state and the application state. The former consists, in essence, of the current method in execution and its instruction pointer together with the sequence of calls that resulted in the current one. The latter is the representation of all the objects that were created during the application's life time and are still reachable. In its simplest form, these operations require the following steps to be made

[6] `http://www.csg.ci.i.u-tokyo.ac.jp/~chiba/javassist/`, visited July 1, 2014

```
1  package org.inescid.gsd.qoevm.agent;
2  import java.lang.instrument.*;
3  import java.security.ProtectionDomain;
4  import javassist.*;
5  import org.inescid.gsd.qoevm.annotation.ProgressMeter;

7  class MethodLookupTransformer implements ClassFileTransformer {
8    public byte[] transform(ClassLoader l, String name,
9                            Class<?> c, ProtectionDomain d, byte[] b)
10     throws IllegalClassFormatException {
11 /* 1. build Class descriptor from byte array 'b' */
12   try {
13     ClassPool pool = ClassPool.getDefault();
14     cl = pool.makeClass(new java.io.ByteArrayInputStream(b));
15     if (cl.isInterface() == false) {
16 /* 2. search for annotated methods and fields */
17       ArrayList<CtField> annotFields = new ArrayList<CtField>();
18       CtField[] fields = cl.getDeclaredFields();
19       for (int i = 0; i < fields.length; i++)
20         // ... keep record of fields with ProgessMeter annotation
21 /* 3. insert call to operation that marks progress */
22       CtMethod[] methods = cl.getDeclaredMethods();
23       for (int i = 0; i < methods.length; i++) {
24         if (methods[i].isEmpty() == false) {
25           Object annotation= methods[i].getAnnotation(
26               org.inescid.gsd.qoevm..ProgressMeter.class);
27           if (annotation!=null)
28             methods[i].insertBefore(
29           "org.inescid.gsd.qoevm.agent.ProgressMarker.markProgress"+
30           "(\""+methods[i].getName()+"\");");
31           final CtMethod method = methods[i];
32           for (CtField field : annotFields) {
33             final CtField annotField = field;
34             methods[i].instrument(new ExprEditor() {
35               public void edit(FieldAccess fa) throws
36                   CannotCompileException {
37                 if (fa.getFieldName().equals(annotField.getName())) {
38                   // ... detect read or write operation
39                   // ... with fa.isReader()
40           } } } }
41       // ... process eventual exceptions
42   }
43 }
```

Figure 4.12: Bytecode instrumentation inserting calls to the markProgess

in sequence: 1) pause all application-level threads, so that no new objects are created or modified, 2) extract and persist the call stack of every application-level thread, 3) extract and persist the state of all application-level objects, and 4) resume all application-level threads.

Regarding the first step, we have already discussed that the JIT compiler inserts yield points in the generated code. The code at these points, inserted in methods' prologues/epilogues and loop back edges, checks if the thread must yield or not, by calling the `RVMThread.checkBlock()` method. This method goes through an extensible system for blocking requests, already used by the GC and the locking sub-systems, to check if the calling thread must be blocked. If this happens, the thread is in what is known as a safe point - it is not executing bytecodes that could eventually trigger garbage collection work or any other runtime mechanism. In this case, a consistent snapshot of all call stacks could be preserved along with application state. After that, execution would be resumed.

Not all threads will be able to reach the mentioned safe points. Examples include threads that have called native code, using JNI, taking an unbounded time to return to the managed context. Threads that are already blocked in synchronization mechanisms will also not reach these safe points. Nevertheless, in all these situations, the thread in a state called "effectively safe". The thread is not necessarily suspended but it is, in practice, in a state where no bytecode is executing. A more detailed overview of each thread internal state is presented in the Jikes RVM project website.[7]

After the JVM is stopped, the second step can proceed. The extraction of the execution state should be done in a way that can be transfered to another running instance of the Jikes RVM in a remote node. To this end, the base checkpointing mechanism takes advantage of a feature of the Jikes RVM compilation system, the on-stack-replacement (OSR) mechanism [Fink and Qian, 2003], which was originally designed to allow dynamic recompilations of currently executing methods. The OSR is able to extract and represent in memory the state of a given method activation. The state is defined by the program counter (as a bytecode index), values of local variables (either literals or references to objects) and their stack location, and a reference to the activation's stack frame.

This mechanism is used to recursively extract the execution state of all methods

[7] `http://jikesrvm.org/Thread+Management`, visited June 30, 2014

in the call stack of a given thread, until the frame corresponding to the activation of `Thread.run()` is found. There was, however, the need to avoid extracting information of methods that could not be restarted in another context. A prominent example is when the top of the call stack is a method of a synchronization primitive managed by the OS. The state of these primitives cannot be migrated to a remote system.

To solve this, the base mechanism walks back in the call stack and starts extracting the state in the method that tried to acquire the lock. When the execution is restored, the thread (i.e., a given execution stack) owning the original lock must be the same. Other threads trying to obtain the lock will eventually acquire it in a different order from the original execution. In this work, we assume that applications are deterministic in the sense that the code path leading to the lock being acquired will surely be executed again. However, the acquisition and release order can be different from the original execution because of scheduling decisions of the operating system. Still, applications that depend on a specific order for these operations are not correct by nature [Silva et al., 2013].

The third and last phase is the persistence of the previously recorded state to an external file. Currently, this step relies on a combination of default and per-type serialization capabilities. Application-specific objects are assumed to have the capacity to be serialized.

The main challenge of the restore procedure is how to resume execution at the point of checkpointing while recovering the complete physical (machine dependent) call stack. The OSR mechanism offers half of the solution by injecting prologue bytecodes to recover the local variables and stack expressions. In addition to this OSR prologue, an *unconditional* call to the next method in the original call stack is inserted, immediately followed by a jump to the instruction that follows this call in the original execution.

These operations are all made at the bytecode level, leaving to the compiler the translation to the native layout, typically subjected to optimization. There is, in fact, two versions of each of these methods that need to be compiled. This is so because the specialized version can only be used during the first activation after restore. For the rest of the time, the prologues cannot be present.

4.4.2 Concurrent checkpointing

Our checkpointing mechanism can also run concurrently with the main program, avoiding full pause of the application during checkpointing, thus further reducing the over-

head 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. Checkpointing, 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 because if would be inconsistent.

The second issue stems from the fact that if we want to simultaneously freeze a *clone* of the application's 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 running applications.

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

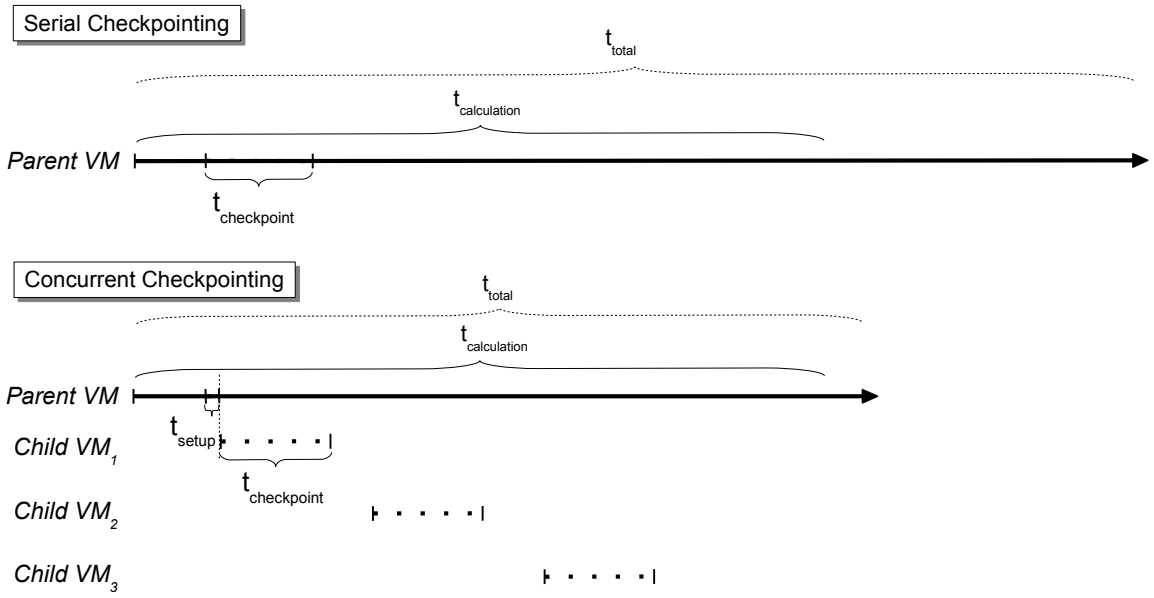


Figure 4.13: Timelines of serial and concurrent checkpoint

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 limited to the memory pages containing objects that are actually modified during the checkpointing. Due to the locality in memory accesses during application execution (locality-of-reference and working set principles), this amount is limited.

Figure 4.13 illustrates how the concurrent checkpoint progresses, along with the application, in comparison with the serial (non-concurrent) approach. $t_{calculation}$ is the free run time, without any checkpoint. t_{total} is the total execution time, considering either serial or concurrent checkpoint.

With serial checkpointing, the total execution time of an application is, expectedly, the sum of the time performing its calculations or processing (hereafter, calculation time), with the time to perform a checkpoint (once in the figure), assuming approximate times, multiplied by the number of checkpoints taken. Therefore, checkpointing is always in the critical path, regarding the total execution time, precluding 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.

The internal functioning is as follows. When checkpointing is triggered, the VM calls `fork()` to create a *child VM*, i.e., another process, sharing the whole address space, responsible solely for carrying out the actual checkpointing operation. The *copy-on-write* semantics ensures that the *child VM*'s working set will be consistent, even while the *parent* VM continues to update data, due to application execution. When checkpointing is complete, the child VM terminates, as it is no longer required.

There are, however, limitations in the semantics of the `fork()` primitive. Although the address space is shared, meaning that references collected in the *parent* VM will still be valid in the *child* VM, threads alive in the *parent* VM will not exist in the *child* VM. Only the thread that called the `fork()` primitive will be cloned. This means that if any of the remaining threads of the *parent* VM are holding a lock, that lock will still exist in the *child* VM (because of memory copy semantics) but the owner thread will not, and so, the protected resource will be permanently inaccessible.

This has consequences in what can be made in the *child* VM. The first two steps presented in Section 4.4.1, that is, stopping the VM and extracting the representation of each stack frame of every thread, must be done while all threads are alive. So, this must be completed in the *parent* VM. Nevertheless, this action should take only a very small fraction of the total checkpointing time.

The third and last step is the one where a significant delay can occur, since it depends on the transversal of an object graph with arbitrary size and the interaction with I/O primitives to write the overall state to disk. This should be done in the *child* VM. The *parent* VM can have threads that are blocked in Java monitors, supported either by Linux mutexes (`pthread_mutex`) or conditional variables (`pthread_cond`). Nevertheless, the *child* VM will not need to request the access to these application-level protected resources, avoiding a potential deadlock.

This strategy can be used in other VMs, besides Jikes RVM. In most cases, it will even be simpler to call the system's service, because most other VMs are implemented

using the same language of the operating system, C and C++.[8] Nevertheless, in the Jikes RVM, these operations (`fork()` and akin) are also efficiently supported by the available JIT compilers, as presented in Section 4.1.3. Our stub calls `fork()`, with reduced overhead, and returns the result to the calling VMs (i.e. *parent* or *child*).

Summary

This chapter described a combination of new resource allocation and scheduling mechanisms that were integrated in a Java Virtual Machine (JVM), the Jikes RVM. These mechanisms allow the management of co-located JVMs in an application-driven way, allowing the enforcement of the model discussed in Section 3.4. The resource management and progress frameworks along with the heap resizing policies allows for a fine-grained control of resource usage. The checkpoint mechanism allows for a more coarse-grain approach where applications can be migrated to nodes with more available resources. The ability to conduct the checkpoint in parallel with the application's execution makes the this mechanism tailored for scenarios were failures can occur in applications with a long execution time.

[8] `http://openjdk.java.net/groups/hotspot/`, visited July 4, 2014

5 Evaluation

Contents

Chapter overview

In this chapter, we show the evaluation of the adaptability mechanisms and the impact of the adaptability strategies, as described in Chapters 3 and 4. In particular, we want to evaluate:

- how do different workloads react to dynamic allocation of resources (Section 5.1)
- how costly is it to monitor and account resource usage in a language runtime (Section 5.2 and Section 5.3)
- what are the prospective benefits of checkpointing in the event of failures (Section 5.4).

The evaluations were made in machines with Intel(R) Core(TM)2 Quad processors (with four cores) and 8GB of RAM, each running Linux Ubuntu 12.04. Our extensions were made to Jikes RVM [Alpern et al., 2005] code base, version 3.1.1.

Measuring performance of code running in a JVM is a challenging task because there are multiple factors to account for, such as, the structure of the application, the configuration of the virtual machine (memory management strategy, heap size, dynamic optimizer, threading systems, and so on), and the hardware [Georges et al., 2008]. One of the main factors of non-determinism during evaluation is usually associated with the sampling mechanism used by the JIT compiler to determine hotspots. Because sampling can observe different values from an execution to the other, the JIT optimization plans can also vary. To avoid this disturbance, and according to the suggestions on the VM web site, we used the replay compilation option when evaluating internal modifications to the Jikes RVM.[1]

5.1 QoE applied to memory and CPU management

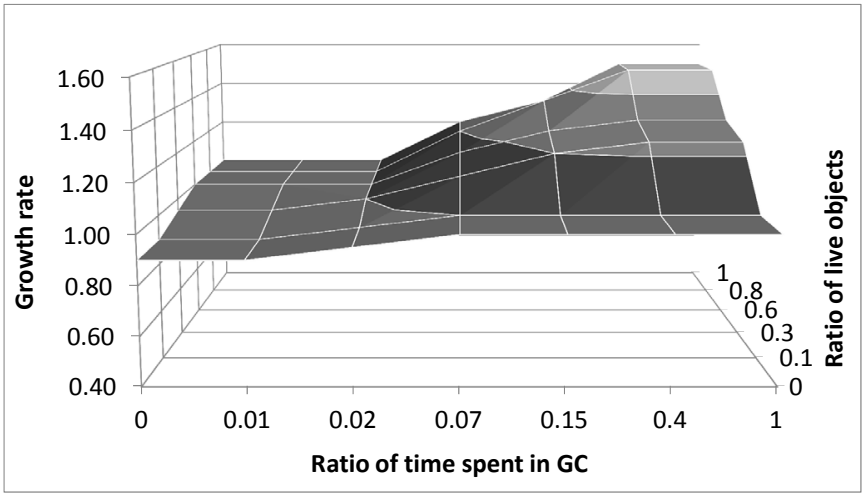
The resource management economics, presented in Chapter 3.4.2, was applied to manage the heap size and CPU usage regarding different types of workloads.

5.1.1 Heap size management

The Jikes RVM default heap growing matrix (hereafter known as M_0) is presented in Figure 5.1. Figure 5.2 shows QoE-JVM alternative matrices. In all matrices, 1.0 is the neutral value, representing a situation where the heap will neither grow nor shrink. Other values represent a factor of growth or shrinkage, depending on whether the value is greater or smaller than 1.0, respectively. To assess the benefits of our resource management economics, we have setup three new heap size changing matrices. The distinctive *factors* are the growth and decrease rates determined by each matrix.

Matrices M_1 and M_2, presented in Figure 5.2.a and 5.2.b, impose a strong reduction on the heap size when memory usage and management activity is low (i.e., few live objects and short time spent on GC). This is especially useful in consolidated multi-tenancy scenarios where there is usually some measure of overcommitment. Nevertheless, they provide very different growth rates, with M_1 having a faster rate when heap space is scarce. Finally, matrix M_3 makes the heap grow and shrink very slowly, enforcing a more rigid and conservative heap size until program dynamics reach a high activity

[1] http://jikesrvm.org/Experimental+Guidelines, visited July 2, 2014



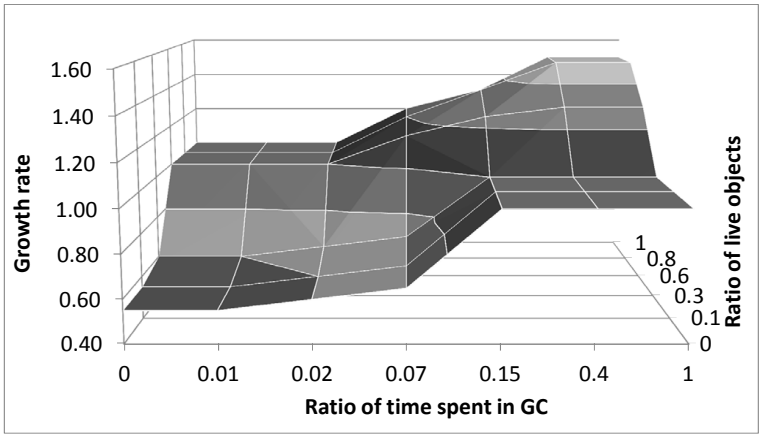
(a) M_0

Figure 5.1: Default heap growth matrix.

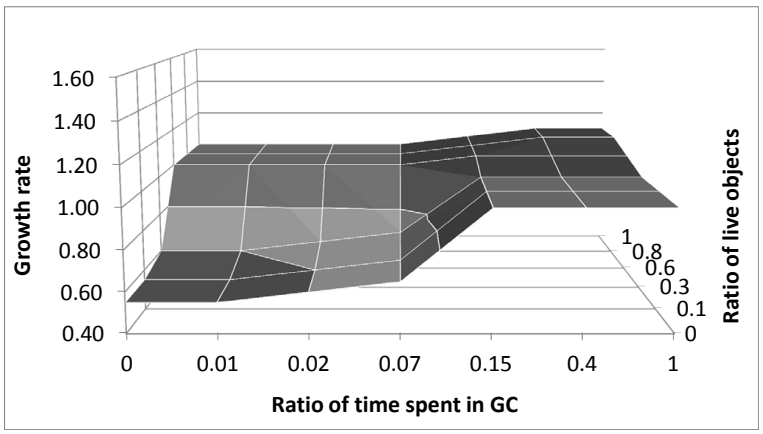
point (i.e., high rate of live objects and longer time spent on GC) or decrease activity sharply.

The overall behavior of the matrices can be visualized, alternatively, as directives for percentage increases and reductions in heap size, being depicted with up or down arrows, in Figure 5.3 to Figure 5.4. These new matrices explore different shrink/grow percentages across an imaginary four-quadrant space. For example, M_1 is more heap conservative, meaning that, when a small percentage of time is spent on GC and live objects remain below 30%, the heap will decrease between 30% and 45%, while in M_0 the same situation implies a resizing between $+10\%$ and -10%.

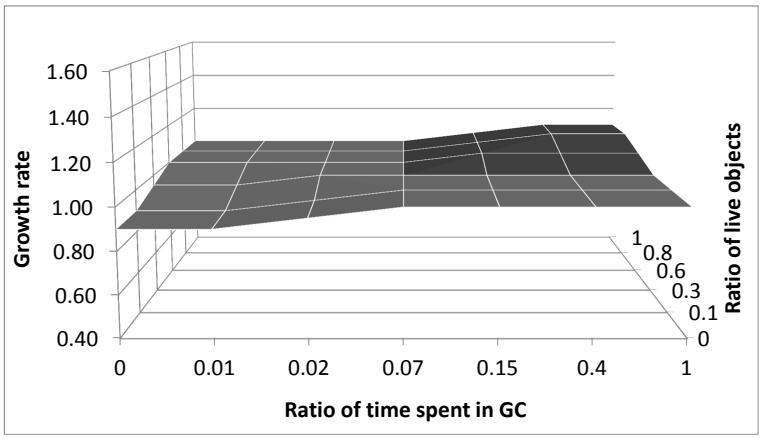
Furthermore, to compare the matrices, from a quantitative point of view, we define two norms, the *growth norm*, presented in Equation 5.1, and the *shrinkage norm*, presented in Equation 5.2. These capture the aggressiveness of impact (when expanding or shrinking), that the different matrices have in the heap size, and its skew/bias towards expansion or shrinkage (more or less expanding, or shrinking-driven). In particular, Equation 5.1 calculates a sum, across the two dimensions of the matrix (inspired by an integral over a plane, but in a discrete domain), aggregating only the net contributions for heap expansion found in the matrix (only the part greater than 1 in each element). Conversely, Equation 5.2 calculates a sum, across the two dimensions of the matrix (also inspired by an integral over a plane), this time aggregating only the net contributions for heap reduction in size.



(a) M_1



(b) M_2



(c) M_3

Figure 5.2: Alternative matrices to control the heap growth.

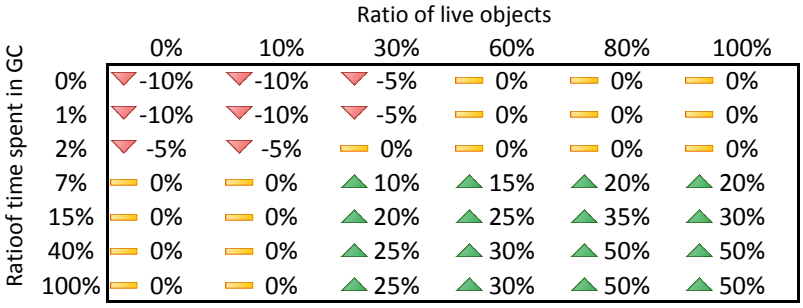


Figure 5.3: Growth and shrink percentage for the M_0 matrix

$$\|M\|_{growth} = \sum_{i=1}^{n}\sum_{j=1}^{m} g(a_{ij}), \text{ where } g(x) = \begin{cases} x-1 & \text{if } x > 1, \\ 0 & \text{if } x <= 1 \end{cases} \tag{5.1}$$

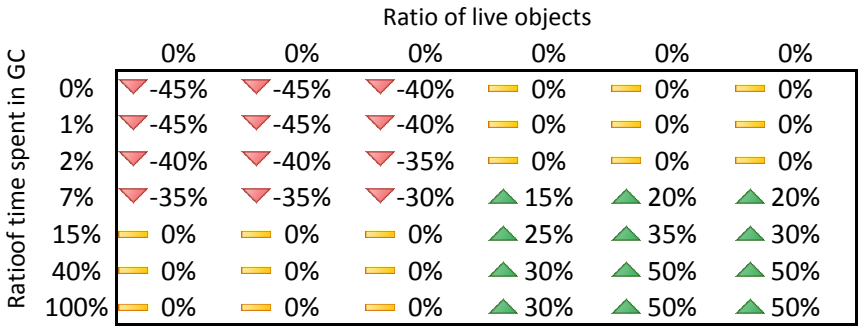
$$\|M\|_{shrinkage} = \sum_{i=1}^{n}\sum_{j=1}^{m} s(a_{ij}), \text{ where } s(x) = \begin{cases} 1-x & \text{if } x < 1, \\ 0 & \text{if } x >= 1 \end{cases} \tag{5.2}$$

Table 5.1 summarizes the norms of the four matrices (M_0 from the base implementation of Jikes RVM, and the M_1, M_2 and M_3 proposed alternatives). This helps us classifying a matrix and to quickly infer its expected behavior. Matrix M_0 is clearly biased towards expansion as its growth norm is greater than its shrinkage norm. This bias is assessed by the 8.08 ratio meaning an eight-fold potential more impact towards expansion than towards shrinkage. For instance, matrix M_2 has an opposite bias, as its norm ratio is 0.14, roughly 7 times more potential impact towards shrinkage than towards growth. Both of them have similar aggressiveness regarding aggregated impact (5.45 and 5.40).

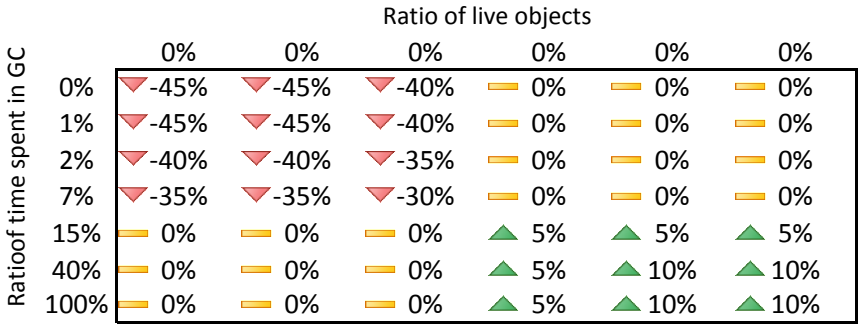
Table 5.1: Growth and shrink norms and their relation

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

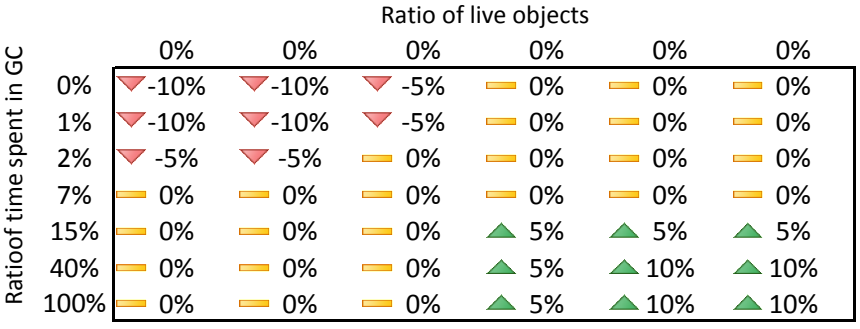
On the other hand, matrices M_1 and M_3 have much reduced and almost no bias, as their growth and shrinkage norms are almost equal, hence the ratio of approximately 1



(a) M_1 percentage increments

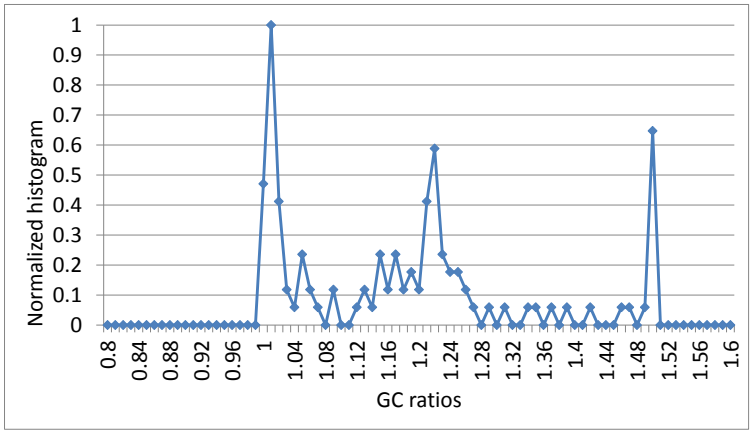


(b) M_2 percentage increments



(c) M_3 percentage increments

Figure 5.4: Growth and shrinkage percentage for each matrix



(a) M_0

Figure 5.5: Histogram of GC ratios for each benchmark using the default matrix

(1.08 and 0.85). However, they are very different regarding aggressiveness or potential impact, top for M_1 and very small for M_3.

