Multifaceted Resource Management on Virtualized Providers

by

Íñigo Goiri

Advisors:
Jordi Guitart and Jordi Torres

A dissertation submitted in partial fulfillment of the requirements for the degree of:

Doctor per la Universitat Politècnica de Catalunya

Barcelona
June 2011
Abstract

Last decade, providers started using Virtual Machines (VMs) in their datacenters to pack users and their applications. This was a good way to consolidate multiple users in fewer physical nodes while isolating them from each other. Later on in 2006, Amazon started offering their Infrastructure as a Service where their users rent computing resources as VMs in a pay-as-you-go manner.

However, virtualized providers cannot be managed like traditional ones as they are now confronted with a set of new challenges. First of all, providers must deal efficiently with new management operations such as the dynamic creation of VMs. These operations enable new capabilities that were not there before, such as moving VMs across the nodes, or the ability to checkpoint VMs. We propose a Decentralized virtualization management infrastructure to create VMs on demand, migrate them between nodes, and checkpointing mechanisms. With the introduction of this infrastructure, virtualized providers become decentralized and are able to scale.

Secondly, these providers consolidate multiple VMs in a single machine to more efficiently utilize resources. Nevertheless, this is not straightforward and implies the use of more complex resource management techniques. In addition, this requires that both customers and providers can be confident that signed Service Level Agreements (SLAs) are supporting their respective business activities to their best extent. Providers typically offer very simple metrics that hinder an efficient exploitation of their resources. To solve this, we propose mechanisms to dynamically distribute resources among VMs and a resource-level metric, which together allow increasing provider utilization while maintaining Quality of Service.

Thirdly, the provider must allocate the VMs evaluating multiple facets such as power consumption and customers’ requirements. In addition, it must exploit the new capabilities introduced by virtualization and manage its overhead. Ultimately, this VM placement must minimize the costs asso-
associated with the execution of a VM in a provider to maximize the provider’s profit. We propose a new scheduling policy that places VMs on provider nodes according to multiple facets and is able to understand and manage the overheads of dealing with virtualization.

And fourthly, resource provisioning in these providers is a challenge because of the high load variability over time. Providers can serve most of the requests owning only a restricted amount of resources but this under-provisioning may cause customers to be rejected during peak hours. In the opposite situation, valley hours incur under-utilization of the resources. As this new paradigm makes the access to resources easier, providers can share resources to serve their loads. We leverage a federated scenario where multiple providers share their resources to overcome this load variability. We exploit the federation capabilities to create policies that take the most convenient decision depending on the environment conditions and tackle the load variability.

All these challenges mean that providers must manage their virtualized resources in a different way than they have done traditionally. This dissertation identifies and studies the challenges faced by virtualized provider that offers IaaS, and designs and evaluates a solution to manage the provider’s resources in the most cost-effective way by exploiting the virtualization capabilities.
Acknowledgements

I would like to express my gratitude to all the people who has made this thesis possible. Especially, I take this opportunity to extend my deepest gratitude to my advisors, Dr. Jordi Guitart, and Dr. Jordi Torres. The first one guiding the day to day work and keeping the low level issues, and the second one giving a broad view and opening new fronts; I think we have done a good job. They also have been responsible for creating a great working environment for me where I could start doing research: a research group with a great team, great ideas, and a fascinating subject to work in.

I would like to especially thank Ferran Julià who introduced me in this research group and made this dissertation possible. I want to thank everybody from my research group. The people with whom I have been sharing this PhD experience: Oriol Fitó, Mario Macías, Gemma Reig. People who used to work with us but that is still very close to us: Ramón Nou; and the current seniors of the group: David Carrera, Yolanda Becerra, and Vicenç Beltran; and last but not least, the fresh Dr. Javier Alonso, whose help has been really valuable.

Dealing with projects at BSC have been a pain but I have to admit, that these projects have been a great foundation and I would not be writing these lines without working on this center. In addition, it has given me the opportunity to work closely with many good people: Jorge Ejarque, Marc de Palol, Rosa Badia, Raül Sirvent, Enric Tejedor and Alex Vaqué.

During these last months, I had the chance to live the American-style research at Rutgers University. Ricardo Bianchini, thanks for welcoming me as if I were one more. I would like to thank all the DARK people with whom I worked and lived those days: Kien Le, Wei Zheng, Rekha Bachwani, Luiz Ramos, Cheng Li, and Qingyuan Deng. I also want to thank and acknowledge Ivan Rodero, my Catalan link in Rutgers. It has been a great experience that I believe has deeply marked me.
I would like to close these acknowledgements breaking with the cliché of citing institutions in a cold manner. The first institution which gave me the chance to start this Ph.D was Barcelona Supercomputing Center. It gave me the chance to start developing projects which later became the foundation of this dissertation. What is better, they paid me for doing something I love to do: research.

I do not want to forget about the three institutions that have support me with grants: Universitat Politècnica de Catalunya for the FPU-UPC grant, it was short but intense; Generalitat de Catalunya has also participated with the FI grant 2009FI B 00249 during a year; and last but not least, the Ministerio de Educación for the FPU grant AP2008-02641 and the funding to stay in the United States. Getting each one of these grants has implied a big headache but I think all these bureaucracy made me stronger.

Finally, I would like to dedicate this dissertation to my family for their support and patience and especially to my father.
Contents

Abstract iii
Acknowledgements iv
Contents vi
List of figures x
List of tables xii

1 Introduction 3
  1.1 Provider evolution ............................... 3
  1.2 Virtualization renaissance ......................... 4
  1.3 Virtualized provider management challenges ....... 5
  1.4 Contributions of the dissertation ................. 7
    1.4.1 Decentralized virtualization management ... 7
    1.4.2 Increase provider utilization ................. 9
    1.4.3 Multifaceted provider management ............ 10
    1.4.4 Tackle load variability by means of federation ... 11
  1.5 Overview of the dissertation ..................... 12

2 Enhancing virtualization fabrics 13
  2.1 Introduction ....................................... 13
  2.2 Managing virtual machine life-cycle ............... 14
  2.3 Managing user data ................................ 15
  2.4 Supporting migration ............................... 16
    2.4.1 Distributed shared file system ............... 17
  2.5 Supporting checkpointing ............................ 17
    2.5.1 Virtual machine checkpoint .................... 18
CONTENTS

2.5.2 Checkpoint compression ........................................... 19
2.5.3 Checkpoint storage .................................................. 20
2.6 Experimental environment ............................................. 21
2.7 Evaluation .............................................................. 22
  2.7.1 Creation performance ............................................. 22
  2.7.2 Distributed shared file system performance .................... 25
  2.7.3 Migration performance ............................................ 26
  2.7.4 Checkpoint performance ......................................... 28
2.8 Related work .......................................................... 38
2.9 Conclusions ............................................................ 41

3 Managing a virtualized host ............................................ 43
  3.1 Introduction .......................................................... 43
  3.2 Resource management ............................................... 45
    3.2.1 Resource monitoring ........................................... 46
    3.2.2 Resource assignment ........................................... 47
  3.3 SLA enforcement ..................................................... 49
    3.3.1 SLA evaluation .................................................. 50
    3.3.2 SLA metric for resource-level guarantees .................... 51
    3.3.3 SLA description ............................................... 55
  3.4 Experimental environment ........................................... 57
  3.5 Evaluation .......................................................... 57
    3.5.1 Resource monitoring ......................................... 58
    3.5.2 Resource assignment ........................................... 59
    3.5.3 SLA enforcement ............................................... 61
    3.5.4 SLA metric ..................................................... 63
    3.5.5 Real workload .................................................. 67
  3.6 Related Work ........................................................ 69
  3.7 Conclusion .......................................................... 72