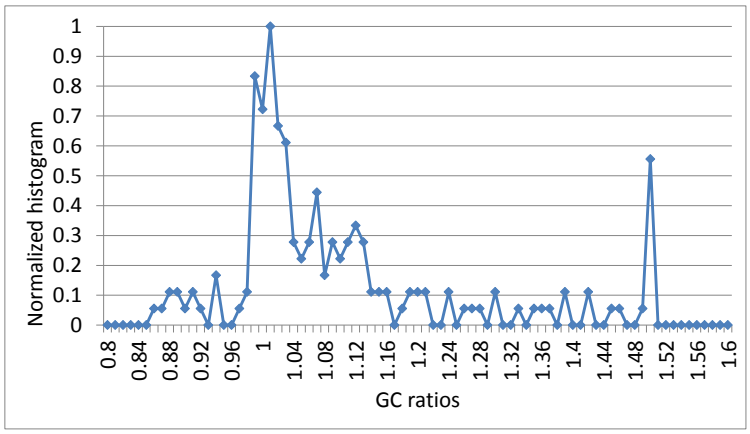
Figure 5.5 and Figure 5.6 show how the previously discussed matrices influence the use of different heap grow/shrink ratios when executing the benchmarks.

Each figure plots in the y-axis the frequency, in a normalized form, with which a given growth (x-value greater than 1) or shrinkage (x-value lower than 1) ratio is used, after a decision to change the heap size. From Figure 5.6.a, we can confirm that matrix M_1 is the one with a potential for larger impact (aggressiveness), because it causes heap change percentages with values from a wider interval, when compared to the other matrices. On the other hand, Figure 5.6.c corroborates that matrix M_3 is the one with the smallest aggressiveness as it uses the smallest interval of values.

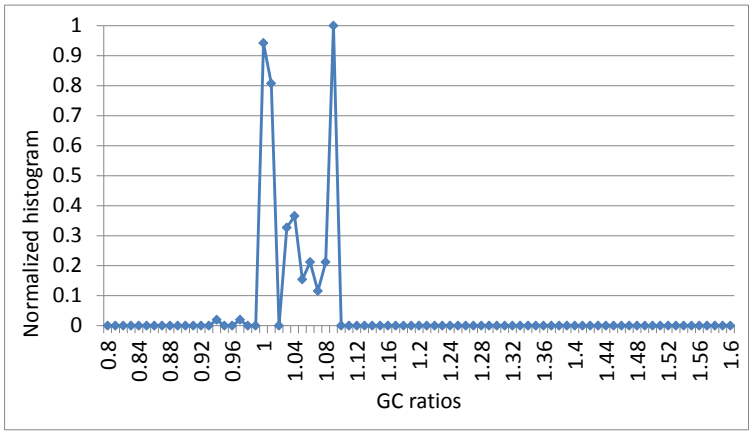
5.1.2 QoE Yield applied to Heap size

Each tenant using the Cloud provider's infrastructure can potentially be running different programs. Each of these programs will have a different *production*, i.e., execution time, based on the applied *capital*, i.e., the growth rate behavior allowed for the heap. To represent this diversity, we used benchmarks from DaCapo [Blackburn et al., 2006] and SPEC JVM 2008, which correspond to different ways of organizing programs in the Java platform.[2]

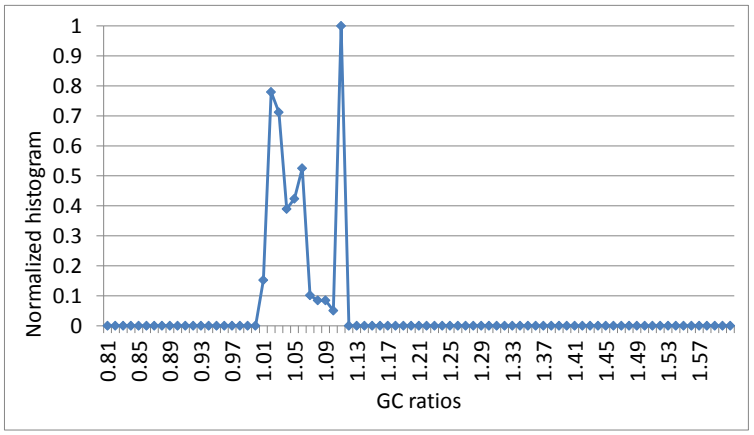
[2] `http://www.spec.org/jvm2008/`, visited November 17, 2012



(a) M_1



(b) M_2



(c) M_3

Figure 5.6: Histogram of GC ratios for each benchmark using other heap management matrices

To measure the *yield* of each matrix, we have setup an *identity matrix* (all 1's), that is, a matrix that never changes the heap size. Using this matrix is equivalent to having a fixed-sized heap. Figure 5.7.a shows the maximum heap size (left axis) after running the DaCapo benchmarks with configuration `large` and a subset of SPECjvm2008 using all the matrices presented in Figure 5.2.[3]

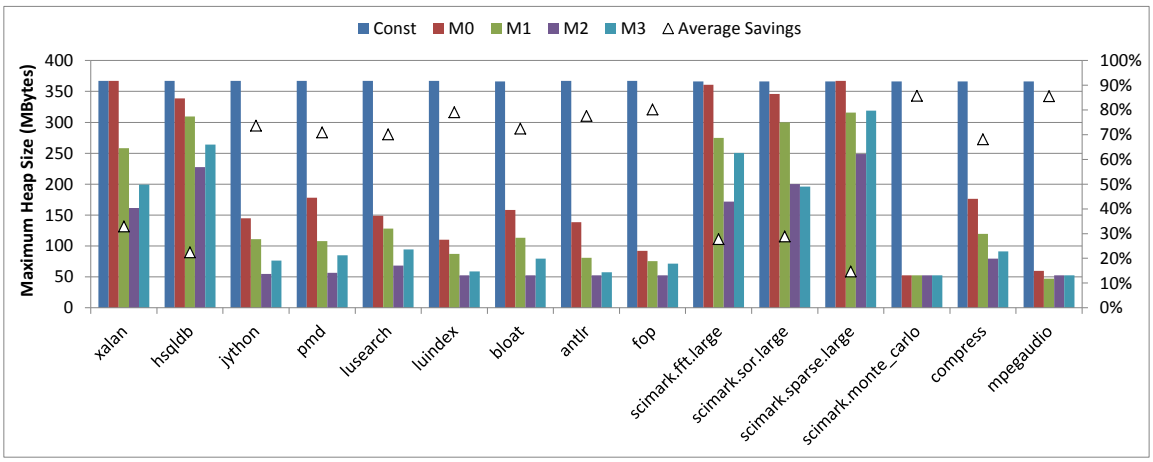
When using the four matrices, the heap size was configured to change between a minimum of 50MiBytes and a maximum of 350 MiBytes. In the right axis, we present the average *resource savings*, as defined in Equation 3.6, for each of the applications used in experimentation. A detailed view of the *resource savings* is presented in the first part of Table 5.2. They are above 40% for the majority of the workloads.

In Figure 5.7.b we present the evaluation time of the benchmarks (left axis) and the average *performance degradation* (right axis), as defined in Equation 3.7, regarding the use of each of the matrices. Degradation of execution time reaches a fourfold value for lusearch, Apache's fast text search engine library, but stays below 25% for the fast majority of the benchmarks. In particular, regarding the SPECjvm2008 benchmarks, most of them have negative *performance degradation*, that is, they run faster with the growth/shrink matrices controlling the heap than with a fixed size.

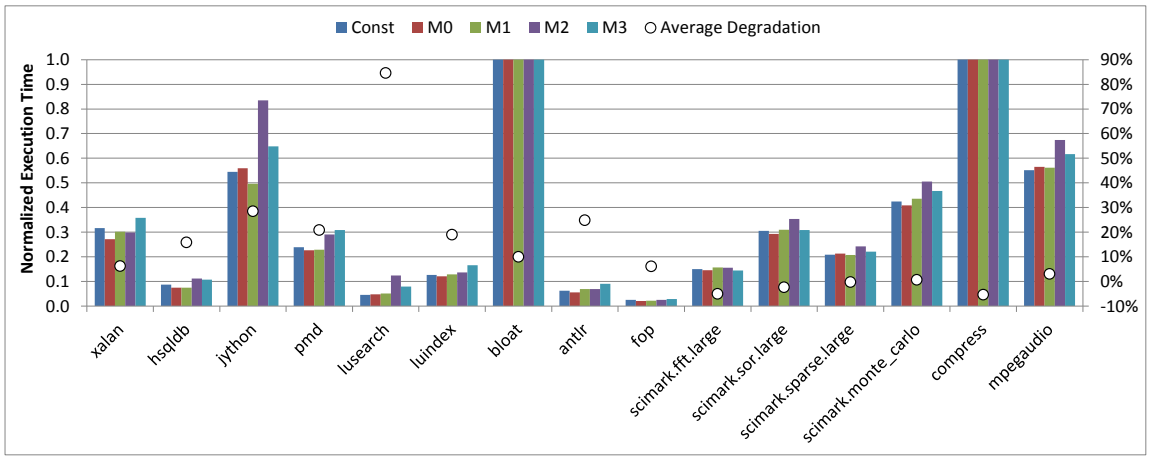
Table 5.2, summarizes the *yield*, as defined in Equation 3.4. Based on these results, the main observation is that under the same resource allocation strategy, that is, using the same resizing matrix, *resource savings* and *performance degradation* vary between applications. This demonstrates the usefulness of applying different strategies to specific applications. In most cases, the use of a matrix different from the default one (M_0) brings higher yields to the provider because, it saves more memory, but the execution time suffers a smaller degradation. For example, the `xalan` workload type has higher yield with M_3 (6.0) while `jython` and `pmd` benefit the most with matrix M_2 (28.4 and 9.2, respectively). When memory resources are scarce because of co-location, the provider will want to use, for each class of applications, the matrix that contributes to the higher yield. This is good for shared infrastructures to minimize the impact on the execution time taking the least of the resources of that workload.

We note also that a negative value represents a strategy that actually saves execution time. Not only memory is saved, but execution time is also lower. These scenarios are

[3]Due to incompatibilities with GNU classpath, not all SPECjvm2008 can be successfully executed in Jikes RVM.



(a) Maximum heap size and average savings percentage



(b) Execution time and average performance degradation percentage

Figure 5.7: Results of using each of the matrices ($M_{0..3}$), including savings and degradation when compared to a fixed heap size.

a minority in the Dacapo benchmarks and more frequent in the SPECjvm2008 benchmarks. They may simply reveal that the 350 MiBytes of fixed heap size is already causing too many page faults for that workload.

Table 5.2: Heap Size Savings, Execution Degradation and Yield

	xalan	hsqldb	jython	pmd	lusearch	luindex	bloat	antlr	fop
					Savings				
M0	0.0%	7.7%	60.6%	51.4%	59.4%	70.0%	56.7%	62.3%	74.9%
M1	29.7%	15.7%	69.7%	70.6%	65.1%	76.3%	69.1%	78.0%	79.4%
M2	56.0%	38.0%	85.1%	84.6%	81.4%	85.7%	85.7%	85.7%	85.7%
M3	45.7%	28.0%	79.1%	76.9%	74.3%	84.0%	78.2%	84.3%	80.6%
					Degradation				
M0	-1.5%	-1.2%	17.6%	8.6%	20.3%	9.1%	14.5%	3.9%	-2.1%
M1	7.4%	-3.5%	2.5%	7.7%	26.1%	14.7%	12.4%	24.8%	-3.2%
M2	11.2%	50.9%	80.5%	43.7%	225.5%	26.9%	17.7%	31.0%	18.7%
M3	7.6%	17.0%	13.1%	23.2%	66.2%	25.0%	-4.9%	39.4%	10.8%
					Yield				
M0	0.0	-6.6	3.4	6.0	2.9	7.7	3.9	15.8	-36.3
M1	4.0	-4.5	28.4	9.2	2.5	5.2	5.6	3.1	-24.7
M2	5.0	0.7	1.1	1.9	0.4	3.2	4.8	2.8	4.6
M3	6.0	1.7	6.1	3.3	1.1	3.4	-15.8	2.1	7.5

5.1.3 QoE Yield applied to CPU usage

Our system also takes advantage of CPU restriction in a coarse-grained approach. Figure 5.8 shows how nine different Java workloads from the DaCapo benchmarks (the same presented in Table 5.2) react to the deprivation of CPU (in slices of 25%), regarding their total execution time.

Figure 5.8 shows the relative performance slowdown, which represents the *yield* of allocating 75%, 50%, and 25%, comparing with 100% of CPU allocation. Note that, comparing with previous graphics, some applications have longer execution times with 0% CPU taken because they are multithreaded and we used only 1 core for this test. As expected, the execution time grows when more CPU is taken. This enables priority applications (e.g. paying users, priority calculus applications) to run efficiently over our runtime, having the CPU usage transparently restricted and potentially transfered to others (a capability in itself currently unavailable in HLL-VMs).

Figure 5.9 depicts the relative slowdown, or performance degradation, for the previously analyzed workloads. We note that the following applications (`hsqldb`, `fop`, and `antlr`) have yields greater than 1 when CPU restriction is equal or above 50%, as they

Table 5.3: Heap Size Savings, Execution Degradation and Yield

	scimark.fft.large	scimark.sor.large	scimark.sparse.large	scimark.monte_carlo	compress	mpegaudio
			Savings			
M0	1.4%	5.4%	-0.3%	85.7%	51.9%	83.7%
M1	24.9%	18.1%	13.8%	85.7%	67.3%	87.1%
M2	53.0%	45.3%	32.1%	85.7%	78.2%	85.7%
M3	31.5%	46.4%	12.9%	85.7%	75.1%	85.7%
			Degradation			
M0	1.2%	-0.1%	6.4%	0.3%	4.2%	6.7%
M1	2.7%	-0.1%	-2.6%	0.8%	-1.9%	-0.1%
M2	-13.0%	-2.9%	-2.7%	-0.2%	-16.2%	2.4%
M3	-11.3%	-6.7%	-2.4%	1.5%	-7.8%	3.0%
			Yield			
M0	1.2	-89.6	0.0	280.8	12.5	12.4
M1	9.4	-297.1	-5.3	106.3	-36.4	-649.2
M2	-4.1	-15.9	-11.9	-393.2	-4.8	35.2
M3	-2.8	-6.9	-5.4	57.8	-9.6	28.7

stay below the neutral efficiency line in Figure 5.9, due to memory or I/O contention. This means that these are applications that, when a certain amount of CPU is taken (e.g. 50%), their performance degradation is not proportional to this restriction. So, they are good candidates to apply CPU restriction when co-located with higher priority workloads. This is more clearly visible in Figure 5.10, where we can see those applications having relative efficiency gains when resources are partially restricted (the four series represent the proportion of the restriction), as well as those that lose relative efficiency, but always to a smaller extent.

5.2 Resource consumption constraints

In this section, we analyze the impact on evaluating *constraints* (i.e., rules regulating resource usage) during regular VM operations.

Therefore, we conducted a series of tests, measuring different aspects of a running application: i) the overhead introduced in the consumption of a specific resource, and ii) policy evaluation in a complete benchmark scenario. All these evaluations are made locally in a single modified Jikes RVM (version 3.1.1), compiled with the `production` profile.[4]

[4]It includes a two-generation garbage collector [Blackburn and McKinley, 2008] and the optimized

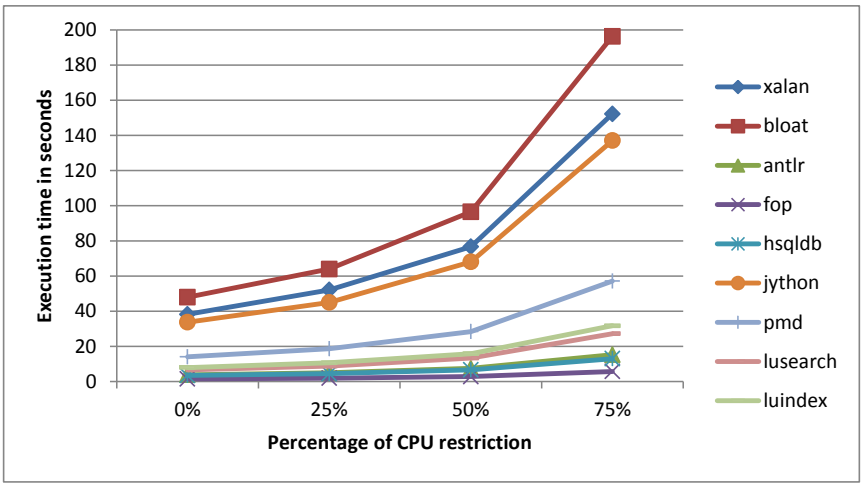


Figure 5.8: Effects of restraining CPU by 25%, 50% and 75%

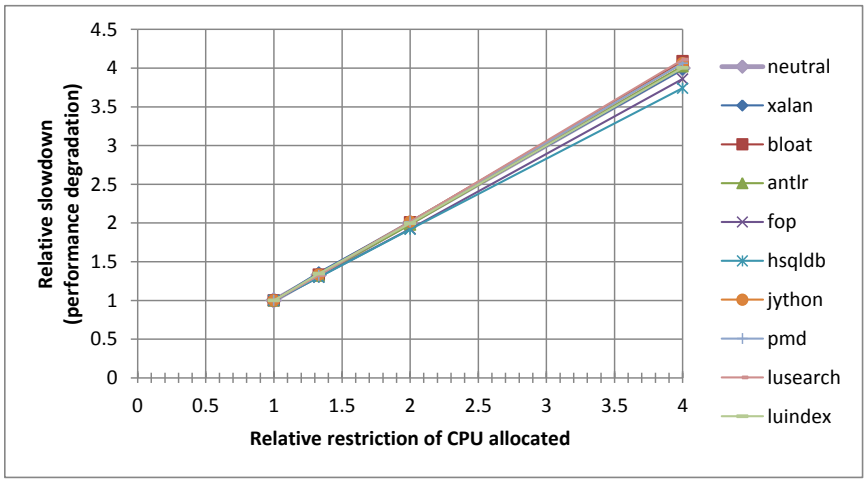


Figure 5.9: Relative slowdown

In Figure 5.11.a, we can observe the evolution of the overhead introduced in thread creation, by measuring average thread creation and start time, as the policy engine has increasingly larger numbers of rules to evaluate, up to 250 (simulating a highly complex policy). The graph shows that this overhead, while increasing, does not hinder scalability, as it is very small. When evaluating 50 constrains (which would correspond to a more reasonable but still complex policy), each check of a certain limit having been reached, the increment is 27 microseconds in each thread creation, which represents an increase of $\approx 6\%$ to the baseline thread creation time (i.e., no constraints evaluated).

and adaptive compilation system.

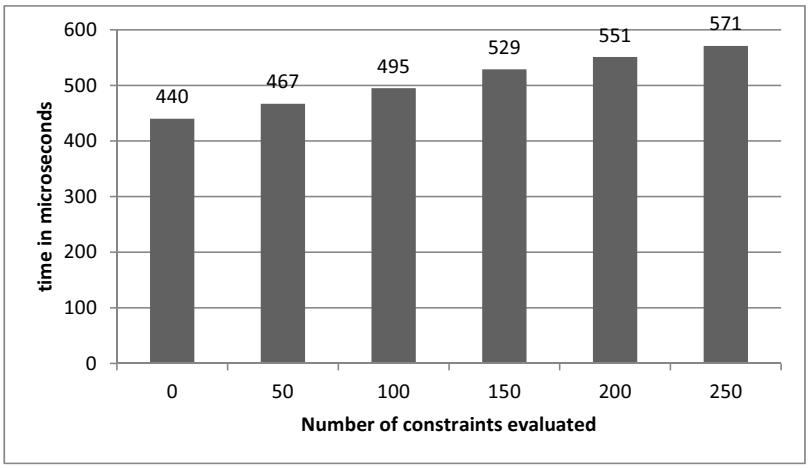
Figure 5.10: Relative efficiency

In Figure 5.11.b, we evaluate whether resource monitoring and policy evaluation (with 300 constraints) introduce any kind of performance degradation as more and more threads are created, resources consumed. Figure 5.11.b clearly shows (omitting Garbage Collection spikes) that thread creation time does not degrade during application execution; although subject to some variation, it presents no lasting degradation.

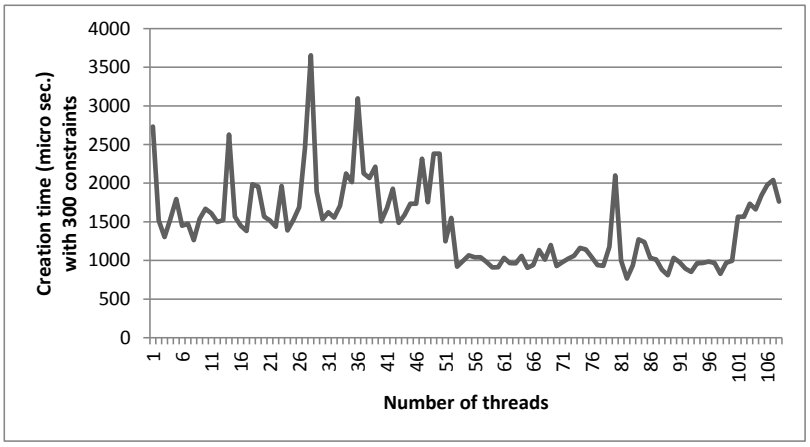
The previous results were obtained monitoring only a single resource, i.e., number of application threads. For other counted resources, e.g. number of bytes sent and received, similar results are expected. Although the allocation of new objects can also be seen as a counted resource, e.g. number of bytes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

Figure 5.12 presents the duration of each GC cycle during the execution of DaCapo's benchmark `lusearch`, with and without evaluating constraints on heap consumption (i.e., RAMM enabled and disabled). The `lusearch` benchmark was configured with a `small` data set, one thread for each available processor (i.e., four threads) and the convergence option active, resulting in some extra warm up runs before the final evaluation.

Because of the generational garbage collection algorithm used in our modified Jikes RVM, we can observe many small collection cycles, interleaved with some full heap transversal and defragmentation operations. The two runs share approximately the same average execution time and a similar standard deviation: $1.38 \pm 0.31ms$ and $1.38 \pm$



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



(b) Thread creation time during execution with 300 constraints (GC spikes omitted)

Figure 5.11: Policy evaluation cost

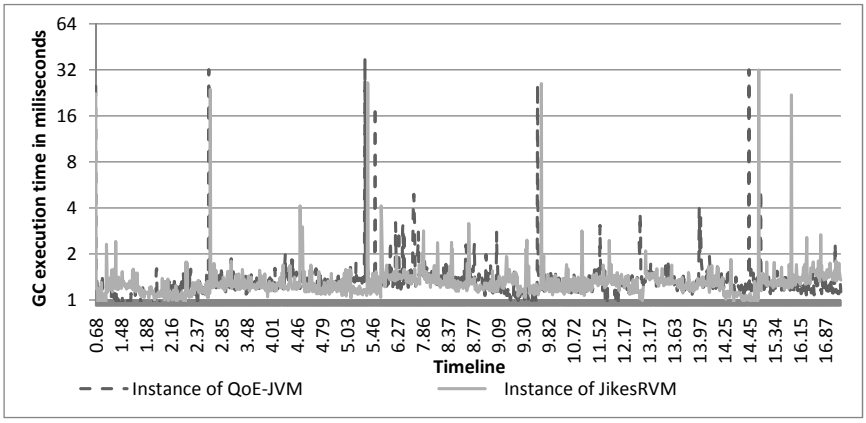


Figure 5.12: GC execution time during Dacapo's LuSearch benchmarck

$0.27ms$, where the former value is when the RAMM module is enabled and the last when RAMM is disabled. With these results we conclude that performance of object allocation and garbage collection is not diminished with the extra work introduced.

To conclude the evaluation of the RAMM module, we stressed an instance of our resource-aware VM with four macro benchmarks, as presented in Figure 5.13. These four benchmarks are multi-threaded applications, allowing us to do a macro evaluation of the proposed modifications. During the execution of these benchmarks there were three resources being monitored: the number of threads, the total allocated memory, and CPU usage. The constraints used in evaluation did not constrain the usage of resources so that the benchmarks could properly assess the impact of monitoring different resources simultaneously in real applications (as opposed to the specific benchmarks presented previously in Figure 5.11). The results show only a negligible overhead: 3% on average.

5.3 Fine-grained progress accounting

In this section, we evaluate our progress framework using both synthetic and real e-Science related workloads. The evaluation is divided into three parts: First, we evaluate the overhead of instrumenting classes at load time. Second, we show the overhead of monitoring progress during application execution. Finally, we show results regarding the experiments with the reallocation of resources.

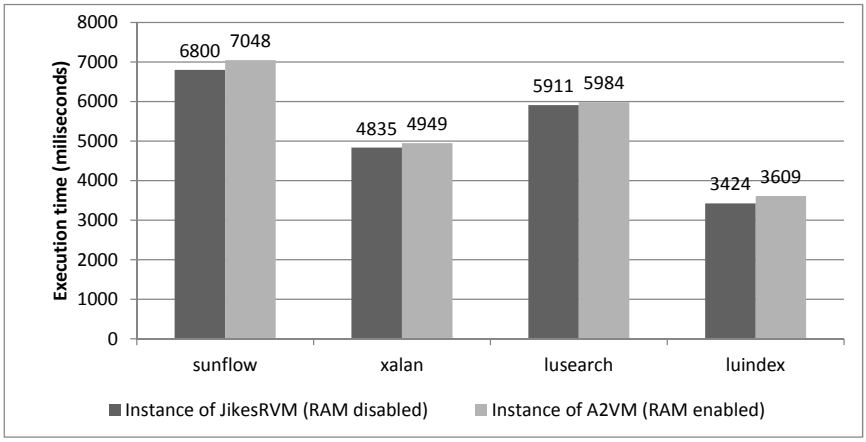


Figure 5.13: Four Dacapo's multi threaded benchmarks with RAMM enabled and disabled

To evaluate the load time overhead, we used the SunFlow render system.[5] SunFlow uses ray tracing and projection techniques which are common in other applications for scientific visualization of data such as biological sequences (e.g. molecules, genes). For this test case, SunFlow is used without any source code modification. The Java instrumentation agent adds $105ms$ to the total time of the application. For the example file used, the rendering process had to load 137 classes which corresponds to an average overhead of $0.76ms$ for each class. This is a very small startup cost, even more so for such a long running application.

To assess the overhead of measuring progress during the application runtime we have setup two scenarios. One aims to micro-benchmark the time taken to call an instrumented method and to execute a write operation on an annotated field. The second scenario's goal is to measure the overhead of using this framework in a complete, real-world application.

In the first scenario, we made a synthetic application in which there is a method performing 1000 write operations on a field. If only the method is annotated, the total execution time is $1ms$, which compares with $45ms$, when the field is annotated, and so, each write operation is instrumented.

In the second scenario we made a single line modification to the SunFlow render system. Sunflow splits the rendering space into spaces called *buckets*. We added an annotation to a method called for each *bucket* that finished rendering. When running

[5] `http://sunflow.sourceforge.net/`, visited July 4, 2014

the original code base, the rendering of a simple scene with a resolution of 1920x1080 took 213 sec. When running SunFlow with QoE-JVM and using the instrumentation described above, it took $214sec.$ (based on an average of 3 executions), which corresponds to an overhead of less than 0.5%. Because the instrumentation code is always the same, regardless of the application, the larger the workload, the smaller the percentage overhead added to the rendering process.

To evaluate how the resource allocation scheduling influences workloads, we used three Java open source e-Science related applications/libraries: SunFlow, Xalan, and Lucene.