4 Managing a virtualized provider ..................................... 73
  4.1 Introduction ........................................................ 73
  4.2 Modeling a virtualized provider ................................... 75
    4.2.1 Unifying Units ............................................... 75
    4.2.2 Time reference ................................................ 76
    4.2.3 Dealing with non-running VMs ................................. 76
    4.2.4 Service Level Agreement terms ................................. 77
4.2.5 Dealing with resource heterogeneity .................. 78
4.3 Cost-benefit analysis ................................. 79
  4.3.1 Task requirements ................................. 79
  4.3.2 Service Level Agreement penalties ................. 79
  4.3.3 Infrastructure ................................. 81
  4.3.4 Energy consumption ............................. 82
  4.3.5 Virtualization overhead ......................... 82
4.4 Managing virtual machine placement ..................... 84
  4.4.1 Scheduling policy ......................... 85
  4.4.2 Management procedures ....................... 89
4.5 Experimental environment ................................ 91
  4.5.1 Simulator .................................. 91
  4.5.2 Scheduling policies ......................... 92
  4.5.3 Provider’s configuration ..................... 93
  4.5.4 Provider workload ......................... 94
  4.5.5 Service Level Agreements .................. 96
4.6 Evaluation ............................................. 97
  4.6.1 Scheduling algorithm scalability ............... 97
  4.6.2 Energy consumption vs. SLA fulfillment tradeoff ... 98
  4.6.3 Scheduling policy performance ................. 99
  4.6.4 Resource heterogeneity ..................... 102
  4.6.5 Fault tolerance .............................. 103
4.7 Related work ........................................... 104
4.8 Conclusions ............................................. 106

5 Managing federated virtualized providers .................. 107
  5.1 Introduction ......................................... 107
  5.2 Analyzing profitability in actual providers ........... 109
  5.3 Federation in Cloud providers ...................... 110
    5.3.1 Federated scheduler ......................... 111
  5.4 Characterizing a federated Cloud ..................... 113
    5.4.1 Allocation within the provider ............... 113
    5.4.2 Outsourcing to federated Clouds ............. 115
    5.4.3 Insourcing from federated Clouds ............ 116
    5.4.4 Insourcing and outsourcing in federated Clouds 117
  5.5 Implementing a federated provider .................... 117
    5.5.1 Capacity planning ......................... 117
5.5.2 Federated resource management .......................... 118
5.5.3 Extending local scheduling policy ......................... 118
5.5.4 Interconnecting providers ................................. 119
5.5.5 Resource availability ...................................... 119
5.5.6 Service Level Agreements ................................. 120
5.6 Experimental environment .................................... 120
5.7 Evaluation ..................................................... 121
5.7.1 Profitability analysis in a federated Cloud ............... 121
5.7.2 Extending local scheduling ............................... 124
5.7.3 Potential benefit in a federated Cloud ................. 125
5.8 Related work ................................................ 127
5.9 Conclusions .................................................. 129
6 Conclusions .................................................... 131
6.1 Future work .................................................. 133
A Architecture for virtualized providers ......................... 135
A.1 Related work ................................................ 136
A.2 Architecture ................................................ 137
A.3 Virtualization technology ................................... 138
A.3.1 Implementation issues .................................. 139
A.3.2 Libvirt .................................................... 141
A.3.3 Xen ....................................................... 141
A.4 Virtualization fabrics ...................................... 143
A.5 Virtual machine manager .................................. 144
A.5.1 Virtualization manager .................................. 144
A.5.2 Resource monitor ....................................... 145
A.5.3 External virtualization manager ....................... 145
A.6 Virtual machine scheduler ................................. 146
A.7 Summary ..................................................... 146

Bibliography ..................................................... 146
List of Figures

1.1 Logical architecture of a virtualized provider . . . . . . . . 12
2.1 Virtualization fabrics . . . . . . . . . . . . . . . . . . . . . 14
2.2 Data stage-in/out . . . . . . . . . . . . . . . . . . . . . . . . 16
2.3 Distributed shared file system . . . . . . . . . . . . . . . . 18
2.4 Checkpoint process . . . . . . . . . . . . . . . . . . . . . . . 19
2.5 Checkpoint storage architecture . . . . . . . . . . . . . . . 21
2.6 VM lifecycle . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
2.7 Network and disk usage on a mencoder remote execution . . . 26
2.8 Migrated mencoder execution . . . . . . . . . . . . . . . . . . 27
2.9 Migrated Tomcat execution . . . . . . . . . . . . . . . . . . . 29
2.10 Average time to checkpoint depending on VM memory size . . . 31
2.11 Average time to checkpoint depending on user disk size . . . 31
2.12 Time to make a series of checkpoints . . . . . . . . . . . . . . . . 32
2.13 Average time to checkpoint depending on compress/copy policy . . 33
2.14 Average time to checkpoint depending on infrastructure . . . 34
2.15 Average time to resume tasks depending on infrastructure . . . 36
2.16 Use case of our checkpoint infrastructure . . . . . . . . . . . . 37
3.1 Virtual machine manager . . . . . . . . . . . . . . . . . . . . 44
3.2 Resource management cycle . . . . . . . . . . . . . . . . . . . . . 45
3.3 Actual and predicted CPU usage of a VM over time . . . . . . . 46
3.4 Surplus resource distribution mechanism . . . . . . . . . . . . . 48
3.5 SLA evaluation cycle . . . . . . . . . . . . . . . . . . . . . . . 50
3.6 Gompertz Function for SLA Penalties . . . . . . . . . . . . . . 56
3.7 Calculated resources of a mencoder with requirement of 50% . . . 58
3.8 Calculated resources of a Tomcat with different requirements . . 59
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.9</td>
<td>Host running three tasks to demonstrate the resource assignment</td>
<td>61</td>
</tr>
<tr>
<td>3.10</td>
<td>Host running four tasks to demonstrate the SLA enforcement</td>
<td>62</td>
</tr>
<tr>
<td>3.11</td>
<td>CPU-Based SLA Metrics Comparison</td>
<td>65</td>
</tr>
<tr>
<td>4.1</td>
<td>Virtual machine scheduler</td>
<td>74</td>
</tr>
<tr>
<td>4.2</td>
<td>SLA fulfillment types</td>
<td>78</td>
</tr>
<tr>
<td>4.3</td>
<td>Simulator power consumption validation</td>
<td>92</td>
</tr>
<tr>
<td>4.4</td>
<td>SLA fulfillment according to the task nature</td>
<td>96</td>
</tr>
<tr>
<td>4.5</td>
<td>Algorithm scalability according to the number of hosts and VMs</td>
<td>97</td>
</tr>
<tr>
<td>4.6</td>
<td>Energy consumption using different turn on/off thresholds</td>
<td>98</td>
</tr>
<tr>
<td>4.7</td>
<td>Client satisfaction using different turn on/off thresholds</td>
<td>99</td>
</tr>
<tr>
<td>4.8</td>
<td>Power consumption comparative</td>
<td>101</td>
</tr>
<tr>
<td>4.9</td>
<td>Frequency histogram of SLA fulfillment</td>
<td>101</td>
</tr>
<tr>
<td>4.10</td>
<td>Energy consumption using different provider configurations</td>
<td>103</td>
</tr>
<tr>
<td>5.1</td>
<td>Provider workload over time</td>
<td>107</td>
</tr>
<tr>
<td>5.2</td>
<td>Provider federation</td>
<td>108</td>
</tr>
<tr>
<td>5.3</td>
<td>Interaction of multiple federated Cloud providers</td>
<td>112</td>
</tr>
<tr>
<td>5.4</td>
<td>Relation between utilization and capacity in a single provider</td>
<td>122</td>
</tr>
<tr>
<td>5.5</td>
<td>Relation between utilization and capacity using outsourcing</td>
<td>122</td>
</tr>
<tr>
<td>5.6</td>
<td>Relation between utilization and capacity when introducing insourcing</td>
<td>123</td>
</tr>
<tr>
<td>5.7</td>
<td>Relation between utilization and capacity when introducing insourcing</td>
<td>123</td>
</tr>
<tr>
<td>5.8</td>
<td>Relation between utilization and capacity when introducing insourcing and outsourcing</td>
<td>124</td>
</tr>
<tr>
<td>5.9</td>
<td>Service provider’s workload during a week</td>
<td>126</td>
</tr>
<tr>
<td>5.10</td>
<td>Comparison of provider’s profit with different capacities</td>
<td>127</td>
</tr>
<tr>
<td>A.1</td>
<td>Virtualized provider architecture</td>
<td>137</td>
</tr>
<tr>
<td>A.2</td>
<td>Paravirtualization difference to full virtualization</td>
<td>139</td>
</tr>
</tbody>
</table>
List of Tables