SunFlow, a photo-realistic rendering system, with a ray tracing core, was already presented. Xalan is an eXtensible Stylesheet Language for Transformation (XSLT) processor, which transforms XML documents into other formats such as HTML or XMLs with different schemas.[6] It implements the XPath W3C standard for addressing parts of an XML document.[7] Frequently, scientists use well-known information tools, such as ontologies, to model interactions between their elements of study. These ontologies are usually represented in XML documents.[8] Transformations and queries on these documents can be made using Xalan or similar libraries.

The Lucene library provides a set of classes for documents indexing (*luindex*) and high-speed search (*luseach*).[9] This is an important feature in a biologic sequences database. Sequences of genetic material are usually represented in a text/readable format (e.g. FASTA format is a text-based format for encoding DNA or protein sequences sequences), that often need to be updated with new samples and searched.

To understand how progress evolves with the allocation of different resources, SunFlow was used to render a 1920x1080 image, using 2040 buckets. Xalan converted a XML file using a complex XSL specification with 53 transformations. Lucene was used to generate index text files containing the ancestral sequences the Pongo abelii (an orangutan). Figure 5.14 shows the average call rate (within windows of 5 seconds) of the three workloads varies using a different number of cores, while Figure 5.15 shows the results for the same metric but with different heap sizes.

We can conclude that with call rates interpreted as measuring different progress by

[6] `http://www.w3.org/TR/xslt`, visited August 12, 2014
[7] `http://www.w3.org/TR/xpath/`, visited August 12, 2014
[8] http://www.w3.org/TR/owl2-overview/, visited August 12, 2014
[9] `https://lucene.apache.org/core/`, visited August 12, 2014

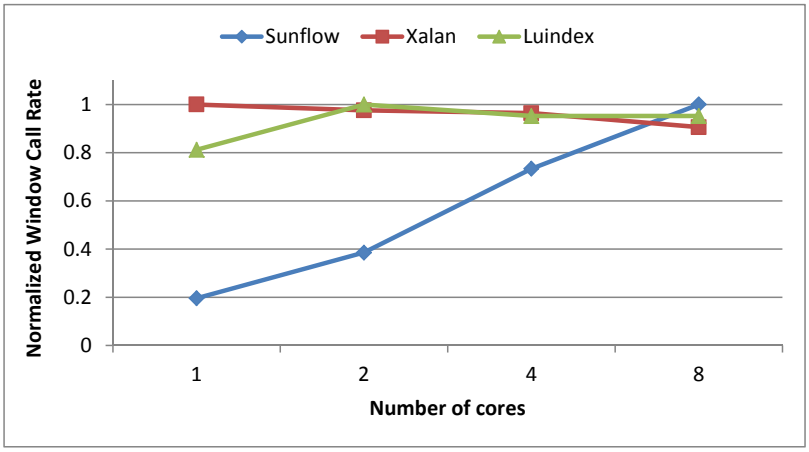


Figure 5.14: Average window call rate for periods of 5 seconds and using a different number of cores

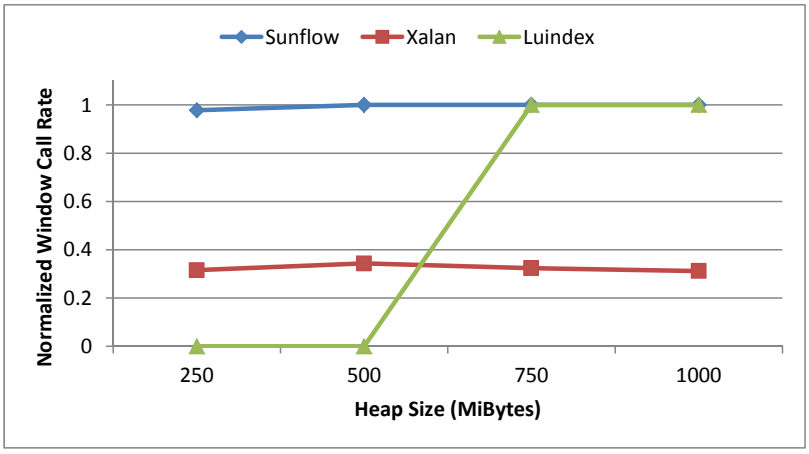


Figure 5.15: Average window call rate for periods of 5 seconds and using a different heap sizes

applications, they are in line with our predictions. While obviously all applications welcome having more resources, they not always take effective advantage from them, more so if we analyze the ratio between additional progress achieved derived from the additional resources. For instance, Lucene mostly always improves performance with any amount of extra resources, while SunFlow is more affected by CPU. Xalan progress is almost immune to extra resources, effectively wasting them, and those could be handed over to other applications.

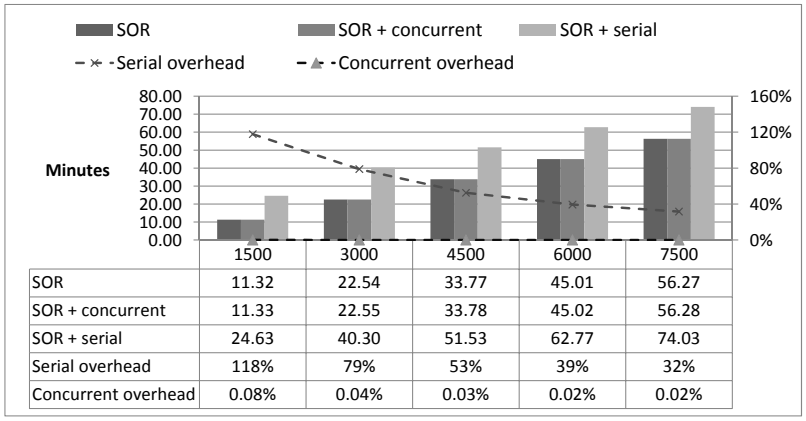
5.4 Concurrent checkpoint

To evaluate concurrent checkpointing specifically, we used Successive Over-relaxation (SOR), a method for solving a linear system of equations. It can have a long execution time and uses raw calculus primitives, necessary to a large set of computer supported research domains. The calculus is supported by a matrix of integers, whose size varies based on the number of equations there is to solve.

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

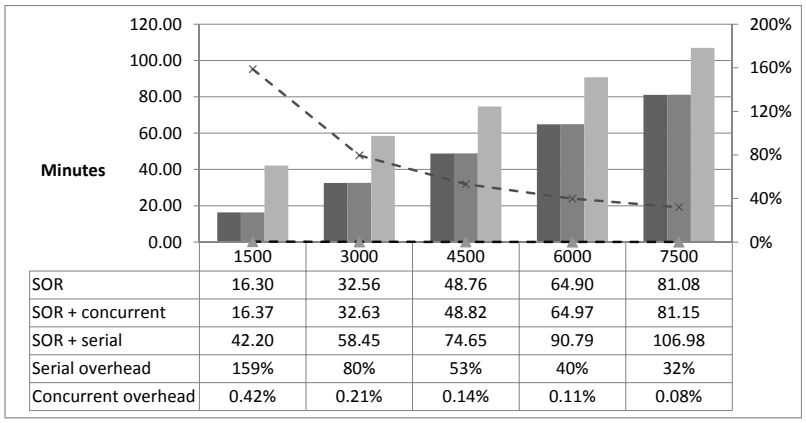
The distinguishing factor between these two types of tests is the event or reason triggering each checkpoint. In *Test 1*, the checkpoint is done when a percentage of the work is completed. In Figure 5.16, checkpointing is done at 20%, 40%, 60%, and 80% of computation progress. We compare SOR's total execution time without any checkpointing (series SOR), with the concurrent checkpoint (series SOR+concurrent) and with the baseline checkpointing mechanisms (series SOR+serial) which can only be done while the application is fully stopped. The left y-axis depicts the total minutes elapsed in each scenario, while the right y-axis plots the serial and concurrent overhead series.

From this, data we conclude that i) the overhead of concurrent checkpointing is negligible - less than 1% in all configurations, and ii) the overhead of the serial check-



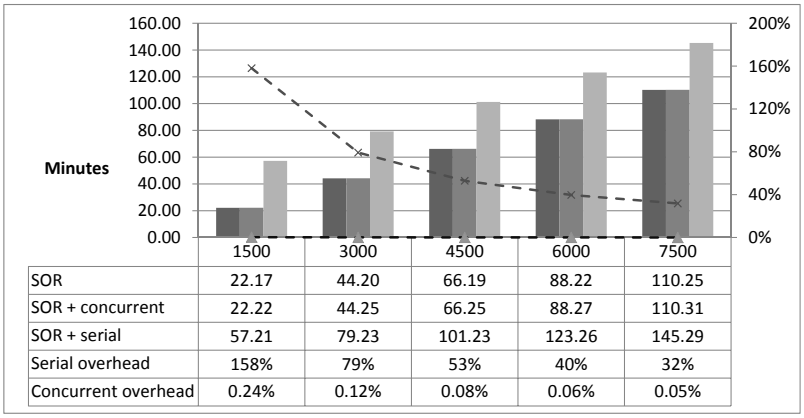
	1500	3000	4500	6000	7500
SOR	11.32	22.54	33.77	45.01	56.27
SOR + concurrent	11.33	22.55	33.78	45.02	56.28
SOR + serial	24.63	40.30	51.53	62.77	74.03
Serial overhead	118%	79%	53%	39%	32%
Concurrent overhead	0.08%	0.04%	0.03%	0.02%	0.02%

(a) Test 1 - 3000 equations



	1500	3000	4500	6000	7500
SOR	16.30	32.56	48.76	64.90	81.08
SOR + concurrent	16.37	32.63	48.82	64.97	81.15
SOR + serial	42.20	58.45	74.65	90.79	106.98
Serial overhead	159%	80%	53%	40%	32%
Concurrent overhead	0.42%	0.21%	0.14%	0.11%	0.08%

(b) Test 1 - 3600 equations



	1500	3000	4500	6000	7500
SOR	22.17	44.20	66.19	88.22	110.25
SOR + concurrent	22.22	44.25	66.25	88.27	110.31
SOR + serial	57.21	79.23	101.23	123.26	145.29
Serial overhead	158%	79%	53%	40%	32%
Concurrent overhead	0.24%	0.12%	0.08%	0.06%	0.05%

(c) Test 1 - 4200 equations

Figure 5.16: Checkpointing experiments triggered on percentage of computation

point has a decreasing impact on the application's execution time, as the number of total iterations increases. This evident decrease is due to the fact that, as computation time increases, the fixed number of serial checkpoints taken (4) will have progressively smaller impact on the total execution time.

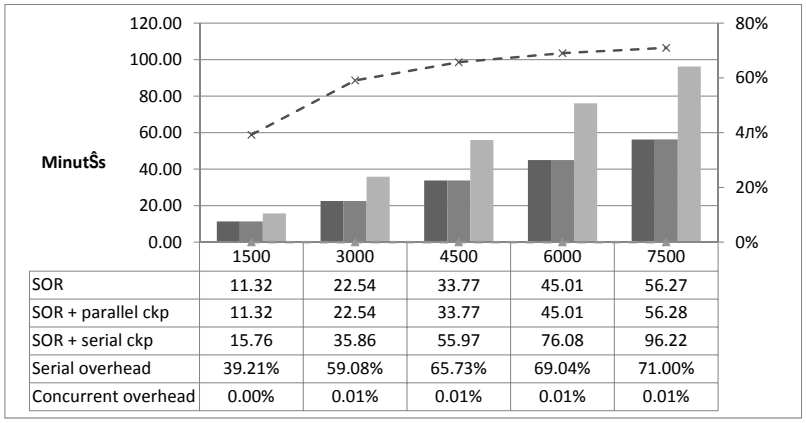
Nevertheless, as the application's total execution time increases, triggering checkpointing based on progress may lead, in case of a failure, to significant loss of work and data (i.e., all the computation done since the previous checkpoint and its results). Furthermore, the percentage of progress may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

To avoid all this, checkpointing could be triggered whenever a given time has elapsed. In the experiments we considered that checkpointing was done approximately every 5 minutes. This results in doing 2 checkpoints when 3000 equations are considered and 4 checkpoints when 4200 equations are considered. The described scenario is represented by *Test 2*. Figures 5.17 presents the results. Here, since longer executions imply more checkpoints taken (with 5 minute periodicity), the serial checkpoint now increasingly stretches the total execution time of the application (up to 70% more, broadly), while the overhead introduced by the concurrent checkpoint always remains very low.

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

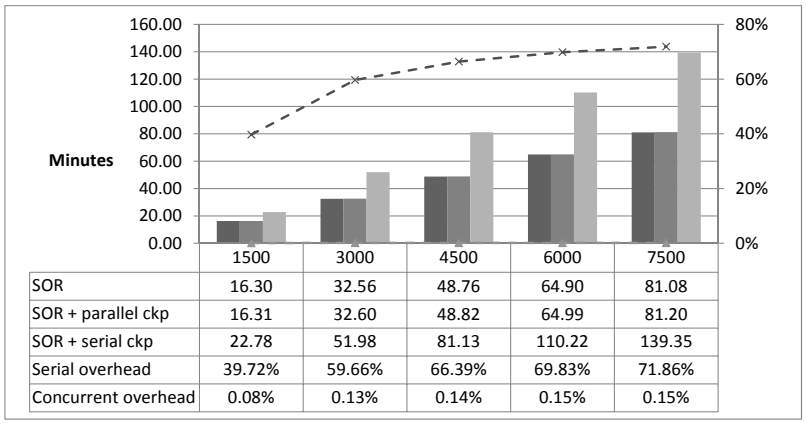
Summary

In this chapter, we have demonstrated the potential of the three major mechanisms to manage resources in a cluster where several instances of an HLL-VM run in competition for a limited number of resources. These mechanisms required extensions to some subsystems of the Jikes RVM, such as the heap resizing mechanism, threading and the base class library. However, they are in general simple and portable to other managed runtimes. Some impose a degradation penalty to the execution time of applications, as in



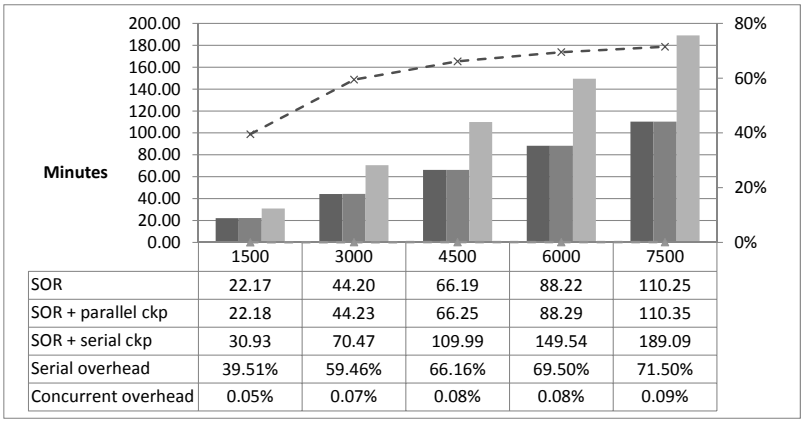
	1500	3000	4500	6000	7500
SOR	11.32	22.54	33.77	45.01	56.27
SOR + parallel ckp	11.32	22.54	33.77	45.01	56.28
SOR + serial ckp	15.76	35.86	55.97	76.08	96.22
Serial overhead	39.21%	59.08%	65.73%	69.04%	71.00%
Concurrent overhead	0.00%	0.01%	0.01%	0.01%	0.01%

(a) Test 2 - 3000 equations



	1500	3000	4500	6000	7500
SOR	16.30	32.56	48.76	64.90	81.08
SOR + parallel ckp	16.31	32.60	48.82	64.99	81.20
SOR + serial ckp	22.78	51.98	81.13	110.22	139.35
Serial overhead	39.72%	59.66%	66.39%	69.83%	71.86%
Concurrent overhead	0.08%	0.13%	0.14%	0.15%	0.15%

(b) Test 2 - 3600 equations



	1500	3000	4500	6000	7500
SOR	22.17	44.20	66.19	88.22	110.25
SOR + parallel ckp	22.18	44.23	66.25	88.29	110.35
SOR + serial ckp	30.93	70.47	109.99	149.54	189.09
Serial overhead	39.51%	59.46%	66.16%	69.50%	71.50%
Concurrent overhead	0.05%	0.07%	0.08%	0.08%	0.09%

(c) Test 2 - 4200 equations

Figure 5.17: Checkpointing experiments with checkpoint triggered by time elapsed

the case of the resource consumptions constraints. This penalty is, in most cases, small and introduces the possibility for the provider to distribute resources in an application-driven way.

Part III

Allocation and Scheduling in Infrastructure-as-a-Service

6 Architecture and Cost Model

Contents

Chapter overiew

Cloud SLAs compensate customers with credits when average availability drops below certain levels. This is too inflexible because consumers lose non-measurable amounts of performance being only compensated later, in upcoming billing cycles. We propose to schedule virtual machines (VMs), driven by range-based non-linear reductions of utility, taking into account user classes and different ranges of resource allocations: partial utility [Simão and Veiga, 2013b, 2014]. This customer-defined metric, allows providers to transfer resources between VMs in meaningful and economically efficient ways.

We define a comprehensive cost model incorporating the partial utility reported by clients to a certain level of degradation, when VMs are allocated in overcommitted environments (Public, Private, Community Clouds). CloudSim was extended to support our scheduling model. Several simulation scenarios with synthetic and real workloads are presented, using datacenters with different dimensions, regarding the number of servers and computational capacity. We show the partial utility-driven scheduling allows more VMs to be allocated. It brings benefits to providers, regarding revenue and resource utilization, allowing for more revenue per resource allocated, and scaling well with the size of datacenters, when comparing with a utility-oblivious redistribution of resources. Regarding clients, their workloads' execution time is also improved, by incorporating an SLA-based redistribution of their VM's computational power.

This chapter is organized in three main sections. It starts by introducing the motivation for scheduling in IaaS overcommited deployment. Section 6.2 presents some related work on the topic of VM scheduling. Section 6.3 discusses a comprehensive cost model that incorporates the partial utility the client specifies for a certain level of depreciation when VMs are allocated in an overcommitted environment.

6.1 Introduction

Currently, cloud providers provide a resource selection interface based on abstract computational units (e.g. EC2 computational unit). This business model is known as Infrastructure-as-a-Service (IaaS). Cloud users rent computational units taking into account the estimated peak usage of their workloads. To accommodate this simplistic interface, cloud providers have to deal with massive hardware deployments, and all the management and environmental costs that are inherent to such a solution. These costs will eventually be reflected in the price of each computational unit.

Today, cloud providers' SLAs already establish some compensation in consumption credits when *availability*, or uptime, fall below a certain threshold.[1] The problem with *availability* is that, from a quantitative point of view, it is often equivalent to all-or-nothing, i.e., either availability level fulfills the agreed uptime or not. Even so, to get their compensation credits, users have to fill a form and wait for the next billing cycle.

Some argue that, although virtualization brings key benefits for organizations, mi-

[1] `http://aws.amazon.com/ec2-sla/`, visited July 4, 2014

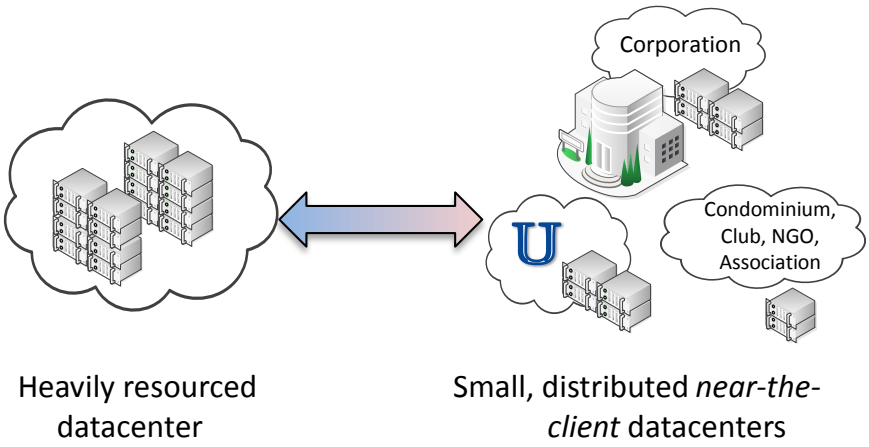


Figure 6.1: Cloud deployments: From heavy clouds to small, geo-distributed near-the-client datacenters

grating all to a public cloud is sometimes not the better option.[2] A middle ground approach is to deploy workloads in a private (or hybrid) cloud. Doing so has the potential to limit costs on a foreseeable future and, also important, keeps private data in-premises. Others propose to bring private clouds even closer to users, to provide a more environmentally reasonable, or cheaper to cool and operate, cluster [Liu et al., 2011; Khan et al., 2013a].

6.1.1 Overcommitted environments

Figure 6.1 shows what means to bring the cloud closer to the user. Small, geo-distributed near-the-client datacenters (private, shared) save money, the environment, and reduce latency by keeping data on premises. This kind of vision is sometimes referred as Community Cloud Computing (C3) [Marinos and Briscoe, 2007], which can take advantage of previous research in peer-to-peer and grid systems [Silva et al., 2010b]. Nevertheless, many of the fundamental research and the technological deployments are yet to be explored.

From a resource management point of view, these new approaches highlight two issues. In the one hand, the deployment sites are more lightly resourced [Saovapakhiran and Devetsikiotis, 2011; Khan et al., 2014], either because the hardware is intrinsically

[2]Adopt the cloud, lose money. Virtualize your datacenter instead. `http://www.theregister.co.uk/2009/04/15/mckinsey_cloud_report/`, visited August 5, 2013

less powerful or the hardware layer is made of unused parts of deployments already used for other tasks. Thus, overcommitment, which is commonly used in virtualized environments [Waldspurger, 2002; Beloglazov and Buyya, 2012; Agmon Ben-Yehuda et al., 2014b], will become more frequent. Techniques such as dynamic resource allocation and accurate cost modeling must be researched to manage this kind of clouds. Because of the federated and low-cost nature, overcommitment of resources is perhaps a more common (and needed) scenario than in public clouds. Second, in such environments there will be many classes of users which, in most cases, are willing to trade the performance of their workloads for a lower (or even free) usage cost.

In a public cloud, overcommitment can be used to reduce the number of machines requiring power, when aiming to reduce energy consumption [Beloglazov and Buyya, 2012; Mastroianni et al., 2013]. In private clouds, given the potential physical resource scarcity, the problem is even more critical. To overcommit with minimal impact on performance and maximum cost-benefit ratio, cloud providers need to related how the partial release of resources will impact on the workload performance and user satisfaction. While users can easily decide about their relative satisfaction in the presence of resource degradation, they cannot easily determine how their workloads react to events such as peak demands, hardware failures, or any reconfiguration in general.

As private clouds become more frequent in medium and large scale organizations, it is necessary to promote a fair use of the available resources. Usually, these organizations consist of several departments, working on different projects. Each project has a budget to rent computational shared resources. For example, Intel owns a distributed compute farm that is used for running its massive chip-simulation workloads [Ohad Shai, 2013]. The various Intel projects that need to use the infrastructure, *purchase* different amount of servers. Also in this context, it is relevant to know how each department values or prioritizes each of its workloads, which will influence the *price* they are willing to pay for the execution environment.

All-or-nothing resource allocation is not flexible enough for these multi-tenant multi-typed user environments, especially when users may not know exactly how many resources are actually required. While no one complaints because there is no real market, it does not mean there is no room for improvements in more flexible pricing models that can foster competition and entry of smaller players. Like other telecommunications and commodity markets before, such as electricity, the still emergent Cloud market is still seen by some as an oligopoly (hence, not a real market with an even playing field) be-

cause it still lacks a large number of suppliers [Jin et al., 2014]. From the provider's or owner point of view, this is important if there can be cost reductions and/or there are environmental gains by restricting resources, which will still be more favorable than simply delaying or queuing their workloads as a whole.

Both memory and CPU/cores [Zhang et al., 2005; Agmon Ben-Yehuda et al., 2014b; Gong et al., 2010] are common targets of overcommitment. The two major approaches consist of adapting the resources based on current observation of the system performance or using predictive methods that estimate the best resource allocation in the future based on past observations. Others incorporate explicit or implicit risk-based QoS requirements and try do decide which requested VMs should be favored but depend on non-deterministic parameters (e.g. client willing to pay) and make uncommon assumptions about the requested VM characteristics (e.g. homogeneous types) [Macias and Guitart, 2014; Morshedlou and Meybodi, 2014]. Moreover, they do not consider the *partial utility* of applying resource allocation, i.e., that reducing shares equally or in equal proportion may not yield the best overall result.

6.1.2 Scheduling Based on Partial-Utility

In this work, we propose to schedule CPU processing capacity to VMs (the isolation unit of IaaS) using an algorithm that strives to account for user's and provider's potentially opposing interests. While users want their workloads to complete with maximum performance and minimal cost, providers will eventually need to consolidate workloads, overcommitting resources and, thus, inevitably degrading the performance of some of them.

The proposed strategy operates when new VM requests are made to the provider, and takes the user's partial utility specification, which relates the user's satisfaction for a given amount of resources, and correlates it with the provider's analysis of the workload progress given the resources applied. This gives an operational interval which the provider can use to maximize user satisfaction and the need to save resources. Resources can be taken from workloads that use them poorly, or that do not mind having an agreed performance degradation (and, thus, pay less for the service), and assign them to workloads that can used them better, or belong to users with a more demanding satisfaction rate (and, thus, are willing to pay more).

We have implemented our algorithm as an extension to scheduling policies of a state

of the art cloud infrastructures simulator, CloudSim [Beloglazov and Buyya, 2012; Calheiros et al., 2011]. After extensive simulations using synthetic and real workloads, the results are encouraging and show that resources can be taken from workloads, while improving global utility of the user's renting cost and of the provider's infrastructure management.

In summary, the contributions presented in the following chapters are:

- An architectural extension to the current relation between cloud users and providers, particularly useful for private and hybrid cloud deployments;
- A cost model which takes into account the clients' partial utility of having their VMs release resources when in overcommitment;
- Strategies to determine, in a overcommitment scenario, the best distribution of workloads (from different classes of users) among VMs with different execution capacities, aiming to maximize the overall utility of the allocation;
- A comparison with a comprehensive list of utility-oblivious algorithms;
- Extensions to a state of the art cloud simulator;
- Implementation and evaluation of the cost model in the extended simulator using a large set of datacenter configurations.

The next section gives a short survey of works that make allocation decisions to comply with previously negotiated level of service with the client. In general, these works use the scheduling mechanisms presented in Chapter 2. Then, Section 6.3 details our utility model, which underlies the scheduling strategies presented in Chapter 7.

6.2 Related Work

With the advent of Cloud Computing, particularly with the Infrastructure-as-a-Service business model, resource scheduling in virtualized environments received prominent attention from the research community [Vaquero et al., 2011; Buyya et al., 2011; Simão and Veiga, 2012a; Ishakian et al., 2012; Dawoud et al., 2012] addressed, as either a resource management, or a fair allocation challenge. At the same time, the research community has built simulation environments to more realistically explore new strategies

while making a significant contribution to *repeatable science* [Malik et al., 2013; Calheiros et al., 2011; Beloglazov and Buyya, 2012].

The management of virtual machines, and particularly their assignment to the execution of different workloads, is a critical operation in these infrastructures. Although virtual machine monitors provide the necessary mechanisms to determine how resources are shared, finding an efficiency balance of allocations, for the customer and the provider, is a non-trivial task. In recent years, a significant amount of effort has been devoted to investigate new mechanisms and allocation strategies, aiming to improve the efficiency of Infrastructure-as-a-Service datacenters. Improvements to allocation mechanisms at the hypervisor level, or in an application-agnostic way, aim to make a fair distribution of available resources to the several virtual machines running on top of an hypervisor, with intervention over CPU, memory shares, or I/O-related mechanisms [Waldspurger, 2002].

We can organize this research space in two main categories: i) scheduling with energy awareness, which is usually transparent to the client; ii) scheduling with negotiated service-level objectives, which has implications in client and provider goals. In this work we focus on the second category, but both topics can benefit by our approach. The following is a briefly survey of these two areas.

6.2.1 Scheduling with Energy Awareness

A low-level energy-aware hypervisor scheduler is proposed in [Kim et al., 2014]. The scheduler takes into account the energy consumption, measured based on in-processor events. It considers the dynamic power, which can change with different scheduling decisions (unlike leakage power which is always constant).

A common approach is to use dynamic voltage and frequency scaling (DVFS). Typically, a globally underloaded system will have its frequency reduced. But this will have a negative and unpredictable impact on other VMs that, although having a smaller share of the system, are using it fully. To avoid inflicting performance penalties on these VMs, recent work [Hagimont et al., 2013] proposes extensions to the credit scheduler so that the allocated share of CPU to these smaller but overloaded VMs remains proportionally the same after the adjustment. Nevertheless, recent findings [Le Sueur and Heiser, 2010] show that frequency scaling and dynamic voltage have a small contribution on

the reduction of energy consumption. Instead, systems based on modern commodity hardware should favor the idle state.

Others determine which is the minimum number of servers that needs to be active in order to fulfill the workload's demand, without breaking the service level objectives [Meng et al., 2010; Beloglazov and Buyya, 2012; Mastroianni et al., 2013]. Meng et al. [Meng et al., 2010] determine which are the best VM pairs to be co-located based on their past resource demand. Given historic workload timeseries and an SLA-based characterization of the VM's demand, they determine the number of servers that need to be used for a given set of VMs. Beloglazov et al. [Beloglazov and Buyya, 2012] detect over and under utilization peaks, and migrate VMs between hosts to minimize the power consumption inside the datacenter. Mastroianni et al. [Mastroianni et al., 2013] have similar goals with their ecoCloud, but use a probabilistic process to determine the consolidation of VMs.

These solutions usually impose constraints on the number of VMs that can be co-located and do not use client's utility to drive allocation, missing the opportunity to explore combinations with advantage to both parties, provider and clients, that is, higher revenue per resource (which is in the provider's interest) and more progress for the price payed (which is in the clients' interest).

6.2.2 Scheduling with Service-Level Objectives

Clouds inherit Grid's potential for resource sharing and pooling due to their inherent multi-tenancy support. In Grids, resource allocation and scheduling can be performed mostly based on initially predefined, a priori and static, job requirements [Silva et al., 2011]. In clouds, resource allocation can also be changed elastically (up or down) at runtime in order to meet the application's load and effective needs at each time, improving flexibility and resource usage.

To avoid strict service level objectives violations, the main research works can be framed into three methods: i) statistical learning and prediction [Gong et al., 2010], ii) linear optimization methods [Hinesa et al., 2011], iii) and economic-oriented strategies.

Resource management can also be based on microeconomic game theory models, mostly in two directions: i) forecast in the number of virtual machines (or their characteristics) a given workload will need to operate [Tsakalozos et al., 2011; Mian et al.,

2012] and ii) change allocations at runtime to improve a given metric, such as workload fairness or the provider's energy costs [León and Navarro, 2013; Ishakian et al., 2012]. Auction-based approaches have also been proposed in the context of provisioning VMs [Xu and Li, 2013; S. Zaman, 2011; Andrzejak et al., 2010] when available resources are less abundant than requests. Commercial systems such as the Amazon EC2 Spot Instances have adopted this strategy.

Fewer works consider that the customer accepts a negotiable performance during workload execution. This type of flexibility usually requires the adoption of an economic or cost-theoretical model. Cloudpack [Ishakian et al., 2012] provides support for users to specify workloads in a way they can declare their quantitative resource requirements and temporal flexibilities. S. Costache et al. [Costache et al., 2013] proposes a market where the users bids for a VM with a certain amount of resources. To guarantee a steady amount of resources, their system migrates VMs between different nodes (which has the potential to impose a significant performance penalty [Xu et al., 2014]).

In [Macias and Guitart, 2014], clients choose the SLA based on a class of risk, which has impact on the price the client will pay for the service - the lower the risk, the higher the price. Based on this negotiation, an allocation policy would be used to allocate resources for each user, either minimizing the risk (to the client) or the cost (to the provider). They are, however, unable to explicitly select the VM or set of VMs to degrade. In [Morshedlou and Meybodi, 2014], a method is presented to decide which VMs should release their resources, based on each client's willingness to pay for the service. This approach is similar to our work, but they assume that some amount of SLA violations will occur because they demand the victim VM to release its full resources. They try to minimize the impact on the user's satisfaction, based on a probabilistic metric, decided only by the provider. Moreover, they assume homogeneous VM types and with explicitly assessed different reliability levels, which in uncommon in cloud deployments.

Seokho et al. [Son et al., 2013] focus on datacenters that are distributed across several sites, and use SLAs to distribute the load among them. Their system selects a data center according to a utility function that evaluates the appropriateness of placing a VM. This utility function depends on the distance between the user and the datacenter, together with the expected response time of the workload to be placed. Therefore, a VM request is allocated in the physical machine that is closest to the user and has a recent history of low utilization. For network-bound workloads, their system could integrate our

approach by also considering the partial assignment of resources, eventually exchanging locality (and so, smaller network delays) by, for example, a small delay in the workload finish time.

SageShift [Sukwong et al., 2012] targets the hosting of web services, and uses SLAs to make admission control of new VMs (Sage) based on the expected rate of co-arrival requests. In addition, it presents an extension to hypervisor scheduler (Shift) to control the order of execution of co-located VMs, and minimize the risk of failing to meet the negotiated response time. Also in the case of Sage, no alternative strategy exists when the system detects that a new VM cannot strictly comply with a given SLA.

6.2.3 Flexible SLAs

In summary, our work is the first that we are aware of that clearly accepts, and incorporates in the economic model, the notions of partial utility degradation in the context of VM scheduling in virtualized infrastructures, such as data centers, public, private or hybrid clouds. It demonstrates that it can render benefits for the providers, as well as reduce user dissatisfaction in a structured and principled-based way, instead of the typical all-or-nothing approach of queuing or delaying requests, while still able to prioritize user classes in an SLA-like manner.

6.3 A partial utility cost model for cloud scheduling

Our model uses a non-linear, range-based, reduction of utility that is different for classes of users, and across different ranges of resource allocations that can be applied. We name it partial utility.

To schedule VMs based on the partial utility of the clients, we have to define the several elements that constitute our system model. The provider can offer several categories of virtual machines, more compute or memory optimized. In each category (e.g. compute optimized) we consider that the various VM types are represented by the set $VM_{types} = \{VM_{t_1}, VM_{t_2}, VM_{t_3}, \ldots VM_{t_m}\}$. Elements of this set have a transitive less-than order, where $VM_{t_1} < VM_{t_2}$ iff *VirtualPower*$(VM_{t_1}) <$ *VirtualPower*(VM_{t_2}). The function *VirtualPower* represents the provider's metric to advertise each VM's computational power, along with details about a particular combination of CPU, memory and storage capacity. For example, Amazon EC2 uses the Elastic Computation Unit (ECU)

which is an aggregate metric of several proprietary benchmarks. Another example is the HP Cloud Compute Unit (CCU).

Currently, infrastructure-as-a-service providers rent virtual machines based on pays-as-you-go or pre-reserved instances. In either case, a price for a charging period is established, e.g. $ / hour, for each VM type. This value, determined by the function $Pr(VM_{t_i})$, is the monetary value to pay when a VM of type t_i is not in overcommitment with other VMs (of the same type or not). Considering that, for a given VM instance, vm, the type (i.e., element of the set VM_{types}) is determined by the function $VMType(vm)$, the price is determined by $Pr(VMType(vm))$.

6.3.1 Degradation factor and Partial utility

For each VM, the provider can determine which is the degradation factor, that is, which percentage of the VM's *virtual power* is diminished because of resource sharing and overcommitment with other VMs. For a given VM instance, vm, this is determined by the function $Df(vm)$. In scenarios of overcommitment, described in the previous section, each user can choose which fraction of the price he/she will pay when workloads are executed. When the provider must allocate VMs in overcommitment, the client will see this by having its VMs with fewer resources, resulting in a potentially perceivable degradation of performance of its workload. Thus, overcommitment and the degradation factor refer to same the process but represent either the provider's or the client's view. We will use these expressions interchangeably throughout the following chapters.

When overcommitment must be engaged, the same client will pay as described in Equation 6.1, where the function Pu represents the partial utility that the owner of the VM gives to the degradation. Both these terms are percentage values. Although this equation naturally shares the goals of many SLA-based deployment [Morshedlou and Meybodi, 2014], it takes into account specific aspects of our approach, as it factors the elements taken from the partial utility specification (detailed in the following paragraphs).

$$\begin{aligned} Cost(vm) = {} & Pr(VMType(vm)) \\ & \cdot(1 - Df(vm)) \cdot Pu(Df(vm)) \end{aligned} \tag{6.1}$$

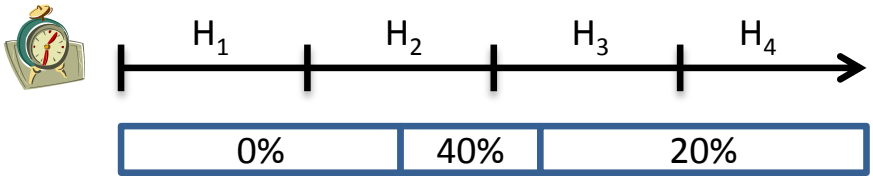


Figure 6.2: Scenario where partial release of resources varies during renting period

For example, if $Df(vm)$ is 20% and $Pu(Df(vm))$ is 100% it means that the client is willing to accept an overcommitment of 20% and still pay a value proportional to the degradation. But if in the same scenario $Pu(Df(vm))$ is 50% it means the client will only pay half of the value resulting from the overcommit, i.e., $Pr(VMType(vm)) \times (1 - 0.2) \times 0.5 = Pr(VMType(vm)) \times 0.4$.

In general, overcommitment can vary during the renting period. During a single hour, which we consider the billing period, a single VM can have more than one degradation factor as depicted in Figure 6.2. In this example, during the first hour no degradation is necessary, while during part of the third and fourth hours, the provider needs to take 20% of the computation power. Thus, because a VM can be hibernated or destroyed by their owners, and new VMs can be requested, Df must also depend on time. To take this into account, $Df_h(vm, i)$ is the i^{th} degradation period of hour h. Jin et al. [Jin et al., 2014] also discuss a fine-grained pricing schema although they focus on the *partial usage waste* problem, which is complementary to the work discussed in this thesis. Ishakian et al. [Ishakian et al., 2012] uses the *epoch* concept, although our approach is more fine-grained and usable in the partial utility model as described next.

Internally, providers will want to control the maximum overcommitment which, on average, is applied to the VMs allocated to a given client and, by extension, to the datacenter as a whole. Equation 6.2 is able to measure this using the Aggregated Degradation Index (ADI) for a generic set of VMs. This metric ranges from 0 (non degraded) to 1 (fully degraded).

$$ADI(VMSet) = 1 - \frac{\sum_{vm \in VMSet} (1 - Df(vm)) \cdot VirtualPower(vm)}{\sum_{vm \in VMSet} VirtualPower(vm)} \tag{6.2}$$

6.3.2 Classes for prices and partial utility

Clients can rent several types of VMs and choose the class associated with each one. Classes have two purposes: the first is to establish a partial utility based on the overcommitment factor; the second is to set the base price for each VM type. Clients, and the VMs they rent, are organized in classes which are represented as a set $C = \{C_1, C_2, C_3, \ldots, C_n\}$. Elements of this set have a transitive less-than order ($<$), where $C_1 < C_2$ iff *base-price*$(C_1) <$ *base-price*(C_2). The function *base-price* represents the base price for each VM type. The class of a given virtual machine instance vm is represented by the function $class(vm)$, while the owner (i.e., the client who is renting the VM) can be determined by $owner(vm)$.

Each class determines, for each overcommitment factor, the partial utility degradation. Because the overcommitment factor can have several values, we define R as a set of ranges: $R = \{]0..0.2[, [0.2..0.4[, [0.4..0.6[, [0.6..0.8[, [0.8..1]\}$. As a result of theses classes, the Pu function must be replaced by one that also takes into account the class of the VM, along with the interval of the overcommitment factor, as presented in definition 6.3. Thus, Pu_{class} is a matrix of partial utilities. Each provider can have a different matrix which it advertises so that clients can choose the best option.

$$Pu_{class} : C \times R \rightarrow [0..1] \tag{6.3}$$

Note that, currently, our model assumes that the partial utility matrix is defined considering the total virtual power of a VM, namely, CPU, memory, and storage capacity. If some overcommitment must be done in any of these dimensions, we consider them equal or simply average them. This value is then used to determine the overall partial utility of the VM's new allocation. However, a more generic (and complex) model could be used, where a matrix like the one defined in Equation 6.3 could be specified for each of the dimensions of the VM. This would result in a vector of partial-utility matrices, whose final value would have to be aggregated to be used in Equation 6.1. This is seen as future work.

The Pr function for each VM must also be extended to take into account the VM's class, in addition to the VM's type. We define a new function, Pr_{class}, as presented in Definition 6.4. Similarly to the matrix of partial utilities, each provider can have a different *price* matrix.

$$Pr_{class} : C \times VM_{types} \rightarrow \mathbb{R} \tag{6.4}$$

In summary, the proposed partial utility model and the associated cost structure is based on three elements: *i)* the base price of each VM type, *ii)* the overcommitment factor, and *iii)* the partial utility degradation class associated with each VM.

6.3.3 Total costs

For a given client, the total cost of renting is simply determined by sum of the costs of renting each VM, as presented in Equation 6.5, where $RentVMs(c)$ represent the VMs rented by client c.

$$RentingCost(c) = \sum_{vm \in RentVMs(c)} VMCost(vm) \tag{6.5}$$

The cost of each VM is presented in Equation 6.6, where N is the number of hours the VM was running, and P the number of overcommitment periods in hour h. If, after allocation, the VM's degradation factor remains constant, the P equals 1.

$$\begin{aligned} VMCost(vm) = \sum_{h=1}^{N} \sum_{p=1}^{P} & \frac{Pr_{class}(class(vm), VMType(vm))}{P} \cdot \\ & \cdot(1 - Df_h(vm, p)) \cdot \\ & \cdot Pu_{class}(class(vm), Df_h(vm, p)) \end{aligned} \tag{6.6}$$

The provider's revenue is given by how much all clients pay for the VMs they rent. The provider wants to maximize the revenue by minimizing the degradation factor imposed to each virtual machine. Because there are several VM classes, each with a particular partial utility for a given degradation factor, the provider's scheduler must find the allocation that maximizes (Equation 6.6). There are different ways to do so, which we analyze in Section 7.1.

6.3.4 Practical scenario

As a practical scenario, we consider that the partial utility model has three classes of users (High, Medium, Low) according to their willingness to relinquish resources in exchange for a lower cost. More classes could be added (these three are illustrative):

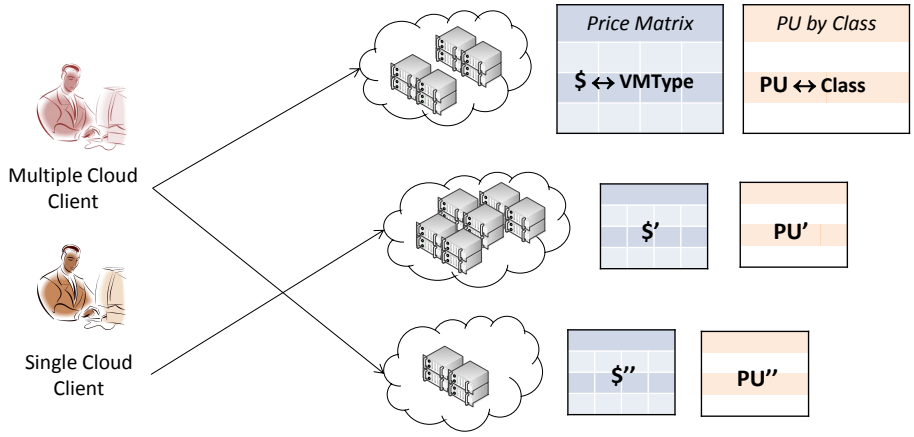


Figure 6.3: A practical scenario of using flexible SLAs in a market-oriented environment

- **High**: users with more stringent requirements, deadlines, and that are willing to pay more for a higher performance assurance but, in exchange, demand to be compensated if those are not met. Compensation may include, not simply refund, but also some level of significant penalization;
- **Medium**: users who are willing to pay, but will accept running their workloads in VMs with less resources for the sake of lower costs, and for other externalities, such as reduced carbon footprint impact, but have some level of expectation on execution time, and;
- **Low**: users who do not mind waiting for their workloads to complete if they pay less;

Partial utility profiles could also be organized around cloud providers, and assume that each provider would be specialized in a given profile. For example, *flexible* would represent shared infrastructures with no obligations, and many well-dimensioned private clouds; *business*, public clouds or high-load private or hybrid clouds; *critical*, clouds where budgets and deadlines of workloads are of high relevance, and penalties are relevant; *SLA-Oriented*, top scenario where penalties should be avoided at all cost. For simplicity we focus on a single cloud provider that supports several classes of partial utility which clients can choose when renting VMs, as illustrated in Figure 6.3.

For the three classes of our example, the cloud provider can define a partial utility matrix, represented by M (Equation 6.7). This matrix defines a profile of partial utility

for each level of resource degradation (resources released) that can be used to compare strictness or flexibility of the proposed resource management.

$$M = \begin{array}{c} \\ [0..0.2[\\ [0.2..0.4[\\ [0.4..0.6[\\ [0.6..0.8[\\ [0.8..1[\end{array} \begin{array}{c} \begin{array}{ccc} High & Medium & Low \end{array} \\ \begin{pmatrix} 1.0 & 1.0 & 1.0 \\ 0.8 & 1.0 & 1.0 \\ 0.6 & 0.8 & 0.9 \\ 0.2 & 0.6 & 0.8 \\ 0.0 & 0.4 & 0.6 \end{pmatrix} \end{array} \tag{6.7}$$

The provider must also advertise the base price for each type of VM. We assume there are four types of virtual machines with increasing *virtual power*, for example, **micro**, **small**, **regular**, and **extra**. Matrix P (Equation 6.8) determines the base price ($/hour) for these types of VMs.

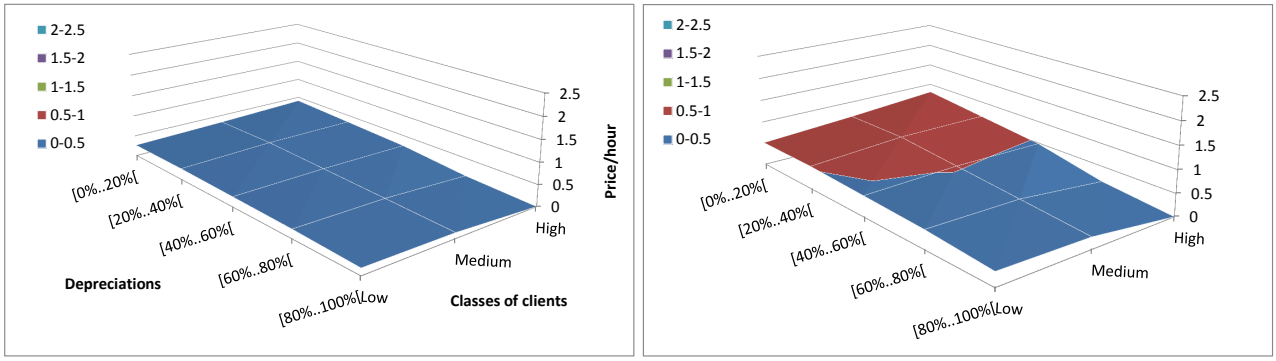
$$P = \begin{array}{c} \\ micro \\ small \\ regular \\ extra \end{array} \begin{array}{c} \begin{array}{ccc} High & Medium & Low \end{array} \\ \begin{pmatrix} 0.40 & 0.32 & 0.26 \\ 0.80 & 0.64 & 0.51 \\ 1.60 & 1.28 & 1.02 \\ 2.40 & 1.92 & 1.54 \end{pmatrix} \end{array} \tag{6.8}$$

6.3.5 Comparing flexible pricing profiles in a cloud market

In a market of cloud providers that are closer to the client, such as the emerging cloud communities [Peter Mell and Tim Grance, 2009], clients will be more mobile and independent of each provider. In this way, clients will more frequently have to look for the best prices and partial utility distributions. To this end, based on matrices P and M, equation 6.9 defines a new set of matrices for each VM type. In this set, the matrices represent, for each VM type, the multiplication of a price's vector (a line in the P matrix) by the matrix of partial utilities of the provider. C is the ordered set of user's classes.

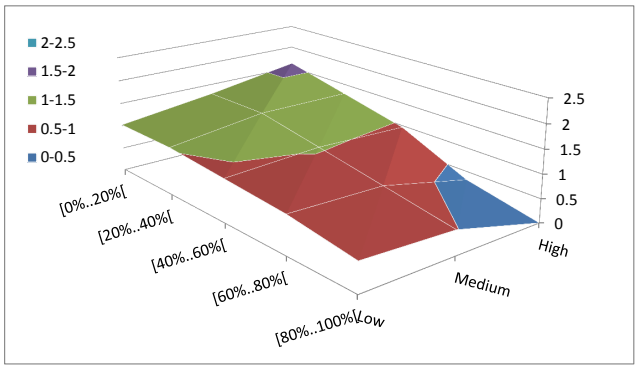
$$PM_{type} = \forall classes \in C : P_{type,classes} \cdot M \tag{6.9}$$

Figure 6.4 illustrates an instance of this new set for the previously described VM types. The differences are increasingly significant as we increase the capacity (and consequently the prices) of the VMs. While these matrices represent related pricing profiles, they can be used by costumers to compare and arbitrate over different providers, either for a given user class and VM size, or for global aggregate assessment. This further allows users to graphically navigate through the providers' pricing profiles. In particular, this make it possible to explore the pricing profile of a given provider, and determine

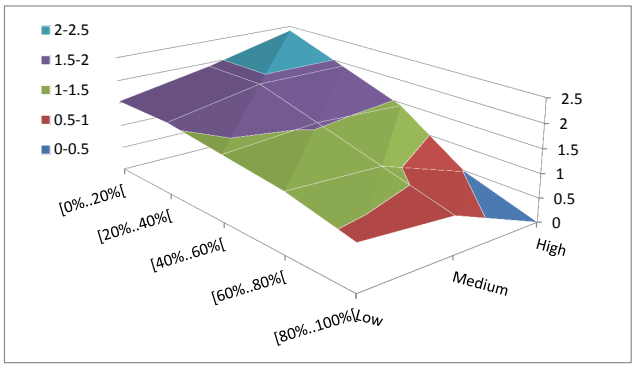


(a) PM_{Micro}

(b) PM_{Small}



(c) $PM_{Regular}$



(d) PM_{Extra}

Figure 6.4: Matrices combining price and utility for the different VM types and partial utilities.

the reallocation of resources a user is willingly to have, in order to fulfill a given level of cost constraints.

Summary

In this chapter, we have presented the motivation to drive VM allocation using a partial-utility strategy together with a cost model that supports this approach. We have shown how this approach compares with other similar SLA-aware systems. In the following chapter, we will present allocation algorithms that use this strategy to place VMs in available nodes of a datacenter.

7 Partial Utility Scheduling Algorithms and Implementation

Contents

Chapter overview

In this chapter, we discuss scheduling algorithms to be used by providers to organize VMs in their data centers, along with the implementation of these algorithms in a state of the art cloud simulator [Simão and Veiga, 2013b].

The chapter is organized in three main sections. Section 7.1 starts by presenting a comprehensive set of scheduling algorithms which aim to maximize the packing of VMs in a datacenter, taking into account the partial-utility cost model discussed in Chapter 6. The scheduling algorithms target the selection of a host inside a data center and the distribution of resources from that host to the set of VMs assigned to it.

In Section 7.2, we introduce the importance of simulators in distributed systems research and delve into details about two widely adopted simulators: SimGrid [Casanova et al., 2008] and CloudSim [Calheiros et al., 2011]. We then focus on the adopted simulator, CloudSim, showing the main building blocks and extension points. Section 7.3 concludes the chapter with some details about the scheduling algorithm in this simulator, in particular, the extensions made to the object model.

7.1 Partial Utility-based Scheduling for IaaS Deployments

In general, the problem we have described is equivalent to a bin packing problem [Garey et al., 1976]. So, the scheduling process must impose constraints, on what would be a heavy search problem, and be guided by heuristics for celerity. We consider as relevant resources of a host, and requirements for a virtual machine, the following: number of cores, the processing capability of each core (expressed as *millions of instructions per second* (MIPS), MFLOPS, or any other comparative reference), and memory (in MiB). The following algorithms focus on the first two requirements but a similar strategy could be used for memory. They allocate new requested VMs to these resources, taking into account the partial utility model described in the previous section.

Algorithm 1 presents what is hereafter identified as the base allocation algorithm. It takes a list of hosts and a virtual machine (with its resource requirements) that needs to be allocated to physical hardware or otherwise fail. It will search for the host with either more or less available cores, depending on the order criterion (Φ). When a greater-than ($>$) criterion is used, we call it First-Fit Increasing (FFI) since the host with more available cores will be selected. When a less-than ($<$) criterion is used, we call it First-Fit Decreasing (FFD), since the host with less cores still available will be selected. This base allocation will eventually fail if no host is found with the number of requested MIPS, regardless of the class of each VM. In this situation, a classic provider cannot fulfill further requests without using extra hardware, which may simply not be available.

Function ALLOCATE checks if a VM can be allocated in a given host (h). Current allocation strategies either i) try to find the host where there are still more physical cores than the sum of virtual ones, and each individually has enough capacity to hold the VM; ii) try to find the host with a core where the VM can fit, even if shared with others; iii) takes resources from VMs in the host to fit the new VM, until no more computational power is available in the host. In the first two cases, if the conditions are

Algorithm 1 Generic base allocation: First-Fit Increasing/Decreasing

Require: $hosts$ list of available hosts
Require: vm VM to be allocated
Require: Φ order criterion
1: **function** BASESCHEDULING($hosts$,vm)
2: $currCores \leftarrow 0$ *or* $+\infty$ *depending on criterion*
3: $selectedHost \leftarrow null$
4: **for all** $h \in hosts$ **do**
5: **if** AVAILABLECORES(h) Φ $currCores$ **then**
6: **if** ISCOMPATIBLE(h, vm) **then**
7: $currCores \leftarrow$ AVAILABLECORES(h)
8: $selectedHost \leftarrow h$
9: **end if**
10: **end if**
11: **end for**
12: **if** $selectedHost \neq null$ **then**
13: ALLOCATE($selectHost, vm$)
14: **return** $true$
15: **end if**
16: **return** $false$
17: **end function**

not met the allocation will fail. In this case, *unused cores* is used in the sense that they are still available to allocate without incurring in overcommitment. All the physical cores will be in use, as usual, but they will not be used to 100% capacity. Thus if, for example, 4 cores have an average of 25% CPU ocupation, we consider it equivalent to saying there are 3 unused cores (i.e., still available to allocate without overcommitment).