2.1 Used tasks for evaluation ........................................ 22
2.2 Testbed machines features ....................................... 22
2.3 Average time in seconds to create a VM using different ap-
proaches ............................................................. 23
2.4 Average time in seconds to copy a file ......................... 25
2.5 Average time in seconds to create a VM locally and remotely 25
2.6 Average time in seconds to run *mencoder* .................... 26
2.7 Average time in seconds to run a *memcoder* with a migration 28
2.8 Size of VM disks .................................................. 29
2.9 Average time in seconds to upload depending on protocol . 30
2.10 Average task execution time in seconds ....................... 35

3.1 Testbed machines features ....................................... 57
3.2 Used tasks for evaluation ........................................ 57
3.3 Tasks description for the resource assignment experiment .... 60
3.4 Tasks description for the SLA enforcement experiment ...... 63
3.5 Duration of SLA violations (proof-of-concept workload) ... 67
3.6 Number of SLA Violations and Profit (Real Workload) ...... 69

4.1 Node features according to its type ............................ 94
4.2 Workload details .................................................. 95
4.3 Scheduling results of policies ................................... 100
4.4 Scheduling results of policies with heterogeneity ............ 102
4.5 Scheduling results of policies with a faulty environment ... 103

5.1 Scheduling results of policies introducing outsourcing ...... 124
5.2 Service provider’s profit in Euros ............................... 127

1
Chapter 1

Introduction

1.1 Provider evolution

With the emergence of the Internet, companies began providing their services through this network. These companies are what we refer to as service providers or simply providers. To offer these services, providers require computing resources to host them. Notice that some companies host their services in third-party resources and thus, hosting other services can be also seen as a service. In this dissertation, we refer to providers to mean companies that manage their own computing resources.

The amount of required resources to provide a service depends on the number of users and it can range from just a single machine to thousands or even millions. These computing resources must be hosted in a place where they have network access, power supply, and cooling. When the amount of computing resources is large enough, they are hosted in datacenters, i.e. are facilities dedicated to house computer systems and other components required to operate them. In fact, some large providers like Microsoft or Google require multiple datacenters to host their services.

According to different sources, the providers’ resources are utilized less than 30% [1, 2]. This underutilization implies providers need lots of resources and energy to provide their services. To be economically profitable, providers tend to share their local resources among concurrent applications owned by multiple users. This implies the cohabitation of applications with very different behaviors and requirements and it can imply security issues. To overcome this problem, they started using virtualization.
Chapter 1. Introduction

1.2 Virtualization renaissance

In the last decade, providers started using Virtual Machines (VMs) in their datacenters to pack users and their applications [3]. This was a good way to consolidate multiple users in the same physical machine while isolating them from each other. Virtualization helps to encapsulate HPC jobs [4] or Web-based applications in VMs [5] and see them as single entities, which can be managed in an easier and more efficient way. This consolidation helps to reduce the amount of required resources and the energy consumed.

As long as virtualization has been introduced in datacenters, it has been opening new opportunities for resource management. Nowadays, it is not just used as a tool for consolidating underused nodes and save energy; it also provides new solutions to well-known challenges, such as resources heterogeneity management or power efficiency.

To use virtualization, providers need to setup the VM images and start them before using them [6]. This requirement brings a new advantage as they can provide customized environments according to the user’s requirements. Thus, users are not tied to a fixed environment.

As users and applications are packed as single entities, the way to manage them has changed. For example, this packing brings the capability to move VMs between physical machines with minimal effort [7]. This migration mechanism opens new ways for resource management as consolidation can also be effected at runtime.

Another example is checkpointing; this mechanism allows storing the status of an application and restoring it later. This was a very application-dependent mechanism and required introducing changes in the programs. Thanks to virtualization, there is no need to modify any application to take advantage of checkpointing [8]. With this mechanism, providers can increase their fault tolerance as they can quickly recover failed executions.

Going a step forward, in 2006, Amazon started offering a new service called Amazon Elastic Compute Cloud (EC2) [9]. This service allowed customers to rent and use VMs on demand for a period of time. It meant the start of the Cloud computing paradigm known as Infrastructure as a Service (IaaS), where the computing resources are offered in a pay-as-you-go manner [10].
1.3 Virtualized provider management challenges

Virtualized providers cannot be managed like traditional ones as they are now confronted with a set of requirements and capabilities that did not exist before. However, just introducing virtualization features in current solutions for virtualized providers has resulted in limited approaches or solutions that do not exploit all the virtualization advantages.

**Virtualization management scalability.** Providers must be able to deal efficiently with the new management operations in virtualization like creation of VMs, and new capabilities like migration and checkpointing.

Virtualization technologies only offer low-level primitives to support them. For instance, VM images must be setup manually by an operator and live migration requires a shared file system. This means that the use of virtualization in providers is not straightforward.

Supporting and managing these new functionalities imply a management overhead and this problem has already been tackled. However, the main disadvantage of typical solutions is that they rely on centralized solutions, which hampers provider scalability:

- Creation is commonly implemented using a centralized repository which contains multiple disk images for each user requirement [11].
- Migration requires VM disk images to be available in every physical machine. This is typically supported with dedicated servers that host these disks [7].
- Some checkpointing infrastructures do not take advantage of virtualization encapsulation [12], and others do not manage the checkpoints and just rely on centralized repositories [8].
- Lately, providers must consider the operational overheads related to the use of virtualization and take advantage of their new capabilities.

**Provider underutilization.** Applications usually do not require all the resources of a single host, so by multiplexing them onto physical resources, providers could initially fight resource underutilization.

However, commercial providers assign resources statically [13] or just consider a re-assignment of un-used resources [14]. Since the applications’ resource requirements vary over time, assigning resources to VMs in a static way moves the problem of underutilized hosts to underutilized VMs, which in turn translates into underutilized resources.
Multiple works have proposed techniques to perform dynamic resource management [15] and try to improve the resource utilization. Some of these works consider resource requirements in the form of service-level agreements (SLAs) [16].

However, current resource-level SLAs specify minimum resource allocations (e.g., a minimum amount of CPU share). Thus, providers have limited resource management leeway, as they cannot take resources away from underutilized VMs.

**Multiple facets to consider.** In the same way a non-virtualized provider must decide which node will run an application, a virtualized one must must decide where to run a VM (i.e., VM placement). However, virtualized providers cannot be managed like traditional ones as virtualization brings another layer of abstraction which prevents conventional policies from performing efficiently or correctly. For example, as seen before, this technology brings new overheads, like VM creation, and new capabilities, like migration.

These new capabilities also open new ways to tackle classical problems like power consumption and SLA fulfillment. For example, there are proposals that try to reduce energy consumption while fulfilling the SLA [17], manage the virtualization overhead [18], or provide fault tolerance [19]. The provider must allocate the VMs evaluating multiple facets and it must exploit all the new capabilities introduced by virtualization.

Ultimately, this VM placement must minimize the costs associated with the execution of a VM in a provider to maximize the provider’s profit, while keeping in mind every aspect of scheduling. Unfortunately, none of the current proposals takes into account all the parameters related to the operation of a virtualized provider.

**Difficult resource provisioning.** Resource provisioning in this kind of providers is a challenge because of the high variability of load over time [20]. Providers can apply local policies and serve most of the requests owning only a restricted amount of resources but under-provisioning may cause customers to be rejected during peak hours. On the other hand, valley hours incur in under-utilization of the resources.

As virtualization and IaaS make the access to external resources easier, providers can work together by covenant and share resources to serve their loads in a federated way [21]. Furthermore, these technologies allow
providers with underused resources to offer them to other providers. Both techniques enable higher profits for the provider, when used adequately. Nevertheless, these techniques are new and providers cannot exploit all their advantages yet.

Summarizing, providers now must be able to manage their own virtualized resources in a different way than before. Additionally, virtualization has enabled new techniques that can be exploited to increase provider economical profitability.

1.4 Contributions of the dissertation

This dissertation identifies the new challenges introduced in a virtualized provider that offers IaaS, and designs solutions to manage its resources in the most cost-effective way by exploiting virtualization capabilities. This dissertation proposes a remodeling of a virtualized provider, focusing on the advantages and inconveniences that virtualization presents. To achieve this, we study and understand each of the levels that compose a virtualized provider, and design a global solution which is able to evaluate multiple facets related to the virtualized provider operation to improve its performance and profitability.