In the last case, the allocation will succeed, but not taking the best choices for the new utility model proposed in Section 6.3. Function ISCOMPATIBLE uses the same strategies but only determines whether the conditions hold, leaving the effective allocation to the ALLOCATE function.

7.1.1 Analysis of the scheduling cost of the utility-oblivious scheduling

Algorithm 1 iterates over M hosts looking for the one with minimum or maximum available cores. In either case, this algorithm determines a compatible host in O(M) iterations. The ISCOMPATIBLE function depends on the total number of cores, C, to determine, i) if there is any unused core and; ii) if any core still has available MIPS. After determining the host where to allocate the requested VM, function ALLOCATE can also complete with the same asymptotic cost. Thus, in summary, Algorithm 1 has a cost of

Algorithm 2 Partial utility allocation strategies

Require: $hosts$ hosts ordered by available resources
Require: vm new VM to be allocated
Require: $maxADI$ maximum aggregated degradation index
1: **function** VMUTILITYALLOCATION($hosts$,vm)
2: **if** BASESCHEDULING($hosts, vm$) $= true$ **then**
3: **return** $true$ ▷ No need to overcommit VM(s)
4: **end if**
5: $selection \leftarrow null$
6: $hosts \leftarrow$ *sort* $hosts$ *in ascending order of available resources*
7: **for all** $h \in hosts$ **do**
8: $needed \leftarrow$ REQUESTED(vm) $-$ AVAILABLE(h)
9: $vmList \leftarrow$ ALLOCATEDVMS(h)
10: $selection \leftarrow$ SELECTVMS($vmList, needed$)
11: **if** ADINDEX($hosts, selection$) $< maxADI$ **then**
12: **for all** $(vm, df) \in selection$ **do**
13: CHANGEALLOCATION(vm, df)
14: **end for**
15: **return** $true$
16: **end if**
17: **end for**
18: **return** $false$
19: **end function**

$O(M \cdot C)$.

7.1.2 Partial utility-aware scheduling strategies

When there are no hosts that can be used to allocate the requested VM, some redistribution strategy must be used, while maximizing the *renting cost* as defined in Section 6.3. This means that the provider can use different strategies to do so, by giving priority to larger or smaller VMs (regarding their *virtual power*) or to classes with higher or lower base price.

We have extended the base algorithm so that, when a VM fails to be allocated, we then have to find a combination of degradation factors that makes it possible to fit the new VM. Four strategies/heuristics were implemented to guide our partial utility-driven algorithm. They differ in the way a host and victim VM are selected for degradation. They all start by taking the host with the most available resources, that is, with more unitary available cores and with more total computation power (MIPS).

Algorithm 2 presents the modifications to the base algorithm to enable partial utility allocation strategies. After a host is selected, a set of VMs must be chosen from the list of

allocated VMs in that host, i.e., operation SELECTVMs presented in Algorithm 3. These VMs are selected either by choosing the ones from the smallest size type (which we call min strategy) or the ones with the biggest size (which we call max strategy). This is controlled by using VM_{types} sorted in ascending or descending order. In both cases, there are variants that combine with the lowest partial utility class (w.r.t. the definition of Section 6.3), either in ascending or descending order, regarding its partial utility class, i.e., min-class and max-class.

Algorithm 3 Partial utility allocation by min/max VM type and minimum class price

Require: VM_{types} ascending/descending list of VM's types
Require: $vmList$ list of VMs allocated in host
Require: $target$ virtual power needed to fit all VMs

```
1:  function SELECTVMS(vmList, target)
2:      selection ← null
3:      sum ← 0
4:      vmList ← sort vmList in ascending order of price's class
5:      while sum < target do
6:          for all t ∈ VM_types do
7:              for all vm ∈ vmList : VMTYPE(vm) = t do
8:                  r_vm ← NEXTRANGE(vm)
9:                  selection ← selection ∪ (vm, r_vm)
10:                 sum ← sum + VIRTUALPOWER(vm) * (1 − r_vm)
11:                 if sum ≥ target then
12:                     break
13:                 end if
14:             end for
15:         end for
16:     end while
17:     return selection
18: end function
```

7.1.3 Analysis of the partial-utility scheduling cost

Algorithm 2 goes through the list of hosts trying to find a combination of VMs whose resources can be reallocated. For each host, VMs are selected based on Algorithm 3. The cost of this procedure depends on a sort operation of N VMs, $O(N \lg(N))$, and a search in the space of minimum degradations to reach a target amount of resources. This search depends on r intervals in matrix M (Equation 6.7) and t classes for prices (currently, three, as presented in Section 6.3.4), with a cost of $O(rtN)$. This results in an asymptotic cost of $O(rtN + N \lg(N)) = O(N \lg(N))$. Overall, the host and VMs selection algorithm cost belongs to $O(M \lg M + MN \lg N)$. Because there will be more VMs (N), across the datacenter, than hosts (M), the asymptotic cost is $O(MN \lg(N))$.

In the next section, we briefly present the more relevant details of extending the CloudSim [Beloglazov and Buyya, 2012] simulator to evaluate these strategies.

7.2 Cloud simulators and the CloudSim framework

Simulators allow researchers to focus on a specific problem (e.g. algorithm, topology) without having to immediately deal with technical problems of the deployment. Furthermore, these tools comply with two important research goals - repeatability of experiences and fair comparison with other strategies [Veiga et al., 2011]. Similarly to the areas related to natural sciences, setting up an *in vivo* experience in distributed systems can also be time-consuming and costly.

Over the years, several simulators have been developed, covering the major topics in the area. Examples for supercomputing and HPC in general include BigSim [Zheng et al., 2010] and MPI-SIM [Prakash and Bagrodia, 1998]. For general network research, the NS2 [Issariyakul and Hossain, 2008] is a well-established simulator whose network format specification was adopted by other simulators with networking requirements. For peer-to-peer networks, PeerSim [Montresor and Jelasity, 2009] and OverSim [Baumgart et al., 2007] are widely used simulators. For volunteer computing there is the EmBOINC [Estrada et al., 2009] which can read configuration settings from real BOINC servers [Anderson, 2004], and outputs performance metrics such as throughput, latency, and starvation.

Grid and Cloud simulators are also well represented. A classic grid simulator is GridSim [Buyya and Murshed, 2002] from where the CloudSim [Calheiros et al., 2011] was adapted and significantly extended to support the typical cloud artifacts. The iCanCloud [Núñez et al., 2012] is a new simulator with a customizable job broker and a rich user-interface to configure the simulation. SimGrid [Casanova et al., 2008] can simulate low-level details of virtual machines migration [Hirofuchi et al., 2013]. GreenCloud [Kliazovich et al., 2010] is an extension to NS2 [Issariyakul and Hossain, 2008] and has support to accurately model the energy consumed by the equipment that is typically present in a datacenter (e.g. computing servers, network switches).

These simulators are, in fact, frameworks that not only have to be parametrized with a particular configuration but can also be extended to support new algorithms regarding the research topic [Sakellari and Loukas, 2013]. The language to program extensions

can be the same used to develop the simulator, usually C++ or Java, or some domain-specific language, as in the case of GreenCloud [Kliazovich et al., 2010].

All these simulators have a significant impact in their research communities. The most prominent examples in our work context are SimGrid and CloudSim, with more than 310 and 665 citations in July 2014, respectively, according to Google scholar.[1] The GreenCloud and iCanCould simulators lag behind, with 142 and 32, respectively.

7.2.1 SimGrid

SimGrid is a particular case because it represent a set of simulators. It has been used in more than one topic in the distributed systems arena. Research with this simulator includes works with peer-to-peer systems [Quinson et al., 2012], networked applications based on the pattern of Message Passing Interface (MPI) [Bedaride et al., 2013] and packet-based network protocols in general [Velho et al., 2013]. Although SimGrid can simulate many forms of distributed systems, only recently was added support for the cloud computing artifacts that enable the simulation of a IaaS service model. These artifacts include the representation and simulation of virtual machines [Hirofuchi and Lebre, 2013] and their migration inside the datacenter [Hirofuchi et al., 2013].

Currently, SimGrid has basic support for IaaS deployments. Important efforts where made to accurately simulate the migration steps used in system-level virtual machines [Hirofuchi et al., 2013], namely the precopy algorithm [Clark et al., 2005] used the Linux kernel-based virtual machine Qemu/KVM.[2]. The simulator captures the essential nature of this algorithm using a parameter, the *memory update speed* measured in MiB/s, which is used to represent different kinds of applications (i.e. with different memory usage patterns) running in each VM. The memory update speed, together with the network bandwidth will determine the total migration time.

However, the SimGrid simulators are progressively integrating these capabilities. There are ongoing efforts to develop a load injector to simulate different migration options inside the data center [Hirofuchi et al., 2013] and simulators of public and private Cloud APIs, such as the OpenStack and Amazon EC2 and S3 [Desprez and Rouzaud-Cornabas, 2013], based on the core SimGrid simulation engine.[3]

[1] `http://scholar.google.pt`, visited July 8, 2014
[2] `http://www.linux-kvm.org/page/Main_Page`, visited July 8, 2014
[3] `http://www.openstack.org/`, visited July 9, 2014

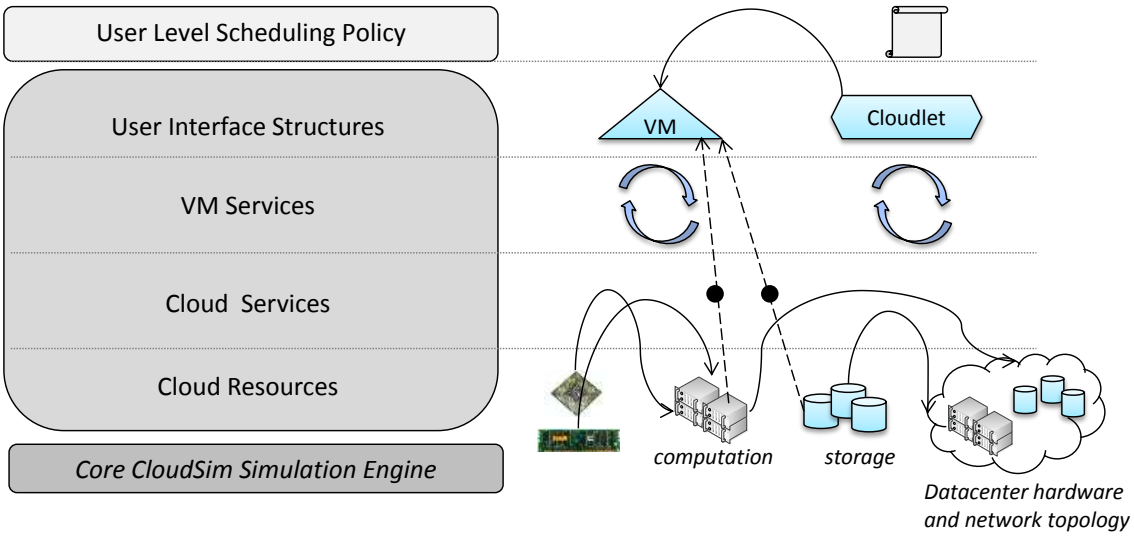


Figure 7.1: Organization of CloudSim simulation environment

7.2.2 CloudSim

Compared with SimGrid, CloudSim has more extension points where different scheduling and migration policies can be researched. We have chosen CloudSim because of its maturity in IaaS simulations, but also because it allows us to focus on the definition and implementation of scheduling policies for the allocation of virtual machines in the data center, in a structured but simple way.

Figure 7.1 shows the modular organization of CloudSim. The first layer, User Level Scheduling Policy, represents the configuration that the user of CloudSim has to perform in order to setup the simulation, which includes an optional external configuration regarding network connections, using the BRITE [Medina et al., 2001] format. The most significant work performed at this layer consists on the specification on how tasks, which are known as *cloudlets*, are assigned to virtual machines. The assignment between *cloudlets* and VMs is done by programming a specific kind of *broker*, in the form of a Java class.

The basic representation of cloudlets is determined by the number of cores, memory, and storage they request, along with the number of millions of instructions (MI) they represent. VMs have a certain number of virtual cores, each with a certain amount of computation capacity, measured in MIPS. Recent additions have made it possible to represent more accuratly common types of applications running in the cloud, such as

web-based multi-tier application, workflows and MPI-based application [Calheiros and Buyya, 2014; Garg and Buyya, 2011].

The CloudSim core is composed by the following four layers. At the top, User Interface Structures, represents the artifacts that the user interacts with, namely virtual machines and cloudlets. Following are the VM Services. This layer determines how VMs use the resources available at their assigned host and how cloudlets make progress based on the resources available at their assigned VMs. The Cloud Services layer's main goal is to provision VMs in hosts and determine how the simulated physical resources (CPU, memory, and storage) are interconnected. The final core layer, Cloud Resources, represents data centers, which can be distributed across different geographic locations.

The bottom layer is the event engine which keeps track of the simulated time. This time is independent of the real system running the simulation. The event engine is used for communication between *simulation* entities, such as broker, datacenter, and network switch. These entities are *active* entities in the sense that they are the ones that receive and generate new events, delegating further action to *passive* entities such as VMs.

Out-of-the box scheduling and Utilization Models

Besides the assignment of tasks to VMs, which is, conceptually, a responsibility of the researcher, as explained in the previous section, scheduling decisions are done in three major points, when: i) assigning VMs to hosts, ii) determining the number of cores assigned to each VM, iii) determining the number of virtual cores, and therefore, the millions of instructions per second available for a cloudlet to make progress, iv) determining a new assignment of VM-to-host using migration. For all these points, there are default policies that can be customized by the researcher. In the first point, different packing strategies can be researched, including the ones that are energy-aware [Khosravi et al., 2013; Beloglazov and Buyya, 2012]. The second layer of scheduling is a representation of the work done by each host's hypervisor in real data centers. The third level of scheduling is akin to the responsibility of an operating system running inside a VM because several tasks can be executing simultaneously.

The last point is where migration policies can be plugged-in. These policies should take into account the current load of the datacenter and, based on a goal to reach (e.g. consolidation to minimize energy consumption), decide a new configuration of assignment of hosts to VMs. By default, CloudSim does not reconfigure VMs during cloudlet

execution.

The configuration of all these policies and the execution of cloudlets is what determines the result of a simulation. Therefore, cloudlets make more progress if they execute in a VM whose virtual cores have more MIPS. The finish time of cloudlets can be simply based on their total number of instructions (#instr) and available capacity computing capacity (CP), which in that case, the finish time (ft) is determining by adding the ratio of these two values to the start time (st), as presented in Equation 7.1. Because #instr is in million of instructions and CP is in millions of instructions per second, the result is a value in seconds.

$$ft = st + \frac{\#instr}{CP} \text{ (seconds)} \tag{7.1}$$

Instead of this purely analytic progress model, cloudlets can make progress based on a percentage of CPU used during a time span. This is a very useful feature because real world applications exhibit different phases during their execution, where they vary their resource consumption. In this case, the simulation has to be configured with the time interval at which the samples where collected. The finish time of these cloudlets now depends on the the percentage of CPU used during a time interval, as presented in Equation 7.2.

$$ft = st + \frac{\sum_{t=0}^{intervals} cpuusage(t) * CP}{CP} \text{ (seconds)} \tag{7.2}$$

Currently, CPU usage can be either *full*, *random*, or based on traces collected from real systems.[4] These utilization models are also available to model memory and bandwidth usage. However, currently, the effective progress of cloudlets does not depend on these two last mentioned models. They are only used for allocation purposes, to determine the current level of utilization of the corresponding resource.

[4]Version 3.0.3 of CloudSim comes with hundreds of CPU traces from PlanetLab's *slices* [Chun et al., 2003], which are virtual machines where experiments are executed.

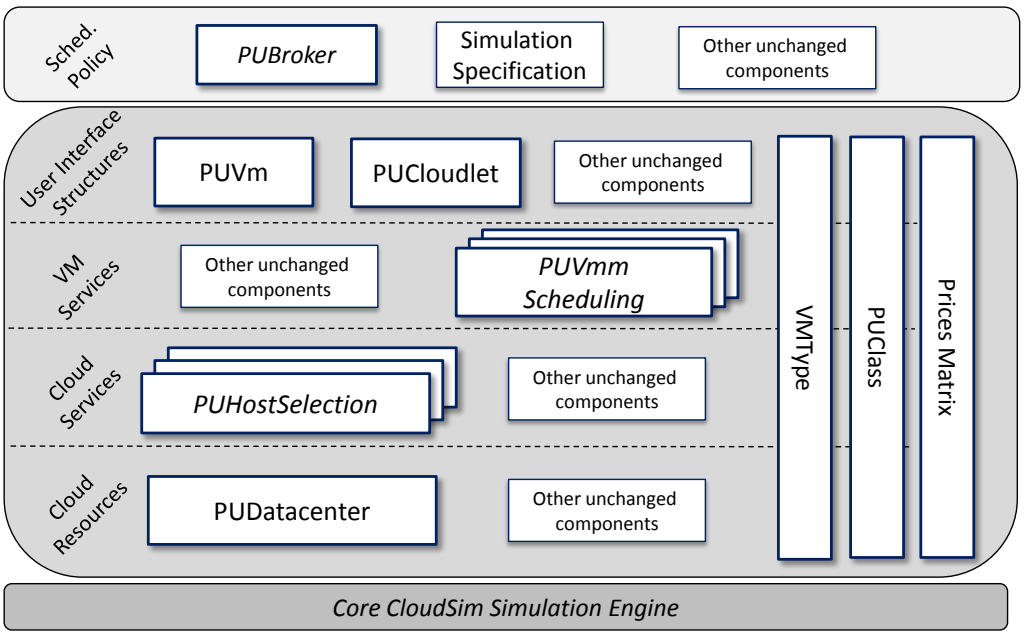


Figure 7.2: Highlighted extensions to the CloudSim simulation environment

7.3 Implementing the Partial Utility-Driven Scheduling in CloudSim

We have implemented and evaluated our partial utility model on a state-of-the-art simulator, CloudSim [Calheiros et al., 2011]. CloudSim is a simulation framework that must be programmatically configured, or extended, to reflect the characteristics and scheduling strategies of a cloud provider.

Figure 7.2 highlights the new classes added to the simulation environment, which range from exploring extension points, like the virtual machine allocation to hosts, to enrichments of the object model to include partial utility-related types (e.g. VM type, specification tables).

Regarding CloudSim's base object model, we have the `PUVm` type which incorporates information regarding its partial utility class. The scheduling algorithms were implemented as extensions of two main types: `VmAllocationPolicy` and `VmScheduler`. The former determines how a VM is assigned to a host while the latter determines how the virtual machine monitor (VMM) of each host allocates the available resources to each VM. It can use and re-use different matrices of partial utility classes and VM base prices, defined in the type that represents the partial utility-driven datacenter.

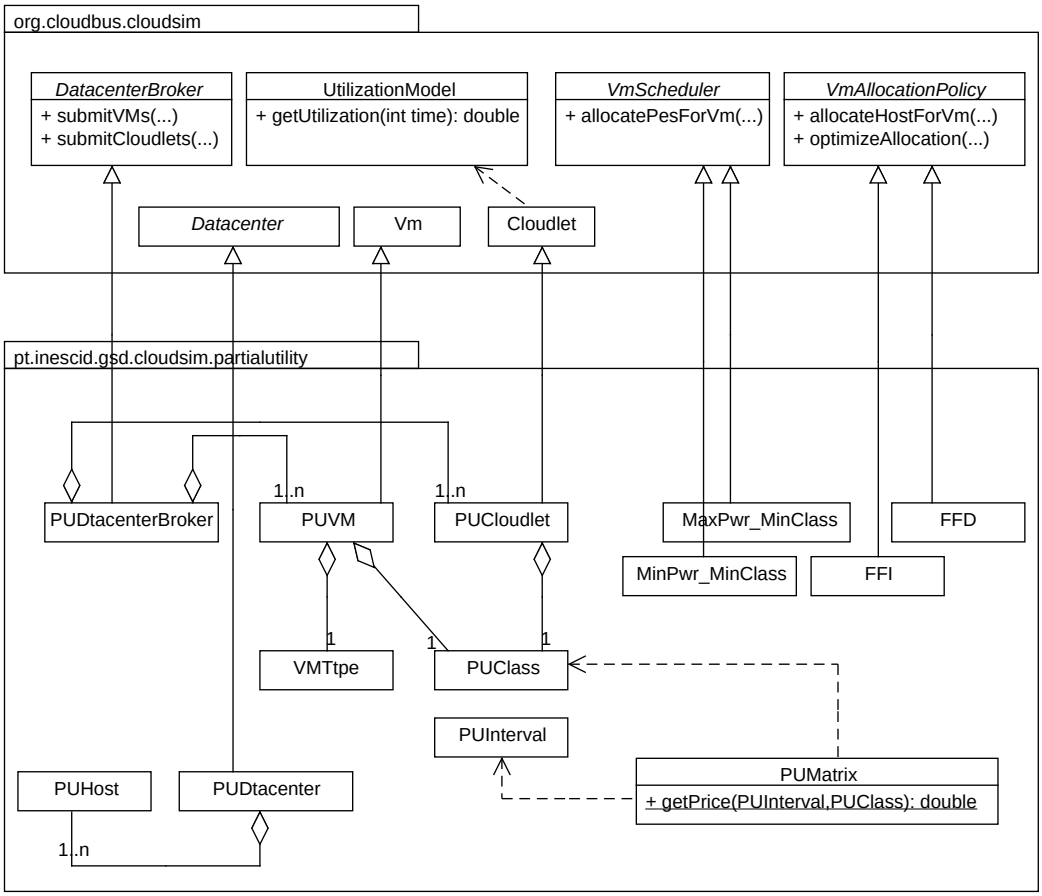


Figure 7.3: Class diagram with extensions to the CloudSim object model

The CloudSim type that represents the dynamic use of the available (virtual) CPU is the `Cloudlet` type. Because cloudlets represent work being done, each cloudlet must run in a VM with the appropriate type, simulating work being done on several VMs with different computational power. Thus, regarding the `Cloudlet` class, we added information about which VM type must be used to run the task. To ensure that each cloudlet is executed in the correct VM (depreciated or not), we also created a new broker (extended from `DatacenterBroker`).[5].

Figure 7.3 shows the relations between the CloudSim base classes and the ones developed to support our allocation strategies.

[5]Code is available at `https://code.google.com/p/partial-utility-cloudsim/`, visited May 14, 2014

Summary

This chapter was dedicated to the algorithms that allocate VMs according to the partial-utility cost model. It started by presenting the algorithms and their cost analysis. The chapter then focused on implementation details, which are essential to make a realistic evaluation. A brief introduction and comparison of the most known and used cloud simulators was made. Finally, the necessary extensions to the code base of CloudSim [Calheiros et al., 2011] were presented. This implementation will be used to evaluate and compare the proposed approach versus strategies that either are fully successful allocating all requested resources or simply fail.

8 Evaluation

Contents

Chapter overview

This chapter reports a detailed evaluation of our partial utility-driven algorithms, while comparing them with some of classic algorithms for placing virtual machines (or servers) in a datacenter.

Section 8.1 describes the configurations of the different datacenters used in our simulation. Section 8.1.1 analyzes what happens when virtual machine allocation is made in an all-or-nothing way. Section 8.2 discusses a first approach to overcommitment. Section 8.3 shows the effects of the different utility-driven overcommitment approaches presented in Chapter 7. The evaluated metrics are relevant to the provider (VM requested but not allocated, resource utilization, revenue) and to the owner of the VMs (total execution time of workloads).

DC size	Hosts	Cores	HT	MHz	Mem (Gbytes)
Size-1	10	2	no	1860	4
	10	2	no	2660	4
Size-2	20	4	yes	1860	8
	20	4	yes	2660	8
Size-3	40	4	yes	1860	8
	40	4	yes	2660	8

Table 8.1: Hosts configured in the simulation. Number of hosts per configuration, number of cores per host, computational capacity, hyper-threading, Memory capacity

8.1 Methodology and Configurations

In this section we evaluate the proposed scheduling based on partial utility. To do so, we first describe the datacenters used in the simulation and the VM types whose base price was already presented in Section 6.3.4. The datacenters are characterized by the number and type of hosts as described in Table 8.1. We used three types of datacenters hereafter known as *Size-1*, *Size-2* and *Size-3*. Each datacenter configuration is representative of a specific scenario we want to evaluate. Configuration *Size-1* represents a typical configuration of a cloud community datacenter [Khan et al., 2014], where low-end processors are used. Configuration *Size-2* represents a set of clusters owned by our research labs, where raw computational capacity is around 300 cores. The simulation uses quad-core processes with hyper-threading and a computational capacity per core in the range used by Xeon processors with this number of cores. Configuration *Size-3* doubles the number of hosts, keeping their computational configuration.

Available VM types are presented in Table 8.2. To enrich the simulation scenario, VMs have different *sizes*, simulating the request of heterogeneous virtual hardware. This is a common practice in the literature [Malik et al., 2013; Beloglazov and Buyya, 2012; Calheiros et al., 2011; Mian et al., 2012]. The configurations chosen for each VM type will put our strategies to the test when a new VM request cannot be fulfilled. The number of cores depends on the size of the datacenter. We simulate different scenarios where the number of cores per VM will increase as more physical resources are available. Configuration *Size-1* uses VMs with 1 core. Configuration *Size-2* and *Size-3* were simulated with VMs having 2 and 4 cores respectively. Each virtual core, of each VM type, will have the CPU power presented in Table 8.2.

	micro	small	regular	extra
Virtual CPU Power ($\times 10^3$ MIPS)	0.5	1	2	2.5
Memory (Gbytes)	1	1.7	2.4	3.5

Table 8.2: Characteristics of each VM type used in the simulation

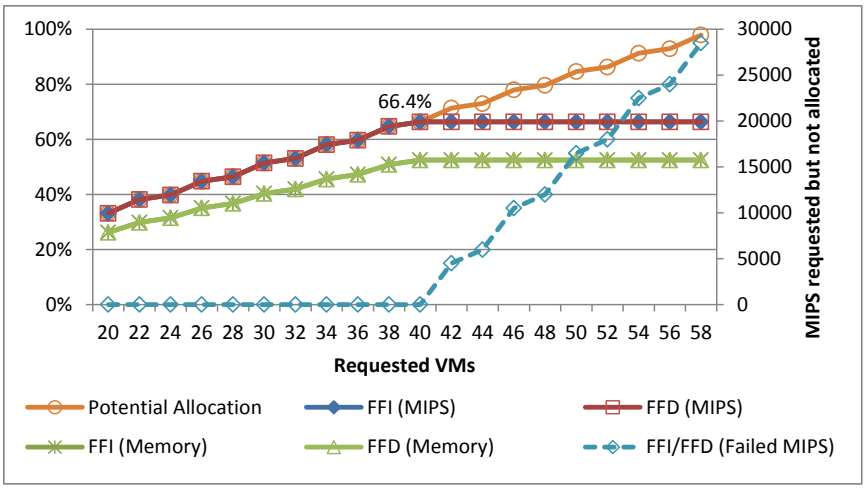


Figure 8.1: Base algorithm which allocates a single VM to each CPU core.