We divide this contribution into four parts, namely distribute virtualization management, increase provider utilization, manage a provider focusing on multiple facets, and tackle provider load variability by means of federation.

1.4.1 Decentralized virtualization management

Providers introduced virtualization in their already existing infrastructure and thus, they do not fully exploit all virtualization potential. Moreover, the way they currently deal with virtualization capabilities relies on centralized solutions which does not implies scalability problems.

For example, when a user asks for a new VM, the provider needs to move and manage its disk image. We propose a system where images are created on demand according to the user’s requirements, and in a distributed way where each node is in charge of its own VMs.

Another example of this scalability problem is the migration of VMs, which usually relies on centralized storage. To overcome this limitation, we
have developed and evaluated a distributed shared file system, which allows efficient access to the disk and allows live migration.

And the last example of a capability that has changed with virtualization is checkpointing. Thanks to virtualization, checkpointing is totally transparent to the application, but the overhead is much higher. We propose a system where we only store the changes performed in a VM and use a distributed and replicated store for checkpoints.

To provide more scalability to the provider and exploit virtualization capabilities in a distributed way, we propose a new \textit{virtualization fabric}. Using this new architecture, every node is almost autonomous when using virtualization advantages.

\textbf{Publications.} These works study virtualization primitives (e.g., creation, migration) and how they can be exploited by a virtualized provider. The publications related to a decentralization virtualization management (a dynamic and distributed mechanism for creating VMs, and a distributed infrastructure that supports live migration and checkpointing for VMs) are presented in:

\begin{itemize}
  
  
\end{itemize}
1.4.2 Increase provider utilization

Current resource management policies take SLAs into account. However, these SLAs are not intended for avoiding resource underutilization as they just state a minimum level of use (e.g., minimum amount of CPU). In this way, providers have limited leeway as they cannot take resources from underutilized VMs and increase the use of their resources. To overcome this limitation, we develop a new SLA metric which not only allows specifying a minimum requirement, but also supports better utilization of idle resources.

Using this metric and resource management techniques from non-virtualized environments [25], we propose a SLA-driven resource management cycle that is able to reduce resource underutilization and provides quality of service to the users. Finally, it can be used to reduce the amount of physical resources required to run the same workload.

Publications. The first two works represent our first proposal for a SLA-driven resource management infrastructure. These works were performed in collaboration with researchers who provided the semantic (inference-based) scheduling policy. The third publication addressed the management of the resources of under-utilized VMs.


1.4.3 Multifaceted provider management

As seen before, virtualization prevents conventional policies from performing efficiently or correctly. For example, this technology brings new overheads, like VM creation, and new capabilities, like migration. These new capabilities open new ways to tackle classical scheduling problems like power consumption and SLA fulfillment.

We propose a novel way for scheduling a virtualized provider, that mainly focuses on the allocation of VMs to the nodes according to multiple factors related to VM execution. To achieve this, the economic revenue obtained by executing a VM is taken into consideration, and different costs are assigned to the multiple factors of any VM operation, namely power consumption, incurred SLA violation penalties, fault tolerance, etc.

Publications. These works reflect the evolution of our main contribution. We started out presenting the trade-off between power and performance, and later considered all the factors related to the operation of a virtualized provider using an economical model:


1.4.4 Tackle load variability by means of federation

Virtualized providers can work together by covenant and share resources in a federated environment to tackle variability in their workloads. However, federating providers requires having a clear understanding of the potential of each load-migration decision.

We present a characterization of providers’ federation, which helps to choose the most convenient decision depending on the status of the provider. These include when to outsource to other providers, rent free resources to other providers (i.e. insourcing), or shutdown unused nodes to save power.

We take these decisions based on several parameters and demonstrate how a provider can use these equations to exploit federation. Finally, we evaluate the profitability of using these techniques in a realistic environment.

Publications. In this part, we discuss the federation of multiple providers and how they can share resources to increase their benefit. The first work is a research effort to discover and discuss how we exploit federation to manage a provider. The second one presents how to add outsourcing in the policies designed to manage local resources:


1.5 Overview of the dissertation

This dissertation is organized into multiple levels according to the logical architecture of a virtualized provider, which is shown in Figure 1.1. An implementation of this architecture, which is the basis for this dissertation, is presented in Appendix A.

![Logical architecture of a virtualized provider](image)

Figure 1.1: Logical architecture of a virtualized provider

Chapter 2: Enhancing virtualization fabrics. Providers have multiple physical resources to host services and these hosts support virtualization to run VMs. In this chapter, we introduce a distributed Fabric that allows the decentralized management of virtualization and helps to understand how virtualization works in these providers.

Chapter 3: Managing a virtualized host. The Virtual Machine Management manages the resources of the VMs that run on top of the virtualized hosts. In this chapter, we present how to manage these resources efficiently and increase provider utilization.

Chapter 4: Managing a virtualized provider. Providers schedule the VMs among the computing resources they own. These resources can be housed in the same location, but they can also be distributed among multiple datacenters owned by the same provider. In this chapter, we present the mechanisms to Schedule Virtual Machines in a provider and we present a multifaceted provider management policy.

Chapter 5: Managing federated virtualized providers. Multiple providers can be federated to share loads and increase their profitability. In this chapter, we analyze the advantages of Provider Federation and we tackle the provider load variability problem by means of federation.

Each chapter evaluates our proposals, discusses the related work, and states our intermediate conclusions. Finally, Chapter 6 gives an overall conclusion to the dissertation.
Chapter 2

Enhancing virtualization fabrics

2.1 Introduction

Providers started using virtualization capabilities to provide consolidation. Nevertheless, they firstly need to be able to manage this new layer and take it into account.

On the one hand, there is a set of basic requirements to be able to exploit virtualization. The most important one is the management the life-cycle of a virtual machine. This includes the creation of new virtual machines which is a process that requires the management of their disk images.

On the other hand, there is a set of other capabilities that can be exploited to do a better management of the provider and bring new capabilities. This is the case of migration and checkpointing.

Virtualization technologies, such as Xen or KVM, only offer low-level primitive to support them. For instance, VM images must be setup manually by an operator and live migration requires a shared file system. This means that the use of virtualization in providers is not straightforward.

The way to currently deal with these capabilities relies on centralized solutions which implies scalability problems. Moreover, they do not exploit all their potential as they were introduced thinking in traditional scenarios.

For instance, VM images must be setup [6] and live migration requires a shared file system [7]. This means that the use of virtualization in providers is not straightforward and needs to be managed.
Chapter 2. Enhancing virtualization fabrics

In this chapter, we identify the challenges to be solved and propose solutions to take the maximum benefit of basic virtualization capabilities. These solutions allow enhancing the basic features of virtualization and create a distributed virtualization fabric that makes them more efficient and their usage easier. These virtualization fabrics are built just on top of the raw computing resources as it is highlighted in Figure 2.1.

Figure 2.1: Logic architecture of a virtualized provider: virtualization fabrics

2.2 Managing virtual machine life-cycle

Virtualization offers the image of a dedicated and customized machine to each user, decoupling them from the underlying resource. With this approach, the virtual machine used by the user is just a piece of software. This reduces the management efforts in providers as it allows having a new VM just copying a previously created.

Virtual Machines are basically composed by some memory space from the host machine and disk space. This disk contains the guest Operating System (OS) data and space for the user data.

Providers usually have a set of disk images which contain a certain OS with software installed. Usually, these images are stored in specific servers and deployed in the working node in charge of running the VM. Nevertheless, this approach requires moving these images to the final location and it implies a big overhead [18]. Moreover, every user and every setup requires a specific image which requires a huge disk space in the provider.

To avoid these problems, we propose a VM creation system where the provider setups the VM according to the user requirements on demand [22]. The creation process requires the following steps: (1) downloading and in-
stalling the guest operating system using debootstrap [33], (2) copying extra software needed by the client in an image that will be automatically mounted in the VM, (3) creating home directories and swap space, (4) setting up the whole environment, (5) packing it in an image, and (6) starting the VM. Once the VM has completely started, the guest operating system is booted. After this, the additional software setup needed by the client needs to be instantiated (if applicable).