We tried to allocate an increasing number of VMs. Each requested VM has a type (e.g. *micro*). We considered VM *types* to be uniformly distributed (realistic assumption) and requested one type at a time. The following sections highlight the differences between the current allocation strategies and the ones that can cope with the proposed flexible SLAs.

8.1.1 Utility Unaware Allocation

Figures 8.1 and 8.2 show the effects of using two different allocation strategies for host selection, and other two regarding the use of cores, but still without taking into account each client's partial utility. Each x-axis value represents a total number of VMs requested, r, and the value in the corresponding left y-axis is the datacenter occupation (MIPS and Memory) obtained when $r - f$ number of VMs are able to run, with $f \geq 0$ being the number of not allocated VMs. The host selection is based on the First-Fit Increasing (FFI) and First-Fit Decreasing (FFD) algorithms, described in Section 7.1. In each of these approaches, we present the total percentage of MIPS and memory allocated, in the left y-axis, for each set of requested VMs. Regarding the 6th series, "FFI/FFD (Failed

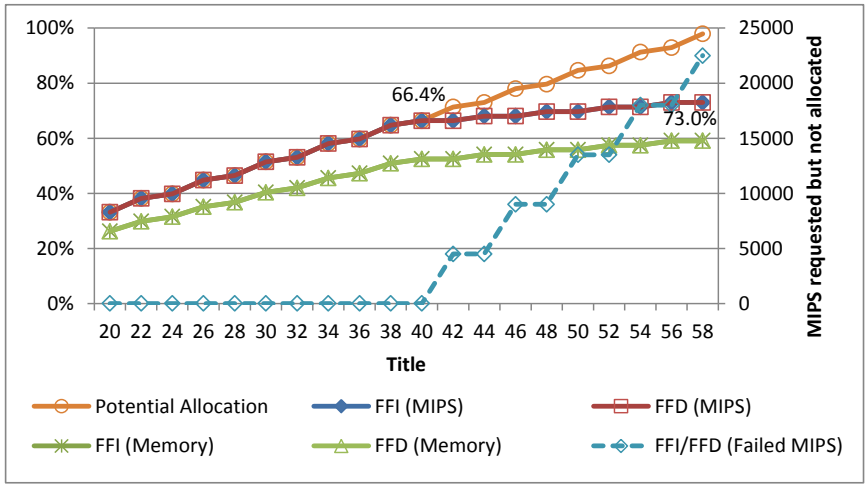


Figure 8.2: Base algorithm which allocates one or more VMs to a single CPU core.

MIPS)", its results are plotted in the right y-axis.

In Figure 8.1, each VMM (one for each host) allocates one or more cores to each VM and does not allow any sharing of cores by different VMs. In Figure 8.2, each VMM (one for each host) allocates one or more cores to each VM and, if necessary, allocates a share of the same core to a different VM.

In both cases, the datacenter starts rejecting the allocation of new VMs when it is about at 66% of its raw processing capacity (i.e., MIPS) and at aproximatly 53% of its memory capacity. Although there are still resources available (cores and memory), they are not able to fit 100% the QoS of the requested VM. As expected, the core sharing algorithm promotes better resource utilization because the maximum effective allocation is 73% of the datacenter, regarding raw processing capacity, and 59%, regarding memory capacity. The effective allocation of core-based sharing still continues to increase, at a slower rate, because there are smaller VMs that can be allocated.

Figure 8.3 shows the counting of VM failures grouped by the VM type and VMM scheduling strategy. The simulation uses hosts with different capacities and heterogeneous VMs, for realism, as workloads are varied and resources not fully symmetric, as it happens in many real deployments in practice. The allocation strategy that enables sharing of resources is naturally the one with fewer failed requests. In the configured *Size-1* datacenter, the no-core sharing strategy starts rejecting VMs when a total of 40 is requested. In both cases, the bigger VMs (i.e., the ones requesting more computing power) are the ones with a higher rejection rate.

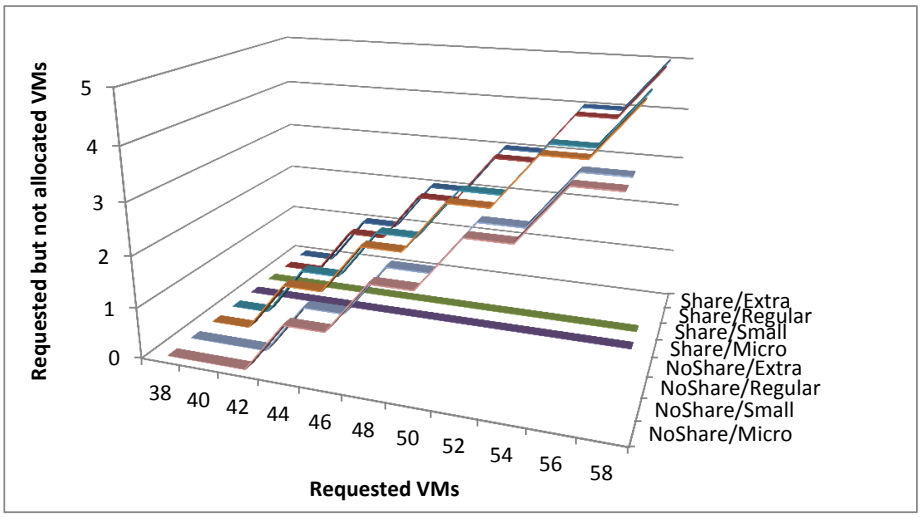


Figure 8.3: Types, sizes, and counting of requested but not allocated VMs

	Base No Core Sharing						Base Core Sharing					
# VMs	Failed	E	R	S	M	Hosts	Failed	E	R	S	M	Hosts
38	0 (0%)	0	0	0	0	+0	0 (0%)	0	0	0	0	+0
42	2 (5%)	1	1	0	0	+1	2 (5%)	1	1	0	0	+1
60	20 (33%)	5	5	5	5	+10	10 (17%)	5	5	0	0	+5
76	36 (47%)	9	9	9	9	+18	18 (24%)	9	9	0	0	+8

Table 8.3: Summary of VMs requested but not allocated and the number of additional hosts when cores are not shared

Table 8.3 (with results for an added number of VM requests) also shows, in the "Hosts" column, the number of extra hosts that would be necessary to fit the rejected VMs. These extra hosts are determined by summing all the resources not allocated and dividing by the resources of the type of host with more capacity (i.e., assuming a perfect fit and ignoring the computational cost of determining such a fit). Our solution avoids these extra hosts by readjusting the allocation of new and accepted VMs, following the utility and price matrices negotiated with the client.

Figure 8.4 shows the evolution of host utilization. This figure presents the result of allocating a total of 76 VMs. It shows that when using FFD with a core sharing approach, the number of unused hosts drops more slowly, while with the FFI approach all hosts start being used with less VMs allocated. If the datacenter is running a number and type of VMs below its rejection point, the FFD scheduling is better because hosts can be turned off or put in an idle state.

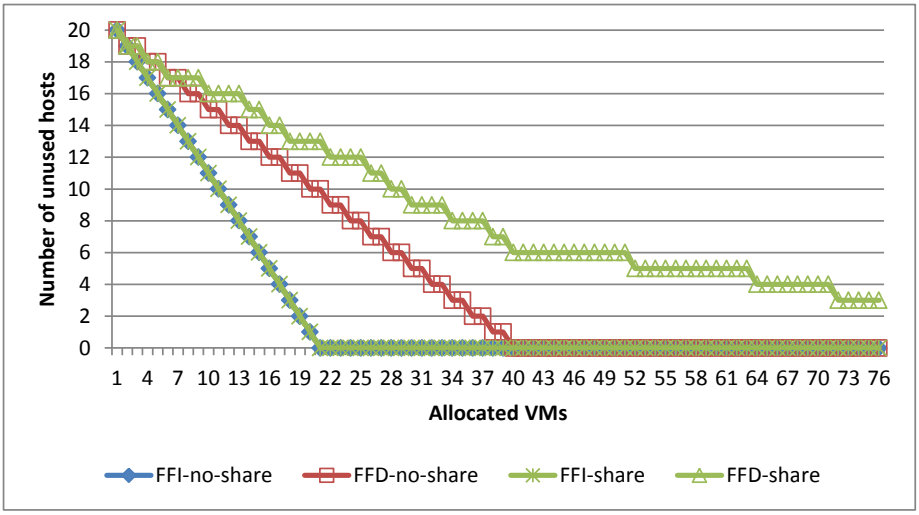


Figure 8.4: Unused hosts

8.2 Over subscription

Looking again to Figures 8.1-8.2, at the time when 58 VMs are requested, both strategies leave a significant part of the datacenter unused.

Figure 8.5 shows the results for the over-subscription algorithm (hereafter known as Base+OverSub), described in Section 7.1, that is oblivious to client classes, because it depreciates all VMs until no more computational power is available in the host. Given that this strategy looks at the host with more cores, ignoring the total computational power, it departs immediately from the potential allocation, because VMs are depreciated even when there is computational power available in other hosts. However, when more than 40 VMs are requested, it will grow more than the previous two allocation strategies.

Differently from the previous two strategies, it will not fail allocations, as can be seen in the right y-axis regarding the series "Failed MIPS (sec. axis)". Nevertheless, the effective allocation still has margin to grow. More importantly, using this approach, there is no way to enforce the SLA negotiated with the clients. This has a significant impact in the provider's revenue, as we will demonstrate next, when we present the results for our strategies that take into account the type of VMs, their classes, and the partial utility negotiated.

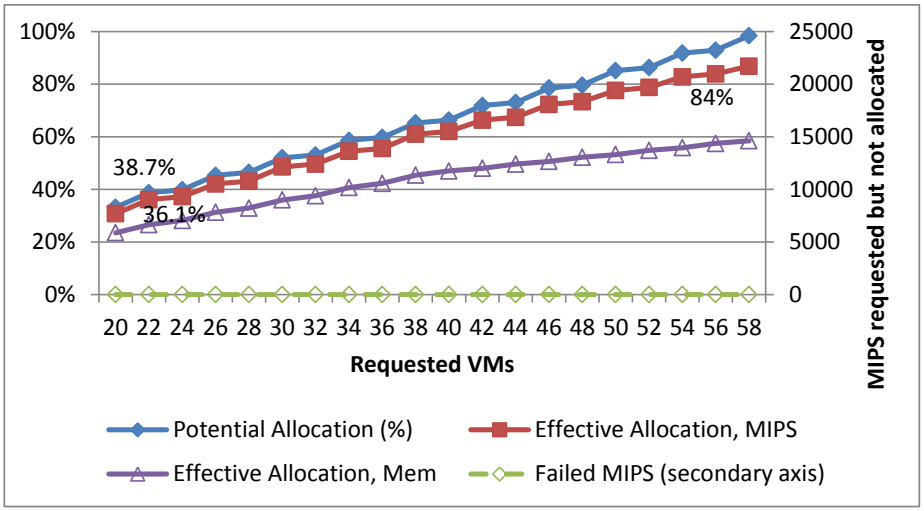


Figure 8.5: Base algorithm with over subscription, taking the first host with more cores, equal depreciation and unaware of client classes

8.3 Utility-driven Allocation

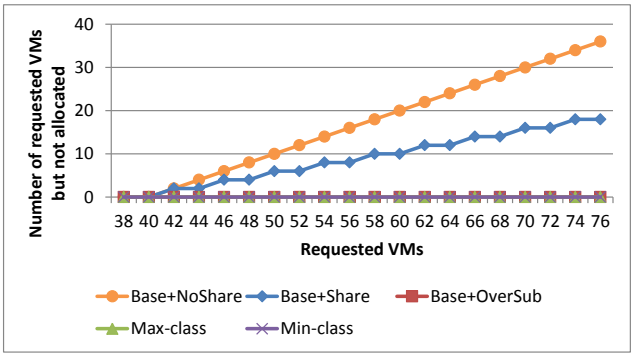
In utility-driven allocation, all requested VMs will eventually be allocated until the datacenter is overcommitted by a factor that can be defined by each provider. Along with the type, VMs are characterized by a partial utility class (e.g. *high*), as described in Chapter 6. In the following results, in each set of allocated VMs there are 20% of class *high*, 50% of class *medium*, and 30% of class *low*.

In this section, we will show how the proposed approach behaves, regarding two important set of metrics: i) allocation of VMs and, ii) execution of workloads by the allocated VMs. The first set of metrics is mainly important for the provider, while the second set of metrics is primarily of interest to the client. We compare utility-unware allocations with two heuristics presented in Section 7.1 - max-class and min-class.

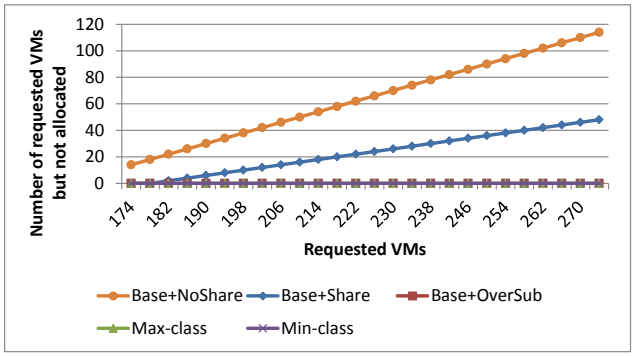
8.3.1 Allocation of VMs

Regarding provider-side metrics, we measure the number of failed VM requests, the resource utilization percentage, and the revenue (per hour). In all of the metrics, our strategies are at least as good as the Base+OverSub strategy, while specifically regarding revenue, we have average increases around 40%.

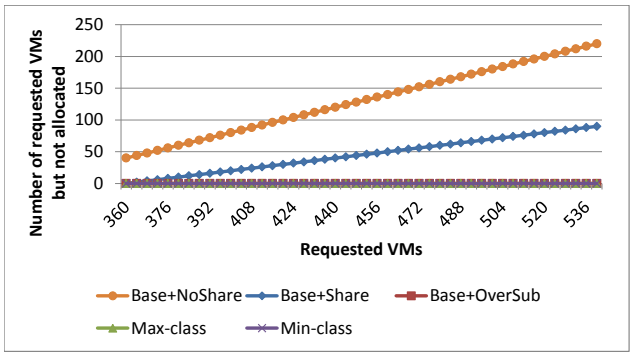
First, we compare our approaches with the base algorithm described in Section 7,



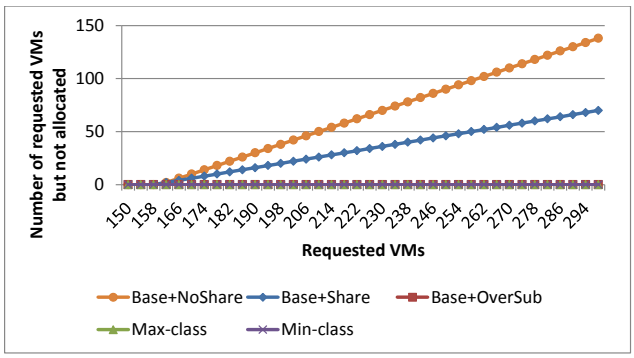
(a) Size-1 datacenter. 1 core/VM



(b) Size-2 datacenter. 2 core/VM

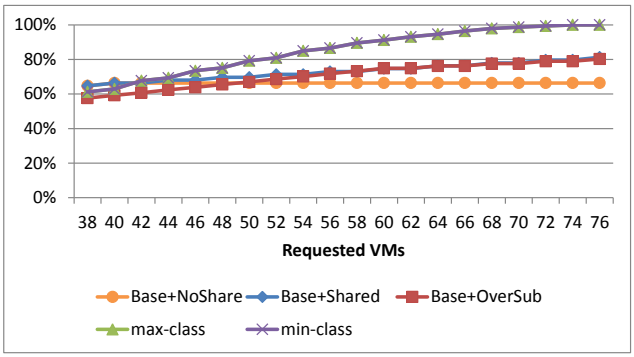


(c) Size-3 datacenter. 2 core/VM

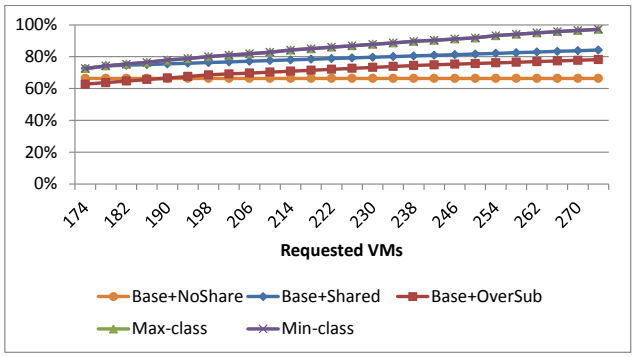


(d) Size-3 datacenter. 4 core/VM

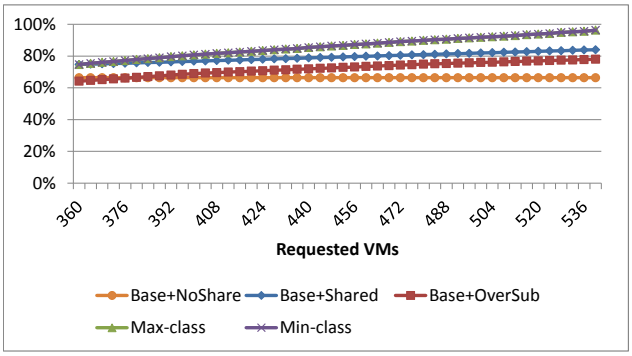
Figure 8.6: Number of requested but not allocated VMs using datacenters with different sizes and VMs with different number of cores.



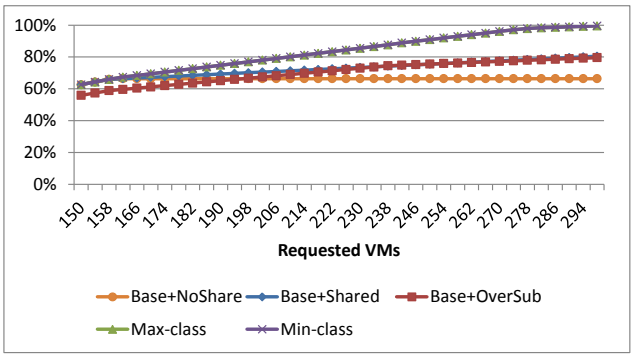
(a) Size-1 datacenter.



(b) Size-2 datacenter. 2 core/VM



(c) Size-3 datacenter. 2 core/VM



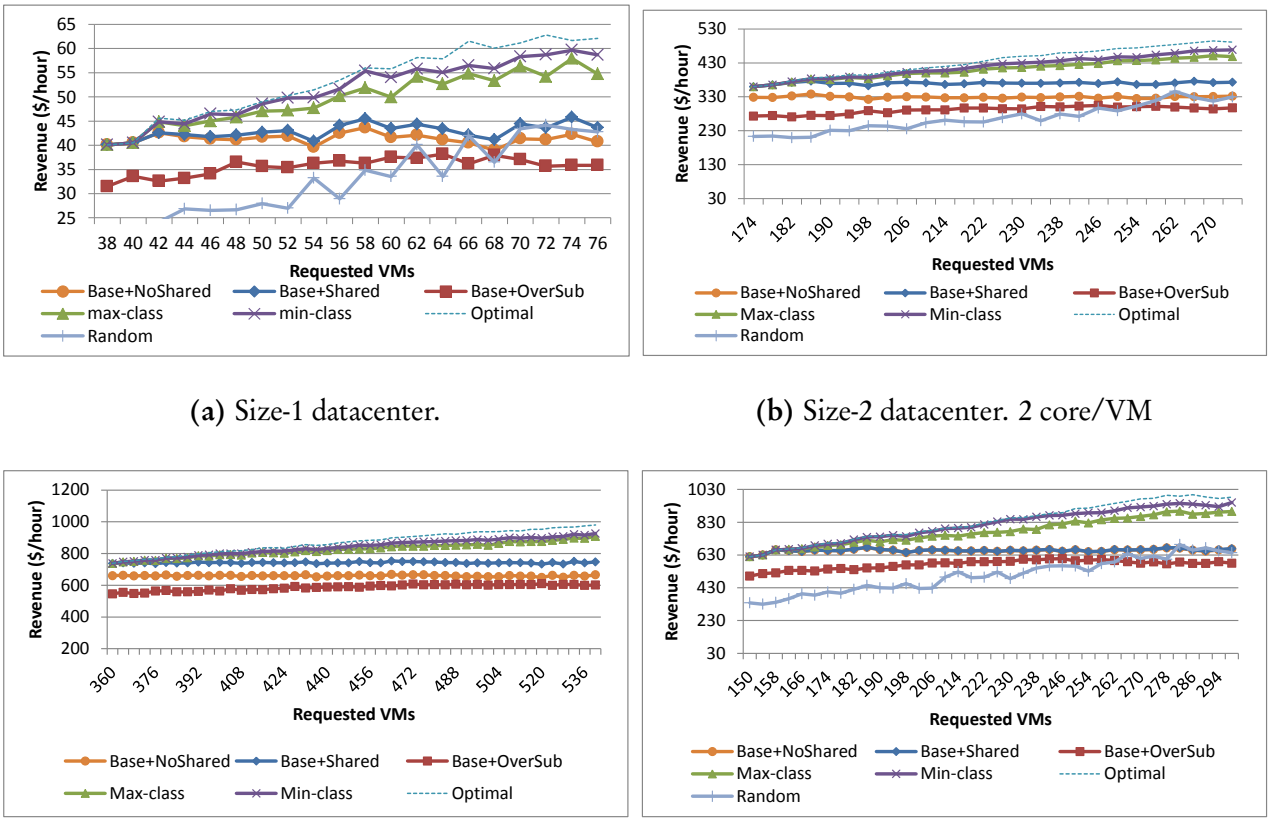
(d) Size-3 datacenter. 4 core/VM

Figure 8.7: Compared resource utilization using datacenters with different sizes

regarding the number of VMs that were requested but not allocated. Figure 8.6 shows that, while the base algorithm fails to allocate some VMs when 40 or more VMs are requested, the Base+OverSub and utility-driven strategies can allocate all requests in this configuration of the datacenter (note the collapsed series). Figure 8.6 presents similar results for a *Size-2* datacenter. In this case, after 180 VMs, the Base allocation algorithm rejects VMs of type *extra* and *regular*.

Second, we evaluate how available resources are utilized. Regarding this metric, Figure 8.7 shows the percentage of *resource utilization* with an increasing number of VMs being requested for allocation. Three observations are worth noting:

i) although with base allocation strategy some VMs are not scheduled, as demonstrated in Figure 8.6, others can still be allocated and can use some of the remaining resources;



(a) Size-1 datacenter.

(b) Size-2 datacenter. 2 core/VM

(c) Size-3 datacenter. 2 core/VM

(d) Size-3 datacenter. 4 core/VM

Figure 8.8: Compared revenue using datacenters with different sizes

ii) second, it is clear that our strategies achieve better resource utilization, while allocating all VMs;

iii) as the size of the datacenter increases, the strategy Base+OverSub lags behind to use all available resources. Our strategies can reach the peak in a similar fashion, across all sizes of datacenters.

The third and last metric evaluated for the provider is the revenue. Figures 8.8.a)-8.8.d) show how the revenue progresses with an increasing number of total VM requests. It clearly demonstrates the benefits of using a degradation and partial utility-driven approach, showing that the provider's revenue can indeed increase if the rejected VMs (above 40 in the *Size-1* datacenter and above 180 in the *Size-2* datacenter) are allocated, even if only with a fraction of their requested resources (i.e., subject to degradation

driven by partial-utility ranges).

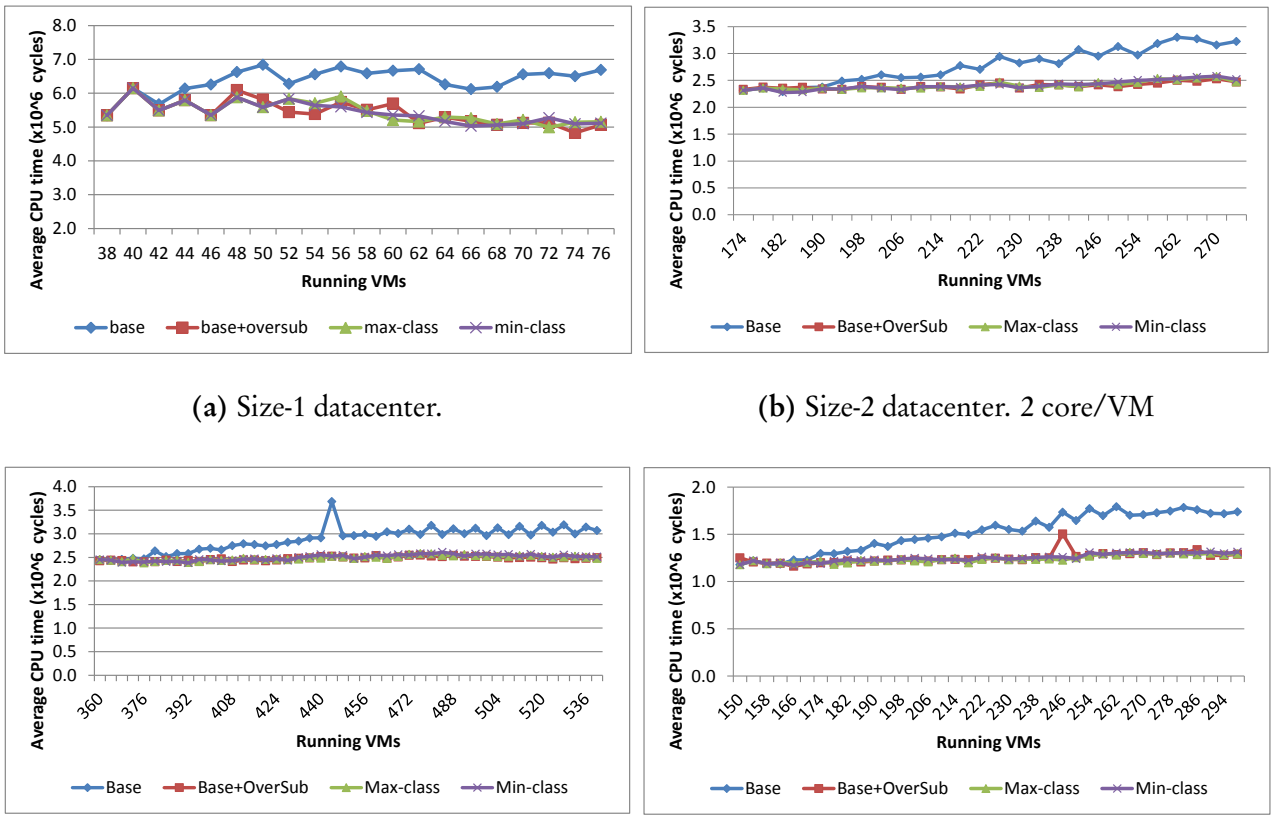
Comparing with the utility-oblivious redistribution, which also allocates all requested VMs (i.e., Base+OverSub), the increase of revenues in a *Size-1* type datacenter can go up to a a maximum of 65% ($35.8 to $59.0). In the case of a *Size-2* datacenter, it can reach a maximum of 53% ($294.3 to $451.2), and 54% ($580.1 to $895.8) in a *Size-3* configuration. When the comparison is done starting from the point where VMs are rejected by the base strategy, the medium increase in revenue is 45%, 40%, and 31%, for each datacenter configuration. This corresponds to an average increase in revenue of 39%, when considering all revenue increases across all datacenters.

We also compare the scheduling heuristics with a *random* and an *optimal* allocation. The *random* method chooses the server according to a random order. At a given server it will iterate over a random number of the available VMs (at most 50%), until it can take the necessary resources. This strategy stays below or slightly above Base+OverSub (which also does not reject VMs) but exhibits worse results than any of our heuristics. The *optimal* allocation was determined by exhaustively testing all the combinations of resource reallocation (a very slow process) and at each step choosing the one with better revenue. Our two main partial utility-driven heuristics are the ones that come closer to this allocation.

8.3.2 Effects on workloads

Finally, and regarding the execution time, we have evaluated the scheduling of VM resources to each profile based on the partial utility. The data used was collected from workloads executed during 10 days by thousands of PlanetLab VMs provisioned for multiple users [Beloglazov and Buyya, 2012; Park and Pai, 2006]. Each of these workloads are represented by traces with the percentage of CPU usage of a given VM running in the PlanetLab network, during a day. We use n of these workloads where n is the number of requested VMs. In our simulation environment, each trace is assigned to a single VM allocated with each strategy.

Figures 8.9 and 8.10 report on the CPU time used by the workloads running on the allocated VMs. The CPU time is based on the simulation clock managed by CloudSim. The average execution time of the workloads in each VM is presented in Figures 8.9 and 8.9 for the three datacenter sizes. The median execution time of the workloads in each VM is presented in Figures 8.10 and 8.10.



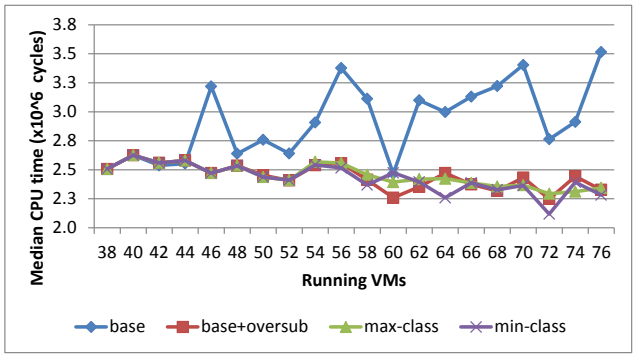
(a) Size-1 datacenter.

(b) Size-2 datacenter. 2 core/VM

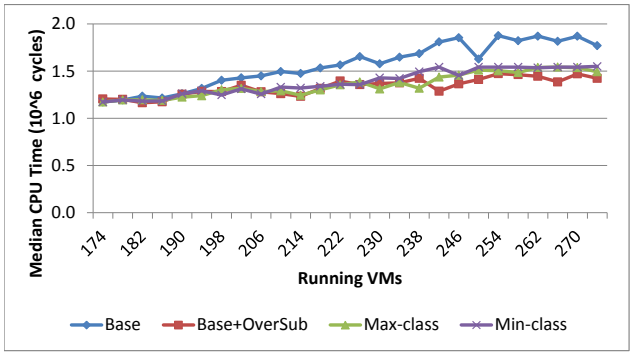
(c) Size-3 datacenter. 2 core/VM

(d) Size-3 datacenter. 4 core/VM

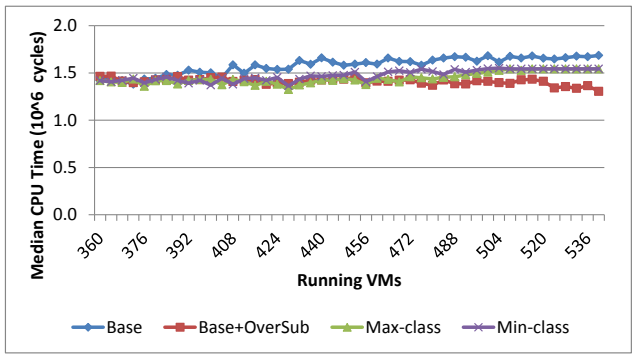
Figure 8.9: Compared average execution time of traces from PlaneLab VMs using datacenters with different sizes



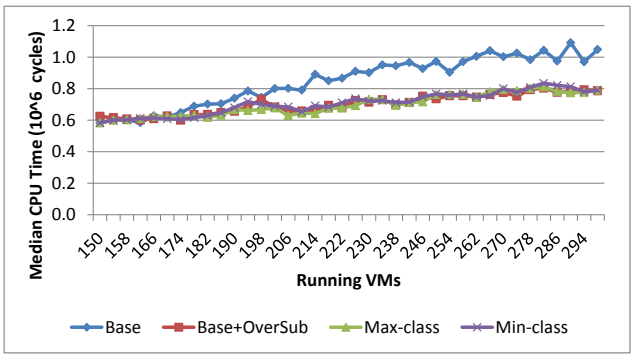
(a) Size-1 datacenter.



(b) Size-2 datacenter. 2 core/VM



(c) Size-3 datacenter. 2 core/VM



(d) Size-3 datacenter. 4 core/VM

Figure 8.10: Compared median execution time of traces from PlaneLab VMs using datacenters with different sizes

In the base strategy, as some requested VMs will be rejected because no host can be found with the complete requirements, there will be VMs that receive more than one PlanetLab VM trace. In the simulation, when these PlanetLab VMs are being reproduced, they receive a fraction of the available CPU, proportionally to the number of co-located PlanetLab VMs.

The results show that with more VMs allocated, even if with less allocated resources than the ones requested, as it is the case, both the average and the median execution time of tasks running on VMs allocated with our partial utility-driven scheduler is below the execution times achieved with the base strategy.

When comparing Base+OverSub with our best strategy (i.e., min-class), we can observe that the former has a marginally better average execution time while the latter has a slightly better median, rendering the differences seemingly non-significant. Nevertheless, as shown before in Section 8.3.1, the Base+OverSub strategy is unable to reach the best revenue for the provider and cannot provide any economic benefits for the clients given its utility obliviousness.

Summary

In this chapter we presented an extensive set of results regarding the evaluation of our partial utility-driven allocation of VMs. We analyzed provider-side metrics such as the number of virtual machines requested but not fulfilled, the percentage of resource allocation and the revenue. The client perspective was also examined, looking at the execution time spent by different workloads when running on the allocated VMs. The results are based on experiments performed on the CloudSim simulation environment.

Part IV

Conclusions and Future Work

9 Conclusions and Future Work

"The whole is more than the sum of its parts."

Aristotle in Metaphysical

Contents

The idea of computing as an utility is being materialized in what is called Cloud Computing. Industry and academia have embraced this new space of opportunities. While still dominated by information technology giants, such as Amazon and Google, with millions of clients across the globe, new companies are exploring this area of business, gaining new clients by their cost model or differentiating services.[1,2] In academia, many of the research groups working on related areas, such as Grid and Cluster environments, are now extending their efforts to the specific challenges of Cloud Computing, namely, virtualization technologies, new cost and revenue models and the big data challenges. The work presented on this thesis has embraced the first two topics.

The Cloud Computing space is largely divided into three service layers:

i) Infrastructure-as-a-Service (IaaS), where assets are virtual physical resources;

ii) Platform-as-a-Service (PaaS), where assets are managed runtimes;

[1] `http://www.lunacloud.com`, visited September 22, 2014
[2] `http://www.jelastic.com`, visited September 22, 2014

iii) Software-as-a-Service (SaaS), where assets are complete or customizable applications.

This thesis is centered on the first two layers proposing new models, algorithms, and mechanisms to be used by providers for the distribution of resources among different tenants.

To better understand current proposals regarding resource allocation in virtual machines (VMs), the first contribution is a classification process that allows different systems to be put into perspective regarding the techniques they used to monitor, decide and act. They are compared in three axis that we argue are independent, namely, responsiveness, comprehensiveness, and intricateness.

In the case of PaaS providers, it is possible to collect application-specific information about the workload running at each moment. We propose to use this information to avoid an equal distribution of resources and instead apply an application-aware allocation that can distribute resources where they are more effective. Our resource allocation model takes into account application *progress* and, when resources are scarce, favors tenants which bring greater *yield*, that is, the ones whose workloads will suffer less performance degradation when deprived of resources.

Regarding the IaaS service model, we have also considered a scenario where not all VM requests can be fulfilled. Our goal was to make overcommitment an explicit variable of the cost model. This way the consumer is easily able to determine the valuation of having its requests partially fulfilled. That is, we want to make clear its inherent and unavoidable drawbacks, but also its opportunities for some users with more relaxed requirements that are paying more than they could today. This variable should not be a hidden, exogenous one, only observable in the limit case scenario of Service Level Agreements violations.

9.1 Platform-as-a-Service

In this thesis, we described the research to design a new execution environment to be used in consolidated PaaS environments. Our goal was to manage resources and use an adaptation strategy that obeys a VM economics model, based on aiming overall quality-of-execution (QoE) through resource efficiency. Essentially, our goal was to put re-

sources where they can do the most good to applications and the cloud infrastructure provider, while taking them from where they can do the least harm to applications.

To fulfill this vision, each managed runtime instance is an extended resource-aware runtime for a high-level language, Java, where resources can be allocated based on their effectiveness for a given workload. QoE-JVM has the ability to monitor base mechanisms (e.g. CPU, memory or network consumptions) in order to assess application performance, so that these mechanisms can be reconfigured at runtime.

We had to devise an allocation model and incorporate resource management mechanisms which are not available in current managed runtimes. Regarding the mechanisms, a Java VM was extended with:

i) Resource accounting mechanisms and APIs. This includes an implementation of the JSR-284 and new matrices to regulate the heap rezising process.

ii) Concurrent checkpointing capabilities. We have extended a serial checkpointing mechanism to operate in concurrency with the application. It can be activated externally to the application in a transparent way.

Moreover, a program-level progress monitoring framework was also designed and implemented. The framework is based on counting the frequency of calls to critical methods as a proxy to determining the application's progress, so that this information can be used in the adaptation model.

We presented the details of our adaptation mechanisms in each VM. Jikes RVM, a Java VM, was extended to incorporate them. Regarding the general resources of heap size and CPU allocation, transversal to a large set of applications, we experimentally evaluated the benefits of our approach, showing resources can be transferred among applications, from where they hurt performance the least (higher yields in our metrics), to more higher priority or requirements applications.

9.2 Infrastructure-as-a-Service

SLA violations (and their avoidance) is a well established field of research. Cloud providers now offer no multi-level SLA agreements, and therefore, the user pays full

price for anything that does not violate SLA (that, for that matter, ends up being carefully handcrafted to be seldom violated). If the user gets fewer resources than the maximum requested, he should pay less for it, instead of having a much more conservative SLA that is never violated. Furthermore, if the user is willing to do this, to take this risk, her price should also be cut down.

Our goal was to build an SLA model that was not based on outages/unavailability but on the actual ability to satisfy the physical resources that were requested. If some VM has to have its resources released for the allocation of further requests, this is immediately taken into account in the pricing. For this risk, the user gets a premium, which is how we make him aware of the overcommitment. If the requested VM is so simple that it will perform acceptably even if severely under-provisioned, it means it belongs to a lower SLA class, with a unitary price that is lower, but with little reduction in performance if full resources are not provided.

In this thesis, we have proposed a cost model for IaaS providers that takes into account the user's partial utility specification when the provider needs to transfer resources between VMs. As the consolidation ratio increases, overcommitment will naturally happen and performance will be prone to degradation. Our model makes overcommitment (and higher consolidation factors) explicit to the users and rewards them, with more flexible and reduce pricing, for the upfront possibility of getting fewer resources than that they have payed for. This way, users pay more closely to what they consume (by what is made available to them) than for what they rent in contract (regardless of the actual physical resources that are made available to them).

We developed extensions to the scheduling policies of a state-of-the-art cloud infrastructure simulator, CloudSim, that are driven by this model. The cost model and partial utility-driven strategies were applied to the oversubscription of CPU. The provider's revenue, resource utilization and client's workloads execution time were measured. Results show that, although our strategies partially degraded and release the computational power of VMs when resources are scarce, they overcome the classic allocation strategy which would not be able to allocate above a certain number of VMs.

With this work we are a step forward in having a price model akin to paying for actual electricity consumption instead of paying for the maximum electrical power reserved contracted, such as a rent. This a step forward toward real pay-per-use, and to a true utility-like scheme.

9.3 Future Work

Several topics can be regarded as future work. There are new research directions that have the potential to bring benefits to our system. There are also improvements to the current design choices. The following are examples that fit into these two areas.

Statistical methods applied to the selection of heap growth matrices. Statistical learning methods, also known as machine learning, can be used to select the matrix that best fits an unknown workload. For each of the current evaluated workloads, a classification of their static and dynamic structure is described in the DaCapo benchmark paper [Blackburn et al., 2006]. Each of the metrics has the potential to be an analysis variable of a supervised learning experiments. Using these variables and the performance results of using different matrices, we could then use the results from previous learning experiments and choose what is expected to be the best matrix for the current workload.

Further integration with distributed scheduling middlewares. In [Simão et al., 2011], we have presented a middleware related to this thesis. It is a single-system image middleware that distributes threads across a set of nodes, giving the illusion to the application of a single address space. We discuss how the distribution policies of the middleware could be controlled by the resource management API available in our extended HLL-VM. In [Simão et al., 2013], this middleware was extended so that the placement of new threads takes into account the load of each node but also the benefits (*yield*) it brings to the overall workload execution. Although these two works were not detailed in this thesis, we consider this line of work to be a valuable one.

Future work must take into account that in multi-threaded application there will be threads interacting among themselves more than with others. Thread interaction can be measured by counting the number (frequency, recency, etc.) of accesses to objects protected by monitors. Identifying these threads is important to co-locate them when threads are spawned over the cluster. Also, during the execution of long running programs, highly correlated threads could be migrated, in groups, to less loaded nodes. The techniques used in our recent work [Silva et al., 2013] about the recording and replay of threads' concurrent access to object fields have the potential to also be used for this propose because they provide a view of thread affinity.

Integration with state of the art cloud scheduling simulators. Currently, state-of-the-art cloud scheduling simulators, such as CloudSim [Calheiros et al., 2011], assume that workloads make progress only based on the assigned CPU (i.e., number of instructions per second). This is useful for simulation because it contributes to the determinism of each experiment and to a fair comparison between strategies. However, we think that the simulator progress model can be made more realistic by accounting for execution degradation due to memory or even bandwidth shortage, thrashing or congestion. There must be further research concerning how we can simulate such behavior, taking into account progress measurement versus allocated resources, based on real workloads.

Experimentations with open source IaaS stacks. We plan to incorporate this approach in open source private cloud solutions such as OpenStack and extend the evaluation of the model to other resources, namely the network bandwidth.[3]

[3] `http://www.openstack.org/`, visited October 8, 2014

References

Adams, K. and Agesen, O. (2006). A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 2–13, New York, NY, USA. ACM.

Agmon Ben-Yehuda, O., Posener, E., Ben-Yehuda, M., Schuster, A., and Mu'alem, A. (2014a). Ginseng: Market-driven memory allocation. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '14, pages 41–52, New York, NY, USA. ACM.

Agmon Ben-Yehuda, O., Posener, E., Ben-Yehuda, M., Schuster, A., and Mu'alem, A. (2014b). Ginseng: Market-driven memory allocation. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '14, pages 41–52, New York, NY, USA. ACM.

Alpern, B., Attanasio, C., Barton, J., Burke, M., Cheng, P., Choi, J., Cocchi, A., Fink, S., Grove, D., Hind, M., et al. (2000). The Jalapeno virtual machine. *IBM Systems Journal*, 39(1):211.

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

Anderson, D. P. (2004). Boinc: A system for public-resource computing and storage. In

Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing, GRID '04, pages 4–10, Washington, DC, USA. IEEE Computer Society.

Andreasson, E., Hoffmann, F., and Lindholm, O. (2002). To collect or not to collect? Machine learning for memory management. In *Proceedings of the 2nd Java Virtual Machine Research and Technology Symposium*, pages 27–39, Berkeley, CA, USA. USENIX Association.

Andrzejak, A., Kondo, D., and Yi, S. (2010). Decision model for cloud computing under SLA constraints. In *Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), 2010 IEEE International Symposium on*, pages 257–266.

Armbrust, M., Fox, A., Grifth, R., Joseph, A. D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., and Zaharia, M. (2009). Above the clouds: A berkeley view of cloud computing. Technical report, UC Berkeley.

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*, pages 449–466.

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

Back, G., Hsieh, W. C., and Lepreau, J. (2000). Processes in kaffeos: Isolation, resource management, and sharing in java. In *In Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, pages 333–346.

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

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.

Baumgart, I., Heep, B., and Krause, S. (2007). Oversim: A flexible overlay network simulation framework. In *IEEE Global Internet Symposium, 2007*, pages 79–84.

Bedaride, P., Degomme, A., Genaud, S., Legrand, A., Markomanolis, G., Quinson, M., Stillwell, Mark, L., Suter, F., and Videau, B. (2013). Toward Better Simulation of MPI

Applications on Ethernet/TCP Networks. In *PMBS13 - 4th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*, pages 158–181, Denver, États-Unis.

Beloglazov, A. and Buyya, R. (2010). Energy efficient resource management in virtualized cloud data centers. In *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, pages 826–831.

Beloglazov, A. and Buyya, R. (2012). Optimal online deterministic algorithms and adaptive heuristics for energy and performance efficient dynamic consolidation of virtual machines in cloud data centers. *Concurrency and Computation: Practice and Experience*, 24(13):1397–1420.

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, S. M., Cheng, P., and McKinley, K. S. (2004). Oil and Water? High Performance Garbage Collection in Java with MMTk. In Finkelstein, A., Estublier, J., and Rosenblum, D. S., editors, *ICSE*, pages 137–146. IEEE Computer Society.

Blackburn, S. M., Garner, R., Hoffmann, C., Khang, A. M., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J. E. B., Moss, B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., 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 programming systems, languages, and applications*, pages 169–190, New York, NY, USA. ACM.

Blackburn, S. M. and McKinley, K. S. (2008). Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 22–32, New York, NY, USA. ACM.

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.

Bobroff, N., Westerink, P., and Fong, L. (2014). Active control of memory for Java virtual machines and applications. In *11th International Conference on Autonomic Computing (ICAC 14)*, pages 97–103, Philadelphia, PA. USENIX Association.

Bonér, 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.

Bouchenak, S., Hagimont, D., Krakowiak, S., De Palma, N., and Boyer, F. (2004). Experiences implementing efficient Java thread serialization, mobility and persistence. *Software: Practice and Experience*, 34(4):355–393.

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

Brown, R. H. F. and Horspool, R. N. (2010). Local redundant polymorphism query elimination. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, PPPJ '10, pages 78–88, New York, NY, USA. ACM.

Buyya, R., Garg, S. K., and Calheiros, R. N. (2011). Sla-oriented resource provisioning for cloud computing: Challenges, architecture, and solutions. In *Proceedings of the 2011 International Conference on Cloud and Service Computing*, CSC '11, pages 1–10, Washington, DC, USA. IEEE Computer Society.

Buyya, R. and Murshed, M. (2002). GridSim: a toolkit for the modeling and simulation of distributed resource management and scheduling for Grid computing. *Concurrency and Computation: Practice and Experience*, 14(13-15):1175–1220.

Buyya, R., Yeo, C. S., Venugopal, S., Broberg, J., and Brandic, I. (2009). Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Gener. Comput. Syst.*, 25(6):599–616.

Calheiros, R. N. and Buyya, R. (2014). Meeting deadlines of scientific workflows in public clouds with tasks replication. *IEEE Transactions on Parallel and Distributed Systems*, 25(7):1787–1796.

Calheiros, R. N., Ranjan, R., Beloglazov, A., De Rose, C. A. F., and Buyya, R. (2011). Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Softw. Pract. Exper.*, 41(1):23–50.

Cameron, C. and Singer, J. (2014). We are all economists now: Economic utility for multiple heap sizing. In *Proceeding of Implementation, Compilation, Optimization of OO Languages, Programs and Systems (ICOOOLPS)*.

Carvalho, F. and Cachopo, J. (2011). Stm with transparent api considered harmful. In Xiang, Y., Cuzzocrea, A., Hobbs, M., and Zhou, W., editors, *Algorithms and Architectures for Parallel Processing*, volume 7016 of *Lecture Notes in Computer Science*, pages 326–337. Springer Berlin Heidelberg.

Casanova, H., Legrand, A., and Quinson, M. (2008). Simgrid: a generic framework for large-scale distributed experiments. In *Proceedings of the Tenth International Conference on Computer Modeling and Simulation*, UKSIM '08, pages 126–131, Washington, DC, USA. IEEE Computer Society.

Castro Fernandez, R., Migliavacca, M., Kalyvianaki, E., and Pietzuch, P. (2013). Integrating scale out and fault tolerance in stream processing using operator state management. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 725–736, New York, NY, USA. ACM.

Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., and Sarkar, V. (2005). X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 519–538, New York, NY, USA. ACM.

Chase, J. S., Anderson, D. C., Thakar, P. N., Vahdat, A. M., and Doyle, R. P. (2001). Managing energy and server resources in hosting centers. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 103–116, New York, NY, USA. ACM.

Chen, L., Serazzi, G., Ansaloni, D., Smirni, E., and Binder, W. (2014). What to expect when you are consolidating: effective prediction models of application performance on multicores. *Cluster Computing*, 17(1):19–37.

Cheng, L. and Wang, C.-L. (2012). vbalance: Using interrupt load balance to improve i/o performance for smp virtual machines. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 2:1–2:14, New York, NY, USA. ACM.

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

Chiu, D.-M. and Jain, R. (1989). Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Comput. Netw. ISDN Syst.*, 17(1):1–14.

Chun, B., Culler, D., Roscoe, T., Bavier, A., Peterson, L., Wawrzoniak, M., and Bowman, M. (2003). Planetlab: An overlay testbed for broad-coverage services. *SIGCOMM Comput. Commun. Rev.*, 33(3):3–12.

Clark, C., Fraser, K., Hand, S., Hansen, J. G., Jul, E., Limpach, C., Pratt, I., and Warfield, A. (2005). Live migration of virtual machines. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 273–286, Berkeley, CA, USA. USENIX Association.

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.

Cobb, C. and Douglas, P. (1928). A theory of production. *The American Economic Review*, 18(1):139–165.

Costache, S., Parlavantzas, N., Morin, C., and Kortas, S. (2013). On the use of a proportional-share market for application SLO support in clouds. In *Euro-Par 2013 Parallel Processing*, volume 8097 of *Lecture Notes in Computer Science*, pages 341–352. Springer Berlin Heidelberg.

Coulson, G., Blair, G., Grace, P., Taiani, F., Joolia, A., Lee, K., Ueyama, J., and Sivaharan, T. (2008). A generic component model for building systems software. *ACM Trans. Comput. Syst.*, 26:1–42.

Cushing, R., Koulouzis, S., Belloum, A. S. Z., and Bubak, M. (2011). Prediction-based auto-scaling of scientific workflows. In *Proceedings of the 9th International Workshop on Middleware for Grids, Clouds and e-Science*, MGC '11, pages 1–6, New York, NY, USA. ACM.

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

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.

Czajkowski, G., Wegiel, M., Daynes, L., Palacz, K., Jordan, M., Skinner, G., and Bryce, C. (2005b). Resource management for clusters of virtual machines. In *Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid - Volume 01*, CCGRID '05, pages 382–389, Washington, DC, USA. IEEE Computer Society.

Dawoud, W., Takouna, I., and Meinel, C. (2012). Dynamic scalability and contention prediction in public infrastructure using internet application profiling. In *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on*, pages 208–216.

Desprez, F. and Rouzaud-Cornabas, J. (2013). SimGrid Cloud Broker: Simulating the Amazon AWS Cloud. Research Report RR-8380, INRIA.

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.

dos Reis, V. Q. and Cerqueira, R. (2010). Controlling processing usage at user level: a way to make resource sharing more flexible. *Concurrency and Computation: Practice and Experience*, 22(3):278–294.

Dragojević, A., Narayanan, D., Castro, M., and Hodson, O. (2014). Farm: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, Seattle, WA. USENIX Association.

Du, J., Sehrawat, N., and Zwaenepoel, W. (2011). Performance Profiling of Virtual Machines. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '11, pages 3–14.

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.

Estrada, T., Taufer, M., Reed, K., and Anderson, D. (2009). Emboinc: An emulator for performance analysis of boinc projects. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–8.

European Commission (2012). Unleashing the potential of cloud computing in europe. http://eur-lex.europa.eu/.

Ferreira, P., Veiga, L., and Ribeiro, C. (2003). OBIWAN: design and implementation of a middleware platform. *Parallel and Distributed Systems, IEEE Transactions on*, 14(11):1086–1099.

Fink, S. and Qian, F. (2003). Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In *Code Generation and Optimization, 2003. CGO 2003. International Symposium on*, pages 241–252.

Frampton, D., Blackburn, S. M., Cheng, P., Garner, R. J., Grove, D., Moss, J. E. B., and Salishev, S. I. (2009). Demystifying magic: High-level low-level programming. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '09, pages 81–90, New York, NY, USA. ACM.

Fries, A. (2012). *The use of Java in large scientific applications in HPC environments*. PhD thesis, Universitat de Barcelona.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Garey, M. R., Graham, R. L., and Johnson, D. S. (1976). Resource constrained scheduling as generalized bin packing. *J. Comb. Theory, Ser. A*, 21(3):257–298.

Garfinkel, S. (1999). *Architects of the Information Society*. MIT press.

Garg, S. and Buyya, R. (2011). NetworkCloudSim: Modelling parallel applications in cloud simulations. In *Utility and Cloud Computing (UCC), 2011 Fourth IEEE International Conference on*, pages 105–113.

Garrochinho, T. (2010). CRM-HLL-VM: Checkpoint, Restore, Migração em Máquinas Virtuais Java. Master's thesis, Instituro Superior Técnico.

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*, pages 544–553.

Georges, A., Eeckhout, L., and Buytaert, D. (2008). Java performance evaluation through rigorous replay compilation. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, OOPSLA '08, pages 367–384. ACM.

Gidra, L., Thomas, G., Sopena, J., and Shapiro, M. (2013). A study of the scalability of stop-the-world garbage collectors on multicores. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 229–240, New York, NY, USA. ACM.

Goldberg, R. P. (1974). Survey of virtual machine research. *Computer*, 7(9):34–45.

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.

Gordon, A., Amit, N., Har'El, N., Ben-Yehuda, M., Landau, A., Schuster, A., and Tsafrir, D. (2012). ELI: Bare-metal performance for I/O virtualization. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 411–422, New York, NY, USA. ACM.

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

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.

Grzegorz Czajkowski (2009). Java Specification Request 284 - Resource Consumption Management API, http://jcp.org/en/jsr/detail?id=284. Java Community Process.

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, A., Merchant, A., Uysal, M., and Varman, P. J. (2007). Efficient and adaptive proportional share I/O scheduling. Technical report, HP Laboratories Palo Alto.

Gupta, D., Lee, S., Vrable, M., Savage, S., Snoeren, A. C., Varghese, G., Voelker, G. M., and Vahdat, A. (2008). Difference engine: harnessing memory redundancy in virtual machines. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 309–322, Berkeley, CA, USA. USENIX Association.