From this description, one can derive that this process has two bottlenecks: the network (for downloading the core of the guest system; around 100MB of data), and the disk (for copying extra software needed by the client and creating all the needed images, namely base system, software, home, and swap; nearly 1.6GB of data).

The network bottleneck has been reduced using a caching system per node that creates a default image of the guest system with no settings when it is downloaded for the first time. Then, this image is copied for each new VM created in that node. This almost eliminates the downloading time (base system is only downloaded once per node and can be reused for each new VM in that node), but contributes to the disk bottleneck.

The disk bottleneck has been reduced by adding a second caching system per node that periodically copies the default base system image and the images with the most commonly used software to a cache space. When a new VM is created, we just need to move these images (an i-node change) to the final location.

This dynamic creation mechanism is used to provide customized VMs in a fast way while saving storage space in the provider. For example, Alonso et al. [34] use this mechanism to dynamically create VMs that host Web applications enhancing the system availability.

### 2.3 Managing user data

Typically, tasks running on top of VMs need some input data to be executed, and when they finish, they generate some output. For example, a video encoder needs an input video file and it generates an encoded video. When submitting a task, the client has to upload the input data. Similarly, when the task finishes, the client must be able to access the output data. In addition, the provider has to deal with a virtual disk image for each user.
We propose a global data repository that stores the information of every client and takes into account the ownership of each file [23]. Notice that besides user data, this repository is also used to store VM images. Furthermore, this repository provides data stage-in and stage-out and can be centralized (e.g. FTP) or distributed (e.g. Google File System).

As shown in Figure 2.2, before submitting a task to the provider, the client uses these services for uploading the required input data to the repository. This uploading process can be performed using multiple protocols like FTP, SFTP, S3 or any storage system that Hadoop File System supports.

When we create a VM to execute a task, the data management service gets the required information from the repository, creates a disk image containing this information and mounts this image in the home space of the VM. When the task finishes, the system extracts the output information from the VM and uploads it to the data repository, where it can be retrieved by the client using the protocols commented above.

### 2.4 Supporting migration

One of the main advantages of virtualization is the easiness to move VM between nodes. Migration process implies creating a snapshot of the memory and the disk, and moving it between nodes. In addition, it is desirable that the VM remains always available to the clients while this is being migrated (i.e., live migration) [7]. This requires the disk to be available on both nodes, and dealing with large data images (order of gigabytes).

To avoid these issues, we propose a distributed shared file system in
which each node can access its own local disk and the disk of the other nodes [23]. This allows each node creating VMs and executing tasks efficiently using the local disk. Furthermore, tasks can be also migrated with minimum overhead, since it is not necessary to transfer the VM image, while they can also be accessible during the whole process. Moreover, migrated tasks can access remotely their required data without noticeable performance penalty.

2.4.1 Distributed shared file system

Works such as the ones proposed in Kallahalla et al. [35] and Sotomayor et al. [36] propose solutions that use dedicated NFS servers to provide the images to every node. However, this approach has some performance problems, since it always implies working with a remote disk. It would imply a big problem when applying the on demand VM creation described in Section 2.2. Moreover, this dedicated server becomes a huge bottleneck since every node running a VM will access to it.

The alternative solution is using only a local file system on every node. This solution exhibits a good performance when creating the VMs (it is done locally to the node) but implies transferring the whole VM image when the task is migrated. Taking into account that a minimal VM image has around 2 GB of data, supporting live migration is not possible.

We propose a system with a distributed data management system where every node generates and manages its own images. To achieve that, the provider have a NFS distributed among the nodes [23]. In other words, each node has a NFS server that can be accessed from all the others, as shown in Figure 2.3. Notice that, though we use NFS, other network file systems such as iSCSI or AFS might be also used. By using this approach, each node can work locally when creating the VMs and these images are locally and remotely available when needed (e.g. during task execution). This avoids explicit image transfers when migrating a VM.

2.5 Supporting checkpointing

Crash and omission failures are common in service providers: a disk can break down or a link can fail anytime. In addition, the probability of a node failure in a provider increases with the number of nodes. Apart from reducing the provider’s computation power and jeopardizing the fulfillment
of his contracts, this can also lead to waste computation time when the crash occurs before finishing the task execution. To avoid this issue, efficient checkpoint infrastructures are required, especially in virtualized environments where these infrastructures must deal with huge VM images [8].

We propose a smart checkpoint infrastructure for virtualized service providers [24]. It uses a union file system to differentiate read-only from read-write parts in the VM image. In this way, read-only parts can be checkpointed only once, while the rest of checkpoints must only save the modifications in read-write parts, thus reducing the time needed to make a checkpoint. The checkpoints are stored in a distributed file system with replication. This allows resuming a task execution faster after a node crash and increasing the fault tolerance of the system, since checkpoints are distributed and replicated in all the nodes of the provider.

### 2.5.1 Virtual machine checkpoint

Making a checkpoint of a task running within a VM can imply moving tens of gigabytes of data, as it must include all the information needed to resume the task execution in another node: the task context, the memory content, and the disks. However, usually just a small amount of data really changes with respect to the VM startup. Keeping this in mind, our checkpoint mechanism mounts the base system as a read-only file system and stores the user modifications in an extra-disk space called *delta disk*.

The distinction between read-only and read-write parts was initially proposed in [8], and has been also used in other environments. For example Linux Live CDs store the modifications in external devices since it is not possible to modify or add any information to the CD. Different implementations for providing this kind of file system exist. Our checkpoint mechanism uses Another Union File System (AUFS) [37].
To apply this idea in a virtualized service provider, we need two different disks: a read-only one containing the base system and the *delta disk*, granted with read-write permissions, which will contain only the user space. These two disks have to be merged to create a root file system before starting the VM. This process is done by the RAM disk as proposed in [38]. Finally, when the VM has been already booted, the user can work with the file system in a transparent way without taking care about its underlying structure.

Notice that this process reduces considerably the time needed to make the checkpoint. Reducing this time is especially important considering that a task should be stopped when doing a checkpoint. Otherwise, if we allow the task to perform changes while the checkpoint is being done, this could result in a checkpoint inconsistency, due to concurrent accesses.

Finally, once the checkpoint is ready, and while the VM is still stopped, it could be directly uploaded to the checkpoint storage. Nevertheless, as the checkpoint size can be large, the upload time can also be large, increasing the VM stop time. For this reason, our mechanism merges the read-only disk with the last changes (as it is shown in Figure 2.4), generates a new delta disk for storing future changes and resumes the VM execution immediately. Finally, it starts uploading the checkpoint to the distributed file system in parallel with the execution of the VM.

Moreover, the creation of the new delta disk that will contain the future changes is performed before the checkpointing process has started or when it has already finished. Therefore, this does not contribute to the time necessary to do a checkpoint.

### 2.5.2 Checkpoint compression

As the *delta disk* only contains the modifications that have been made from the last checkpoint and the rest of the image just contains zeros, it can be highly compressed. Considering this, we could replace the direct upload of the delta disk checkpoint to the distributed file system, with the compression
of the checkpoint, and then, the upload of the compressed version. However, this must only be carried out if it allows saving time.

If there are not many changes, the delta disk can be highly compressed in a fast way. Nevertheless, if this disk contains many changes, it would be faster to just upload it than compress it and then upload this compressed file to the distributed file system. Therefore, it is worth compressing the checkpoint while:

\[ t_{\text{compress}} + t_{\text{upload}_c} < t_{\text{upload}_u} \]  

In this formula, \( t_{\text{compress}} \) refers to the time needed to compress the checkpoint, \( t_{\text{upload}_c} \) refers to the time needed to upload the compressed checkpoint, and \( t_{\text{upload}_u} \) refers to the upload time for uncompressed checkpoints. Therefore, our mechanism checks the amount of data contained in the disk to decide whether it has to be compressed or not.

According to this, \( t_{\text{compress}} \) can be estimated since this delta compression is done by just removing the large amount zeros and depends on the amount of contained data and this speed to compress can be deduced using previous compression times. Furthermore, the upload time can be also estimated from the time to upload the whole disk and the disk usage.

Our system will just upload the checkpoint if it evaluates that this is faster than compressing and uploading it. Notice that the compression of the checkpoints is especially intended for applications with low disk consumption, since their delta disks will have fewer changes.

Finally, this compression also allows the system to reduce the size of the checkpoints that must be stored. To save space in case too many delta images have been stored, they can also be merged.

2.5.3 Checkpoint storage

Obviously, checkpoints cannot be stored in the node where the task is running, because if this node crashes, the checkpoint will not be accessible. According to this, checkpoints must be stored in a remote disk space. However, storing the checkpoints in a single remote node is not a good solution either. This node would become a bottleneck for the performance of the checkpoint infrastructure. In addition, it would be a single point of failure, resulting in a not fault-tolerant solution.
To overcome these issues, our mechanism uploads the checkpoints to a distributed file system which supports replication, namely the Hadoop Distributed File System (HDFS) [39]. This system splits the stored data in blocks that are distributed and replicated among the nodes of the provider.

![Checkpoint storage architecture](image)

Every node is part of the distributed file system and stores some of the blocks of the checkpoints (Figure 2.5). The checkpoint of the base system and the delta disks are replicated and distributed among the multiple nodes. Thus, if one node crashes, the checkpoint can be restored from the other nodes by merging the base image and the delta disks. In addition, as the checkpoint is replicated, it can be concurrently downloaded from several nodes. Notice that the base system only needs to be uploaded to HDFS once, while the delta disk and the memory are uploaded every checkpoint.

### 2.6 Experimental environment

To evaluate our proposals, we will use three applications with different topologies: a batch application with high CPU consumption, a batch application that uses a big amount of data stored in the disk, and an application server. The first one is a composition of different tasks of the NAS parallel benchmarks [40] (Task A). The second one is a video file encoder, mencoder [41] (Task B). The third application, RUBiS (Rice University Bidding System) [42], simulates a auction service like eBay, and has been deployed in the Tomcat [43] server (Task C). The features of these tasks are summarized in Table 2.1.

Our experimental testbed consists of two hosts for running VMs, Host A which is a 64-bit architecture with 4 Intel Xeon CPUs at 3.0GHz and 16Gb
of RAM memory, and Host B which is a 64-bit architecture with an Intel Xeon CPU at 2.6GHz and 16Gb of RAM memory. We also have Host C acting as storage server which is a 64-bit architecture with 2 Intel Pentium D at 3.2GHz with 2 GiB of RAM. This information together with the hard disk features of the hosts (extracted with hdparm) are presented on Table 2.2.

Virtual machines run on top of Host A and Host B, while Host C contains the storage services. Moreover, when using HDFS, all nodes act as data nodes. In addition, other machines are used as clients to stress the application servers. All machines are connected through a Gigabit Ethernet. The average times are calculated from multiple experiments and they have a confidence interval of 95%.

2.7 Evaluation

In this section we evaluate each one of the enhancements we propose to improve the basic virtualization features. First of all, we test the performance of creating VM dynamically on demand. Secondly, we evaluate how tasks behave when working with the proposed file system. Thirdly, we test the proposed disk architecture when migrating VMs among nodes of a provider. And finally, we evaluate the checkpointing architecture.

2.7.1 Creation performance

This experiment measures the average time needed to create a VM with a Debian Lenny in Host A and make the user able to submit tasks on it. These
Table 2.3: Average time in seconds to create a VM using different approaches

<table>
<thead>
<tr>
<th>Action</th>
<th>No</th>
<th>1 level</th>
<th>2 level</th>
<th>Central</th>
</tr>
</thead>
<tbody>
<tr>
<td>Download base system</td>
<td>88.1 ± 23.2</td>
<td>-</td>
<td>-</td>
<td>9.8 ± 0.6</td>
</tr>
<tr>
<td>Create base system image</td>
<td>68.4 ± 1.2</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Copy base system image</td>
<td>-</td>
<td>45.2 ± 0.6</td>
<td>2.3 ± 0.4</td>
<td>-</td>
</tr>
<tr>
<td>Copy software image</td>
<td>13.9 ± 0.6</td>
<td>13.9 ± 0.6</td>
<td>0.0</td>
<td>-</td>
</tr>
<tr>
<td>Create home &amp; swap</td>
<td>13.7 ± 0.5</td>
<td>13.7 ± 0.5</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>Load image</td>
<td>4.4 ± 0.2</td>
<td>4.4 ± 0.2</td>
<td>4.4 ± 0.2</td>
<td>4.4 ± 0.2</td>
</tr>
<tr>
<td>Total time for running</td>
<td>184.6 ± 25.7</td>
<td>77.2 ± 2.0</td>
<td>6.7 ± 0.6</td>
<td>14.2 ± 0.8</td>
</tr>
</tbody>
</table>

measurements are summarized in Table 2.3, which shows that thanks to the two caching levels the time for creating a new VM can be highly reduced.

If caching systems are not used (column ‘No’), the system must download the guest system packages (around 100MB of data), which takes 88 seconds, and create a default base system image by installing these packages, which takes 68.4 seconds. When using the first caching level (column ‘1 level’), this is done once per node and the image can be reused for creating each new VM in that node just by copying it and this copy takes 45 seconds. In both cases, the creation of the VM image requires also creating the software and the home & swap images, which takes 13.9 and 13.7 respectively.

The second caching level (‘2 level’) pre-copies all the images (base system, software image, home and swap), which allows creating a full VM image in only 2.3 seconds. Notice that, once the image is ready, it must be loaded, which takes around 4.4 seconds. Thus, the time needed to start a fully configured VM, using the two caching mechanisms, is less than 7 seconds.

To compare our solution against the typical approach in virtualized providers, where images are deployed in a dedicated repository, we have built a repository in Host C which uses FTP. As shown in column ‘Central’ of the table, this approach takes almost 10 seconds to download the image from the centralized repository and it takes 14.2 seconds to start the machine running. This experiment is the optimal case for this approach where the repository is just serving one creation. The creation of multiple VMs at the same time would increase linearly the time. For this reason, we can conclude that our proposal works a little better than a centralized solution but it has no penalties when creating more than one machine at the same time.

Nevertheless, the above VM creation time does not include the time needed by the guest operating system to boot and be available to the user.
This time can be appreciated in Figure 2.6, which shows the CPU usage of a VM that executes Task A during its whole lifetime (from the VM creation to the VM destruction), including also the CPU usage of the Xen Domain-0.

As shown in Figure 2.6, during phase A, the Xen Domain-0 creates the VM. This spends almost one CPU. During phase B, the guest operating system is booted (first peak in the CPU usage graph). At this point, the customer’s task is executed during phase C. Finally, during phase D, the Xen Domain-0 destroys the VM. Notice that the CPU consumption of the Xen Domain-0 is only noticeable during the creation and destruction of the VM. The results in this figure confirm that the creation of a VM less than 7 seconds and according to this while the full creation of the VM takes around 20 seconds. In addition, VM destruction takes 6 seconds. This numbers depend on the experimental environment but they give a good approach of the improvement of our approach regarding classical solutions.

We have demonstrated the benefit of using our distributed solution with two cache systems to reduce the VM creation time compared with the typical approaches. Notice that, since our system is intended for the execution of medium and long running tasks (with running times ranging from minutes to hours or even days), the overhead incurred by using virtualization (mainly in the creation and destruction of the VM) is not significant. Moreover, this system can be applied to other virtualization technologies such as VMWare and VirtualBox, and it has been already tested with KVM.
2.7.2 Distributed shared file system performance

Sharing the hard disk enables the migration of running application from one to another but working remotely implies a certain loss of performance. Nevertheless, the influence in the performance depends on the type of application. For example, an application server will probably use fewer disk than a video encoding application or than a simple file copy. In these experiments, we evaluate the performance of different types of applications using both local and remote disks.

The first experiment consists on stressing the disk by copying a file of 580 MB. Table 2.4 shows the time needed to execute this task in our two hosts when accessing the local disk and a remote disk using NFS. Again, using a local disk provides better performance for copying a file. It is also noticeable the influence of the disk speed on the obtained speedup.