Hagimont, D., Mayap Kamga, C., Broto, L., Tchana, A., and Palma, N. (2013). DVFS aware CPU credit enforcement in a virtualized system. In *Middleware 2013*, volume 8275 of *Lecture Notes in Computer Science*, pages 123–142. Springer Berlin Heidelberg.

Halappanavar, M., Feo, J., Villa, O., Tumeo, A., and Pothen, A. (2012). Approximate weighted matching on emerging manycore and multithreaded architectures. *Int. J. High Perform. Comput. Appl.*, 26(4):413–430.

Hargrove, P. and Duell, J. (2006). Berkeley lab checkpoint/restart (BLCR) for linux clusters. Technical Report 60520, Lawrence Berkeley National Laboratory.

Hauswirth, M., Sweeney, P. F., and Diwan, A. (2010). Temporal vertical profiling. *Software Practice and Experience*, 40(8):627–654.

Hayden, C. M., Smith, E. K., Denchev, M., Hicks, M., and Foster, J. S. (2012). Kitsune: Efficient, general-purpose dynamic software updating for c. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 249–264, New York, NY, USA. ACM.

Heilig, L. and Vob, S. (2014). A scientometric analysis of cloud computing literature. *Cloud Computing, IEEE Transactions on*, 2(3):266–278.

Heo, J., Zhu, X., Padala, P., and Wang, Z. (2009). Memory overbooking and dynamic control of xen virtual machines in consolidated environments. In *Proceedings of the 11th IFIP/IEEE international conference on Symposium on Integrated Network Management*, IM'09, pages 630–637, Piscataway, NJ, USA. IEEE Press.

Heroku (2014). Heroku - cloud computing designed and built for developers.

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

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.

Hiden, H., Woodman, S., Watson, P., and Cala, J. (2013). Developing cloud applications using the e-science central platform. *Phil Trans Royal Society*, 371:1983.

Hinesa, 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*, pages 130–137.

Hirofuchi, T., Lèbre, A., and Pouilloux, L. (2013). Adding a live migration model into simgrid: One more step toward the simulation of infrastructure-as-a-service concerns. In *Proceedings of the 2013 IEEE International Conference on Cloud Computing Technology and Science - Volume 01*, CLOUDCOM '13, pages 96–103, Washington, DC, USA. IEEE Computer Society.

Hirofuchi, T. and Lebre, A. (2013). Adding virtual machine abstractions into simgrid: A first step toward the simulation of infrastructure-as-a-service concerns. In *Proceedings of the 2013 International Conference on Cloud and Green Computing*, CGC '13, pages 175–180, Washington, DC, USA. IEEE Computer Society.

Hoffmann, H., Eastep, J., Santambrogio, M. D., Miller, J. E., and Agarwal, A. (2010). Application heartbeats: a generic interface for specifying program performance and goals in autonomous computing environments. In *Proceedings of the 7th international conference on Autonomic computing*, ICAC '10, pages 79–88.

Hoffmann, H., Sidiroglou, S., Carbin, M., Misailovic, S., Agarwal, A., and Rinard, M. (2011). Dynamic knobs for responsive power-aware computing. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '11, pages 199–212.

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, 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.

IBM (2005). An architectural blueprint for autonomic computing. Technical report, IBM.

Ishakian, V., Sweha, R., Bestavros, A., and Appavoo, J. (2012). CloudPack: Exploiting workload flexibility through rational pricing. In *Middleware 2012*, volume 7662 of *Lecture Notes in Computer Science*, pages 374–393. Springer Berlin Heidelberg.

Issariyakul, T. and Hossain, E. (2008). *Introduction to Network Simulator NS2*. Springer Publishing Company, Incorporated, 1st edition.

Janik, A. and Zielinski, K. (2010). AAOP-based dynamically reconfigurable monitoring system. *Information & Software Technology*, 52(4r):380–396.

Jin, H., Wang, X., Wu, S., Di, S., and Shi, X. (2014). Towards optimized fine-grained pricing of iaas cloud platform. *Cloud Computing, IEEE Transactions on*, PP(99):1–1.

Jones, R., Hosking, A., and Moss, E. (2011). *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall/CRC, 1st edition.

Kächele, S. and Hauck, F. J. (2013). Component-based scalability for cloud applications. In *Proceedings of the 3rd International Workshop on Cloud Data and Platforms*, CloudDP '13, pages 19–24, New York, NY, USA. ACM.

Kesavan, M., Gavrilovska, A., and Schwan, K. (2010). On disk i/o scheduling in virtual machines. In *Proceedings of the 2nd conference on I/O virtualization*, WIOV'10, pages 6–6, Berkeley, CA, USA. USENIX Association.

Khan, A., Buyuksahin, U., and Freitag, F. (2014). Prototyping incentive-based resource assignment for clouds in community networks. In *Advanced Information Networking and Applications (AINA), 2014 IEEE 28th International Conference on*, pages 719–726.

Khan, A., Navarro, L., Sharifi, L., and Veiga, L. (2013a). Clouds of small things: Provisioning infrastructure-as-a-service from within community networks. In *Wireless and Mobile Computing, Networking and Communications (WiMob), 2013 IEEE 9th International Conference on*, pages 16–21.

Khan, A. M., Navarro, L., Sharifi, L., and Veiga, L. (2013b). Clouds of small things: Provisioning infrastructure-as-a-service from within community networks. In *WiMob*, pages 16–21.

Khosravi, A., Garg, S. K., and Buyya, R. (2013). Energy and carbon-efficient placement of virtual machines in distributed cloud data centers. In Wolf, F., Mohr, B., and an Mey, D., editors, *Euro-Par*, volume 8097 of *Lecture Notes in Computer Science*, pages 317–328. Springer.

Kim, N., Cho, J., and Seo, E. (2014). Energy-credit scheduler: An energy-aware virtual machine scheduler for cloud systems. *Future Generation Computer Systems*, 32(0):128 – 137.

Kliazovich, D., Bouvry, P., Audzevich, Y., and Khan, S. (2010). Greencloud: A packet-level simulator of energy-aware cloud computing data centers. In *Global Telecommunications Conference (GLOBECOM 2010), 2010 IEEE*, pages 1–5.

Krampis, K., Booth, T., Chapman, B., Tiwari, B., Bicak, M., Field, D., and Nelson, K. E. (2012). Cloud biolinux: pre-configured and on-demand bioinformatics computing for the genomics community. *BMC Bioinformatics*, 13:42.

Kulkarni, S. and Cavazos, J. (2012). Mitigating the compiler optimization phase-ordering problem using machine learning. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 147–162, New York, NY, USA. ACM.

Kumar, V., Frampton, D., Blackburn, S. M., Grove, D., and Tardieu, O. (2012). Work-stealing without the baggage. *SIGPLAN Not.*, 47(10):297–314.

Lagar-Cavilla, H. A., Whitney, J. A., Bryant, R., Patchin, P., Brudno, M., de Lara, E., Rumble, S. M., Satyanarayanan, M., and Scannell, A. (2011). Snowflock: Virtual machine cloning as a first-class cloud primitive. *ACM Trans. Comput. Syst.*, 29(1):2:1–2:45.

Lam, K. T., Luo, Y., and Wang, C.-L. (2010). Adaptive sampling-based profiling techniques for optimizing the distributed jvm runtime. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1 –11.

Le Sueur, E. and Heiser, G. (2010). Dynamic voltage and frequency scaling: The laws of diminishing returns. In *Proceedings of the 2010 International Conference on Power Aware Computing and Systems*, HotPower'10, pages 1–8. USENIX Association.

Lèbre, A., Pastor, J., Bertier, M., Desprez, F., Rouzaud-Cornabas, J., Tedeschi, C., Anedda, P., Zanetti, G., Nou, R., Cortes, T., Rivière, E., and Ropars, T. (2013). Beyond The Cloud, How Should Next Generation Utility Computing Infrastructures Be Designed? Rapport de recherche RR-8348, INRIA.

León, X. and Navarro, L. (2013). A stackelberg game to derive the limits of energy savings for the allocation of data center resources. *Future Generation Computer Systems*, 29(1):74–83.

Li, D., Srisa-an, W., and Dwyer, M. B. (2011). Sos: Saving time in dynamic race detection with stationary analysis. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 35–50, New York, NY, USA. ACM.

Li, J., Shuang, K., Su, S., Huang, Q., Xu, P., Cheng, X., and Wang, J. (2012). Reducing operational costs through consolidation with resource prediction in the cloud. In *Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGRID '12, pages 793–798, Washington, DC, USA. IEEE Computer Society.

Libič, P., Bulej, L., Horky, V., and Tůma, P. (2014). On the limits of modeling generational garbage collector performance. In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering*, ICPE '14, pages 15–26, New York, NY, USA. ACM.

Liu, J., Goraczko, M., James, S., Belady, C., Lu, J., and Whitehouse, K. (2011). The data furnace: heating up with cloud computing. In *Proceedings of the 3rd USENIX conference on Hot topics in cloud computing*, HotCloud'11, pages 15–15, Berkeley, CA, USA. USENIX Association.

Lublin, U., Kamay, Y., Laor, D., and Liguori, A. (2007). KVM: the Linux virtual machine monitor. In *Ottawa Linux Symposium*.

Ma, R., Wang, C., and Lau, F. (2002). M-JavaMPI: A Java-MPI binding with process migration support. *The Second IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 1–9.

Macias, M. and Guitart, J. (2014). A risk-based model for service level agreement differentiation in cloud market providers. In *Distributed Applications and Interoperable Systems*, Lecture Notes in Computer Science, pages 1–15. Springer Berlin Heidelberg.

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.

Makarov, D. and Hauswirth, M. (2013). Jikes RDB: a debugger for the Jikes RVM. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '13, pages 169–172, New York, NY, USA. ACM.

Malik, S., Khan, S., and Srinivasan, S. (2013). Modeling and analysis of state-of-the-art vm-based cloud management platforms. *Cloud Computing, IEEE Transactions on*, 1(1).

Mankiw, G. (2011). *Principles of Microeconomics*. Cengage Learning.

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

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.

Marinos, A. and Briscoe, G. (2007). Community cloud computing. In *First International Conference on Cloud Computing*, pages 472–484.

Mastroianni, C., Meo, M., and Papuzzo, G. (2013). Probabilistic consolidation of virtual machines in self-organizing cloud data centers. *Cloud Computing, IEEE Transactions on*, 1(2):215–228.

Mc Evoy, G. and Schulze, B. (2011). Understanding scheduling implications for scientific applications in clouds. In *Proceedings of the 9th International Workshop on Middleware for Grids, Clouds and e-Science*, MGC '11, pages 3:1–3:6, New York, NY, USA. ACM.

Medina, A., Lakhina, A., Matta, I., and Byers, J. (2001). BRITE: An approach to universal topology generation. In *Proceedings of the Ninth International Symposium in Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, MASCOTS '01, pages 116–135, Washington, DC, USA. IEEE Computer Society.

Meng, X., Isci, C., Kephart, J., Zhang, L., Bouillet, E., and Pendarakis, D. (2010). Efficient resource provisioning in compute clouds via vm multiplexing. In *Proceedings of the 7th International Conference on Autonomic Computing*, ICAC '10, pages 11–20, New York, NY, USA. ACM.

Mian, R., Martin, P., Zulkernine, F., and Vazquez-Poletti, J. L. (2012). Estimating resource costs of data-intensive workloads in public clouds. In *Proceedings of the 10th International Workshop on Middleware for Grids, Clouds and e-Science*, MGC '12, pages 3:1–3:6, New York, NY, USA. ACM.

Min, C., Kim, I., Kim, T., and Eom, Y. I. (2012). Vmmb: Virtual machine memory balancing for unmodified operating systems. *J. Grid Comput.*, 10(1):69–84.

Montresor, A. and Jelasity, M. (2009). PeerSim: A scalable P2P simulator. In *Proc. of the 9th Int. Conference on Peer-to-Peer (P2P'09)*, pages 99–100, Seattle, WA.

Morshedlou, H. and Meybodi, M. R. (2014). Decreasing impact of SLA violations: A proactive resource allocation approachfor cloud computing environments. *IEEE T. Cloud Computing*, 2(2):156–167.

Núñez, A., Vázquez-Poletti, J. L., Caminero, A. C., Castañé, G. G., Carretero, J., and Llorente, I. M. (2012). icancloud: A flexible and scalable cloud infrastructure simulator. *Journal of Grid Computing*, 10(1):185–209.

Ohad Shai, Edi Shmueli, D. G. F. (2013). Heuristics for resource matching in intel's compute farm. In *Proceedings of the 17th Workshop on Job Scheduling Strategies for Parallel Processing (co-located with IPDPS)*.

Ongaro, D., Cox, A. L., and Rixner, S. (2008). Scheduling I/O in virtual machine monitors. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '08, pages 1–10, New York, NY, USA. ACM.

Osman, S., Subhraveti, D., Su, G., and Nieh, J. (2002). The design and implementation of zap: A system for migrating computing environments. *SIGOPS Oper. Syst. Rev.*, 36(SI):361–376.

Ousterhout, J. K. (1982). Scheduling techniques for concurrent systems. In *ICDCS*, pages 22–30. IEEE Computer Society.

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.

Park, K. and Pai, V. S. (2006). Comon: a mostly-scalable monitoring system for planetlab. *SIGOPS Oper. Syst. Rev.*, 40(1):65–74.

Park, S.-M. and Humphrey, M. (2009). Self-tuning virtual machines for predictable escience. In *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, CCGRID '09, pages 356–363, Washington, DC, USA. IEEE Computer Society.

Peter Mell and Tim Grance (2009). The NIST Definition of Cloud Computing.

Peter Mell and Tim Grance (2011). The NIST Definition of Cloud Computing. Technical Report 800-145, National Institute of Standards and Technology, http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf.

Pierre, G. and Stratan, C. (2012). Conpaas: A platform for hosting elastic cloud applications. *IEEE Internet Computing*, 16(5):88–92.

Pina, L., Veiga, L., and Hicks, M. (2014). Rubah: DSU for java on a stock JVM. In *Proceedings of the ACM Conference on Object-Oriented Programming Languages, Systems, and Applications (OOPSLA)*.

Prakash, S. and Bagrodia, R. L. (1998). MPI-SIM: Using parallel simulation to evaluate MPI programs. In *Proceedings of the 30th Conference on Winter Simulation*, WSC '98, pages 467–474, Los Alamitos, CA, USA. IEEE Computer Society Press.

Price, D. W., Rudys, A., and Wallach, D. S. (2003). Garbage collector memory accounting in language-based systems. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, SP '03, pages 263–, Washington, DC, USA. IEEE Computer Society.

Quinson, M., Rosa, C., and Thiery, C. (2012). Parallel simulation of peer-to-peer systems. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2012)*, CCGRID '12, pages 668–675, Washington, DC, USA. IEEE Computer Society.

Quitadamo, R. and Leonardi, L. (2008). *The Issue of Strong Mobility: an Innovative Approach based on the IBM Jikes Research Virtual Machine*. PhD thesis, University of Modena and Reggio Emilia.

Ram, K. K., Santos, J. R., and Turner, Y. (2010). Redesigning xen's memory sharing mechanism for safe and efficient I/O virtualization. In *Proceedings of the 2Nd Conference on I/O Virtualization*, WIOV'10, Berkeley, CA, USA. USENIX Association.

Ranger, C., Raghuraman, R., Penmetsa, A., Bradski, G., and Kozyrakis, C. (2007). Evaluating mapreduce for multi-core and multiprocessor systems. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 13–24.

Ribeiro, C., Zúquete, A., Ferreira, P., and Guedes, P. (1999). Spl: An access control language for security policies with complex constraints. In *In Proceedings of the Network and Distributed System Security Symposium*, pages 89–107.

Rosenblum, M. (2004). The reincarnation of virtual machines. *Queue*, 2(5):34–40.

S. Zaman, D. G. (2011). Combinatorial auction-based dynamic VM provisioning and allocation in clouds. In IEEE, editor, *Third International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 107–114.

Sakamoto, T., Sekiguchi, T., and Yonezawa, A. (2000). Bytecode transformation for portable thread migration in java. In Kotz, D. and Mattern, F., editors, *Agent Systems,*

Mobile Agents, and Applications, volume 1882 of *Lecture Notes in Computer Science*, pages 16–28. Springer Berlin Heidelberg.

Sakellari, G. and Loukas, G. (2013). A survey of mathematical models, simulation approaches and testbeds used for research in cloud computing. *Simulation Modelling Practice and Theory*, 39:92–103.

Salehi, M. A., Javadi, B., and Buyya, R. (2013). Resource provisioning based on preempting virtual machines in distributed systems. *Concurrency and Computation: Practice and Experience*, 26(2):412–433.

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

Salomie, T.-I., Alonso, G., Roscoe, T., and Elphinstone, K. (2013). Application level ballooning for efficient server consolidation. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 337–350, New York, NY, USA. ACM.

Saovapakhiran, B. and Devetsikiotis, M. (2011). Enhancing computing power by exploiting underutilized resources in the community cloud. In *Proceedings of IEEE International Conference on Communications*, pages 1–6. IEEE.

Shahriyar, R., Blackburn, S. M., Yang, X., and McKinley, K. S. (2013). Taking off the gloves with reference counting immix. In Hosking, A. L., Eugster, P. T., and Lopes, C. V., editors, *OOPSLA*, pages 93–110. ACM.

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.

Sharifi, L., Rameshan, N., Freitag, F., and Veiga, L. (2014). Energy efficiency dilemma: P2p-cloud vs. datacenter. In *IEEE 6th International Conference on Cloud Computing Technology and Science*. to appear.

Silva, J., Ferreira, P., and Veiga, L. (2010a). Service and resource discovery in cycle-sharing environments with a utility algebra. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–11.

Silva, J. a. N., Veiga, L., and Ferreira, P. (2008). Heuristic for Resources Allocation on Utility Computing Infrastructures. In *Proceedings of the 6th international workshop on Middleware for grid computing*, MGC '08, pages 9:1–9:6, New York, NY, USA. ACM.

Silva, J. M., Simão, J., and Veiga, L. (2013). Ditto - deterministic execution replayability-as-a-service for java vm on multiprocessors. In Eyers, D. M. and Schwan, K., editors, *Middleware*, volume 8275 of *Lecture Notes in Computer Science*, pages 405–424. Springer.

Silva, J. N., Ferreira, P., and Veiga, L. (2010b). Service and resource discovery in cycle-sharing environments with a utility algebra. In *IPDPS*, pages 1–11. IEEE.

Silva, J. N., Veiga, L., and Ferreira, P. (2011). A^2HA - Automatic and adaptive host allocation in utility computing for bag-of-tasks. *J. Internet Services and Applications*, 2(2):171–185.

Simão, J., Garrochinho, T., and Veiga, L. (2012). A checkpointing-enabled and resource-aware Java Virtual Machine for efficient and robust e-Science applications in grid environments. *Concurrency and Computation: Practice and Experience*, 24(13):1421–1442.

Simão, J. and Veiga, L. (2012a). A classification of middleware to support virtual machines adaptability in iaas. In *Proceedings of the 11th International Workshop on Adaptive and Reflective Middleware*, ARM '12, pages 5:1–5:6, New York, NY, USA. ACM.

Simão, J. and Veiga, L. (2012b). VM Economics for Java Cloud Computing: An Adaptive and Resource-Aware Java Runtime with Quality-of-Execution. In *Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGRID '12, pages 723–728. IEEE Computer Society.

Simão, J., Lemos, J., and Veiga, L. (2011). A^2-VM a cooperative Java VM with support for resource-awareness and cluster-wide thread scheduling. In *19th International Conference on Cooperative Information Systems (CoopIS 2011)*. LNCS, Springer.

Simão, J., Rameshan, N., and Veiga, L. (2013). Resource-aware scaling of multi-threaded java applications in multi-tenancy scenarios. In *IEEE 5th International Conference on Cloud Computing Technology and Science, CloudCom 2013, Bristol, United Kingdom, December 2-5, 2013, Volume 1*, pages 445–451. IEEE.

Simão, J., Singer, J., and Veiga, L. (2013). A comparative look at adaptive memory management in virtual machines. In *IEEE CloudCom 2013*. IEEE.

Simão, J. and Veiga, L. (2012). QoE-JVM: An Adaptive and Resource-Aware Java Runtime for Cloud Computing. In Meersman, R., Panetto, H., Dillon, T. S., Rinderle-Ma, S., Dadam, P., Zhou, X., Pearson, S., Ferscha, A., Bergamaschi, S., and Cruz, I. F., editors, *OTM Conferences (2)*, volume 7566 of *Lecture Notes in Computer Science*, pages 566–583. Springer.

Simão, J. and Veiga, L. (2013a). Adaptability driven by quality of execution in high level virtual machines for shared cloud environments. *International Journal of Computer Systems Science & Engineering*, 29(6).

Simão, J. and Veiga, L. (2013b). Flexible SLAs in the cloud with a partial utility-driven scheduling architecture. In *CloudCom*, pages 275–281. IEEE.

Simão, J. and Veiga, L. (2013c). A progress and profile-driven cloud-VM for resource-efficiency and fairness in e-science environments. In Shin, S. Y. and Maldonado, J. C., editors, *SAC*, pages 357–362. ACM.

Simão, J. and Veiga, L. (2014). Partial Utility-driven Scheduling for Flexible SLA and Pricing Arbitration in Clouds. *IEEE Transactions on Cloud Computing*. to appear.

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, J. and Jones, R. (2011). Economic utility theory for memory management optimization. In Rogers, I., editor, *Proceedings of the workshop on Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, page 4. ACM. (Position paper).

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

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

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

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.

Son, S., Jung, G., and Jun, S. (2013). An SLA-based cloud computing that facilitates resource allocation in the distributed data centers of a cloud provider. *The Journal of Supercomputing*, 64(2):606–637.

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

Sukwong, O., Sangpetch, A., and Kim, H. (2012). SageShift: Managing SLAs for highly consolidated cloud. In *INFOCOM, 2012 Proceedings IEEE*, pages 208–216.

Suri, N., Bradshaw, J., Breedy, M., Groth, P., Hill, G., and Jeffers, R. (2000). Strong mobility and fine-grained resource control in nomads. In Kotz, D. and Mattern, F., editors, *Agent Systems, Mobile Agents, and Applications*, volume 1882 of *Lecture Notes in Computer Science*, pages 2–15. Springer Berlin Heidelberg.

Suri, N., Bradshaw, J. M., Breedy, M. R., Groth, P. T., Hill, G. A., and Saavedra, R. (2001). State capture and resource control for java: the design and implementation of the aroma virtual machine. In *Proceedings of the Symposium on JavaTM Virtual Machine Research and Technology Symposium*, JVM'01, pages 11–11, Berkeley, CA, USA. USENIX Association.

Sweeney, P. F., Hauswirth, M., Cahoon, B., Cheng, P., Diwan, A., Grove, D., and Hind, M. (2004). Using hardware performance monitors to understand the behavior of Java applications. In *Proceedings of the 3rd conference on Virtual Machine Research And Technology Symposium - Volume 3*, pages 5–5, Berkeley, CA, USA. USENIX Association.

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

Tay, Y. C., Zong, X., and He, X. (2013). An equation-based heap sizing rule. *Perform. Eval.*, 70(11):948–964.

Tejedor, E., Farreras, M., Grove, D., Badia, R. M., Almasi, G., and Labarta, J. (2012). A high-productivity task-based programming model for clusters. *Concurrency and Computation: Practice and Experience*, pages 2421–2448.

Tene, G., Iyengar, B., and Wolf, M. (2011). C4: The continuously concurrent compacting collector. *SIGPLAN Not.*, 46(11):79–88.

Tsakalozos, K., Kllapi, H., Sitaridi, E., Roussopoulos, M., Paparas, D., and Delis, A. (2011). Flexible use of cloud resources through profit maximization and price discrimination. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 75–86, Washington, DC, USA. IEEE Computer Society.

Vaquero, L. M., Rodero-Merino, L., and Buyya, R. (2011). Dynamically scaling applications in the cloud. *SIGCOMM Comput. Commun. Rev.*, 41(1):45–52.

Vaquero, L. M., Rodero-Merino, L., Caceres, J., and Lindner, M. (2008). A break in the clouds: Towards a cloud definition. *SIGCOMM Comput. Commun. Rev.*, 39(1):50–55.

Veiga, L., Silva, J. a. N., and Garcia, J. a. C. (2011). Peer4Peer: e-science community for network overlay and grid computing research. In Yang, X., Wang, L., and Jie, W., editors, *Guide to e-Science*, Computer Communications and Networks, pages 81–113. Springer London.

Velazquez-Garcia, F., Andersen, H., Hansen, H., Goebel, V., and Plagemann, T. (2013). Sockman: Socket migration for multimedia applications. In *Telecommunications (ConTEL), 2013 12th International Conference on*, pages 115–122.

Velho, P., Schnorr, L. M., Casanova, H., and Legrand, A. (2013). On the validity of flow-level TCP network models for grid and cloud simulations. *ACM Trans. Model. Comput. Simul.*, 23(4):23:1–23:26.

VMware (2009). VMware vSpher 4: The CPU Scheduler in VMware ESX 4.

Waldspurger, C. A. (2002). Memory resource management in VMware ESX server. *SIGOPS Oper. Syst. Rev.*, 36:181–194.

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.

Weng, C., Liu, Q., Yu, L., and Li, M. (2011). Dynamic adaptive scheduling for virtual machines. In *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, HPDC '11, pages 239–250, New York, NY, USA. ACM.

Weng, C., Wang, Z., Li, M., and Lu, X. (2009). The hybrid scheduling framework for virtual machine systems. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '09, pages 111–120, New York, NY, USA. ACM.

White, D. R., Singer, J., Aitken, J. M., and Jones, R. E. (2013). Control theory for principled heap sizing. In *Proceedings of the 2013 International Symposium on Memory Management*, ISMM '13, pages 27–38, New York, NY, USA. ACM.

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.

Xu, F., Liu, F., Jin, H., and Vasilakos, A. (2014). Managing performance overhead of virtual machines in cloud computing: A survey, state of the art, and future directions. *Proceedings of the IEEE*, 102(1):11–31.

Xu, H. and Li, B. (2013). Dynamic cloud pricing for revenue maximization. *Cloud Computing, IEEE Transactions on*, 1(2):158–171.

Yang, T., Berger, E. D., Kaplan, S. F., and Moss, J. E. B. (2006). Cramm: Virtual memory support for garbage-collected applications. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 103–116, Berkeley, CA, USA. USENIX Association.

Zhang, C., Kelsey, K., Shen, X., Ding, C., Hertz, M., and Ogihara, M. (2006). Program-level adaptive memory management. In *Proceedings of the 5th international symposium on Memory management*, ISMM '06, pages 174–183, New York, NY, USA. ACM.

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, L. and Krintz, C. (2005). The design, implementation, and evaluation of adaptive code unloading for resource-constrained devices. *ACM Trans. Archit. Code Optim.*, 2(2):131–164.

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.

Zheng, G., Gupta, G., Bohm, E., Dooley, I., and Kale, L. V. (2010). Simulating Large Scale Parallel Applications using Statistical Models for Sequential Execution Blocks. In *Proceedings of the 16th International Conference on Parallel and Distributed Systems (ICPADS 2010)*, number 10-15, Shanghai, China.

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.