<table>
<thead>
<tr>
<th></th>
<th>Local</th>
<th>Remote</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host A</td>
<td>4.1 ± 0.3</td>
<td>11.4 ± 1.2</td>
<td>2.78x</td>
</tr>
<tr>
<td>Host B</td>
<td>11.3 ± 0.8</td>
<td>21.0 ± 2.2</td>
<td>1.85x</td>
</tr>
</tbody>
</table>

Table 2.4: Average time in seconds to copy a file

The second experiment repeats the VM creation process performed in Section 2.7.1 but in this case, we evaluate the impact of working with a remote disk. As shown in Table 2.5, our experiments reveal that the whole creation of a VM needs 6.7s at Host A and 2.8s at Host B if the local disk is used, while it takes 13.5s at Host A and 5.3s at Host B when using a remote disk accessed via NFS. Notice that, our VM creation approach benefits from this because it uses the local file system for creating the VMs.

<table>
<thead>
<tr>
<th>Disk host</th>
<th>Local</th>
<th>Remote</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host A</td>
<td>6.7 ± 0.6</td>
<td>13.5 ± 0.8</td>
</tr>
<tr>
<td>Host B</td>
<td>5.3 ± 0.5</td>
<td>2.8 ± 0.3</td>
</tr>
</tbody>
</table>

Table 2.5: Average time in seconds to create a VM locally and remotely

The third experiment consists on running a mencoder application (Task B). Table 2.6 shows the time needed to execute this task in our two hosts when accessing the local disk and a remote disk using NFS. Notice that, though this task makes intensive disk usage, there is not any difference on executing remotely and locally. This occurs because the bottleneck of this application
is the CPU and accessing the remote disk using a fast Ethernet is enough to fulfill the disk requirements of this task.

<table>
<thead>
<tr>
<th></th>
<th>Local</th>
<th>Remote</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host A</td>
<td>583 ± 1</td>
<td>584 ± 1</td>
</tr>
<tr>
<td>Host B</td>
<td>640 ± 1</td>
<td>640 ± 1</td>
</tr>
</tbody>
</table>

Table 2.6: Average time in seconds to run mencoder

Figure 2.7 shows the CPU load and the disk accesses that produces the application (top graphic) and the overhead generated to access the remote data (bottom graphic). We can see how writes are cached and sent periodically to the server while reads are being made during all the execution.

Figure 2.7: Network and disk usage on a mencoder remote execution

Our approach uses the local disk on each node for task execution, which has demonstrated to be the fastest solution for all the types of applications. In addition, the only tasks which have been migrated access to data remotely. Nevertheless, this does not involve a performance problem, since we have shown that most applications (e.g. memcoder) exhibit similar performance when using local or remote disks. In fact, using remote disks will only have influence with applications with an extensive disk usage (e.g. file copy).

2.7.3 Migration performance

Sharing disks is needed to provide live migration of virtual machines between nodes. Once the loss of performance due to the remote disk has been
proved to be negligible with our test applications, this section will prove the migration of an application during its execution.

**Overhead of task migration**

First experiment measures the overhead introduced on the run time of a task which is migrated during its execution regarding the local execution. Figure 2.8 shows the execution of a *mencoder* that is migrated from *Host A* to *Host B* when it has been executed 300 seconds. The top graphic shows the CPU usage in both hosts of the task and the Xen Domain-0. The bottom graphic shows the network traffic of *Host A* and *Host B*. Notice that this graphic has been scaled to appreciate the transfer of data during the remote execution of the task, and not only the transfer during the migration.

![Figure 2.8: Migrated memcoder execution](image)

In this figure, we can see that when the task is running locally, there is no network traffic. Nevertheless, when the VM is migrated, a big amount of data (around 1 GB), including the whole memory of the VM and a snapshot of the system, is transferred to the other host. In this phase, the CPU usage of the Xen Domain-0 is noticeable on both hosts, since they are migrating the VM. Finally, when the task is running remotely, this can be noticed in the network traffic.
Table 2.7: Average time in seconds to run a *memcorder* with a migration

<table>
<thead>
<tr>
<th></th>
<th>run time</th>
<th>expected time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host A to B</td>
<td>611 ± 1</td>
<td>610.67</td>
</tr>
<tr>
<td>Host B to A</td>
<td>612 ± 1</td>
<td>610.25</td>
</tr>
</tbody>
</table>

This experiment also provides information about the total execution time of a task which has been migrated during its execution. This is important for measuring the migration overhead. Table 2.7 shows the execution time of *memcorder* if it is migrated at 300s from *Host A* to *B* and vice versa. The table also shows the expected execution time in these two situations. Notice that the overhead introduced when migrating an application is negligible.

**Accessibility of a migrated task**

Another issue is the accessibility of a task when it is being migrated. This is especially important in tasks such as a web server that provide clients’ requests. Next experiment evaluates the accessibility of such a task when it is being migrated. Figure 2.9 displays the execution of a Tomcat server stressed by several clients (*Task C*), showing its CPU usage and the Domain-0 CPU usage in both *Host A* (middle graphic) and *Host B* (bottom graphic). As expected, the Domain-0 CPU usage is only noticeable when the application is being migrated.

In addition, the top graphic of Figure 2.9 shows the reply rate of the server (i.e., the throughput), demonstrating that it is always available to the clients that try to access the web page. Notice that the reply rate is maintained during the whole execution of the server including the migration phase, obtaining a 100% uptime in the service.

**2.7.4 Checkpoint performance**

This section evaluates the performance of the presented checkpoint infrastructure and compares it with other alternatives. The service provider used in the evaluation consists of nodes: *Host A*, *Host B*, and *Host C*.

The size of the different disks needed by a VM supporting checkpoint are shown in Table 2.8. The size of the memory image depends on its size and the size of the user disk image depends on the user requirements and the usage of this image. Furthermore, a system that does not support the delta disks does not require the ramdisk, and the user disk only acts as a place
Figure 2.9: Migrated Tomcat execution

<table>
<thead>
<tr>
<th>Size (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuration file</td>
</tr>
<tr>
<td>Ramdisk</td>
</tr>
<tr>
<td>Kernel</td>
</tr>
<tr>
<td>Base system</td>
</tr>
<tr>
<td>Memory (m)</td>
</tr>
<tr>
<td>User disk (δ)</td>
</tr>
</tbody>
</table>

Table 2.8: Size of VM disks

to store user files: it does not save any changes of the system. Changes are directly stored in the base system image.

The task used in this experimentation is Task A which can be executed with a memory size of 256 MB. The execution of this application in Host B takes 3221 seconds.

Checkpoint storage protocol

Table 2.9 shows the time to upload the checkpoint of a VM using different storage protocols. This checkpoint, which has a size of 1496 MB, is composed of the base OS image, the delta disk, the memory and VM state, the
configuration file, the kernel, and the ramdisk of the VM.

<table>
<thead>
<tr>
<th>Storage</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDFS 3 replicas</td>
<td>53.2 ± 3.1</td>
</tr>
<tr>
<td>HDFS 2 replicas</td>
<td>31.3 ± 2.2</td>
</tr>
<tr>
<td>HDFS 1 replica</td>
<td>16.8 ± 1.2</td>
</tr>
<tr>
<td>FTP</td>
<td>18.6 ± 0.3</td>
</tr>
<tr>
<td>SFTP</td>
<td>33.3 ± 2.2</td>
</tr>
<tr>
<td>NFS</td>
<td>26.7 ± 1.1</td>
</tr>
</tbody>
</table>

Table 2.9: Average time in seconds to upload depending on protocol

As shown in this table, SFTP is slower than the other approaches due to the encryption of the channel, whereas other centralized solutions such as FTP and NFS are faster than the distributed ones (based on HDFS). However, centralized solutions imply a single point of failure and they are a bottleneck to store/recover multiple checkpoints concurrently. Furthermore, HDFS when using just one replica per file is similar to the centralized approaches, but it also stores parts of the file in the same node. Meanwhile, HDFS with a higher number of replicas has worse performance but distributes the checkpoints among multiple nodes, giving the system better fault tolerance and better performance for concurrent access.

Checkpoint performance

The time to make a checkpoint mainly depends on two different aspects: creating the checkpoint and uploading it to the checkpoint storage. This section evaluates the cost of making a checkpoint depending on the size of the memory of a VM, the size of user disk and its usage.

It is important to minimize the time to make a checkpoint as the VM must be stopped during this time. In these experiments, the VM is stopped during the ‘Save memory’, ‘Save user disk’, and ‘Restart VM’ stages.

Figure 2.10 shows the time needed to make a checkpoint and uploading it depending on the amount of memory of the VM. We have measured these times in a just created VM with a delta disk of 100 MB that only contains the changes required to boot the VM. The base system has been already uploaded to the checkpoint storage, thus only regular checkpoints must be uploaded. Tests have been executed in Host B.

As shown in this figure, the time needed to make a checkpoint increases
Figure 2.10: Average time to checkpoint depending on VM memory size

linearly with the size of the VM memory. This is due to the time required to save the memory and restart the VM execution after the checkpoint has been done. In addition, the size of the memory checkpoint is coupled with the memory size. Therefore, the time to upload the checkpoint of the memory is proportional with the memory size.

Figure 2.11 shows the time required to make a checkpoint on Host B of a VM with 128 MB of memory depending on the size of the user disk. As in the previous experiment, the base disk is already uploaded to the checkpoint storage, thus only delta checkpoints must be uploaded. The time to make a checkpoint increases with the size of the user disk, because of the time needed to recreate the delta image. However, it can be rapidly merged with the base disk and highly compressed because the disk usage is low. For this reason, the time to upload the user disk is just its compression time.
Previous results for the time needed to make a checkpoint assumed that the system disk had been already uploaded to the checkpoint storage. As commented before, this should be done only when the first checkpoint is performed. Figure 2.12 shows the required time to make a series of checkpoints (one every 100 seconds) of VMs with different memory sizes (in MB). As shown in the figure, the first checkpoint is more expensive than the rest, as all the disks must be uploaded to the checkpoint storage, while in the rest of checkpoints only the modifications on the user disk and the memory must be uploaded.

![Figure 2.12: Time to make a series of checkpoints](image)

If the checkpoint infrastructure would not use the delta disk approach, it would be required to upload all disks for every checkpoint. Hence, the time needed to perform a checkpoint would always be the time needed for the first checkpoint.

Finally, after testing our checkpoint with different parameters, we have seen the memory and user disk usage are key factors that determine the time that the VM is not available. In addition, our delta image technique makes the VM stop time stable if just small changes are performed in the file system. This is the most typical situation, where the user just writes some bytes in the disk space between two checkpoints.

**Checkpoint compression**

As commented, we propose compressing the user disk checkpoint to save space and reduce the upload time. To perform this operation we use “gzip” without any compression, which just removes the consecutive zeros of this image. For example, in an image of 800 MB which contains 554 MB of
data, making the compression take around 34 seconds in Host B while just copying the whole image is around 11s.

Nevertheless, compressing the checkpoint is not always the best solution, as we have discussed in this chapter. As the usage of the user disk increases, its compressibility decreases. This makes that from a given user disk usage, it is more efficient just to upload the checkpoint than to compress and upload it. We evaluate if it is worth doing it with Equation (2.1).

Figure 2.13 compares the time needed to make a checkpoint of a VM with 128 MB of memory depending on the usage of a user disk of 800 MB when using three different techniques: compress always, copy always, and our mixed technique. As shown in the figure, always compressing the user disk is a good approach when the usage of the disk is low, but it becomes more and more expensive as the usage grows. On the other side, always copying the checkpoint is expensive when the disk usage is low, but it is more efficient with high disk usage. Obviously, the mixed solution combines the benefits of the other two.

![Figure 2.13: Average time to checkpoint depending on compress/copy policy](image)

As shown in this figure, when the usage of the user disk is lower than 300 MB, our system compresses the disk checkpoint, since it estimates that this is faster than working with the original disk. With bigger usages, our system switches from compressing the checkpoint to just upload it without any compression. Using this mixed approach allows making the checkpoint of the user disk in the most efficient way independently of its usage.
In this section, we compare the performance in a real environment of our checkpoint infrastructure against other realistic approaches. We execute the afore-mentioned task within a VM with 256 MB of memory and 100 MB of user disk, which is enough to store the required files (around 10 MB).

First, we evaluate the performance of the different solutions regarding the time needed to make a checkpoint. The experiment consists of submitting the task and, after 300 seconds of execution, making a checkpoint every 600 seconds, until it finishes. Therefore, five checkpoints will be performed.

Compared alternatives include using AUFS to support checkpoints or storing always all the disks, and storing the checkpoints in a HDFS system with 3 replicas or a centralized storage server accessed using FTP. Figure 2.14 shows the time needed to make a checkpoint for each one of the five checkpoints when using the different checkpoint infrastructures.

When using All+FTP, it takes around 20 seconds to save all the disks of the VM. Once the checkpoint is created, it is uploaded to the FTP server used. Notice that there is no appreciable difference among the different checkpoints, since all of them must save and upload all the disks of the VM. This also occurs when using All+HDFS. The time to save the disks is basically the same as All+FTP, but the upload time has increased. This occurs because the checkpoints are being replicated and distributed across HDFS nodes and this requires more time to be accomplished than using...
FTP. However, this does not affect the time that the VM is stopped, which is the same as All+FTP. In addition, the time to recover a VM is reduced when using HDFS, as will be demonstrated in next experiments.

When using AUFS+FTP and AUFS+HDFS, the system disk is only uploaded with the first checkpoint. This can be clearly appreciated in the figure. Notice that the first checkpoint takes much longer than the rest, as it must upload the system disk. The next checkpoints only need to upload the user disk and the memory. The distinction between using FTP and HDFS described in the previous paragraph also applies in this case.

This experiment has demonstrated that the AUFS approach has better performance than the one that always checkpoint the whole disk. Moreover, FTP gives better performance, but it does not replicate information among multiple nodes. Thus, it becomes a single point of failure and a possible bottleneck if multiple nodes try to store or recover checkpoints concurrently.

Another interesting result of the previous experiment refers to the time needed to execute the task when using the different checkpoint infrastructures, which is summarized in Table 2.10. Notice that for all the solutions, the task execution time is basically increased with the time that the VM has been stopped making checkpoints.

Finally, we evaluate the performance of the different solutions when recovering the task execution in another node. In terms of checkpoint recovering, there is no significant difference between using AUFS or not, as the memory and all the disks must be recovered in any case. We only compare two alternatives: recovering checkpoints stored in a FTP server, and stored in a HDFS distributed file system.

Figure 2.15 shows the amount of time needed to resume some tasks execution when recovering a checkpoint stored in different storage systems. The infrastructure has to download all the checkpoints, merge all the delta
Chapter 2. Enhancing virtualization fabrics

Figure 2.15: Average time to resume tasks depending on infrastructure disks and finally, resume its execution. As this application just only uses around 10 MB of data, this process is very fast. The top part of this figure shows the time needed when only one task at a time wants to be restored. In this case, the three compared approaches (FTP, HDFS with 2 replicas, and HDFS with 3 replicas) behave similarly.

Nevertheless, when two tasks want to be recovered concurrently, differences arise. We have evaluated this by running two different tasks in two different VMs in a single node (i.e., Host B) that suddenly crashes. At this point, these are resumed in two other nodes: Task 1 will be resumed at Host A and Task 2 at Host C. As shown in the bottom part of Figure 2.15, FTP is considerably slower than HDFS when recovering the tasks, since the FTP server becomes a bottleneck for performance when accessed in parallel. Using HDFS allows resuming the tasks faster as their checkpoints can be recovered from multiples nodes at the same time. Notice that, which of the two tasks is slower depends on which recover started first. In this case, having more replicas of the checkpoint allows even faster recovery. This is denoted when comparing HDFS with 2 and 3 replicas.

Use case

Finally, we present an example of how the checkpoint infrastructure works and at the same time, we give a proof of concept of the presented checkpoint system. The use case consists of the execution of aforementioned task within a VM with 256 MB of memory and 100 MB of user disk space (Task A).
Evaluation

AUFS and HDFS with 3 replicas are used to manage checkpoints.

Figure 2.16 shows the CPU usage of the task, which is initially executing at Host B. Our system starts making checkpoints after 300 seconds of task arrival, and a checkpoint is performed every 600 seconds. At second 1800, Host B crashes and it is decided to resume this task at Host A. When the task execution is resumed at Host A, this node continues making checkpoints.

![CPU usage graph]

Figure 2.16: Use case of our checkpoint infrastructure

This figure shows how each checkpoint has a noticeable impact on the load of the VM hypervisor (i.e., Xen Domain-0). It also shows that making a checkpoint in one node also implies some load in the other nodes, as they have to store certain blocks of the checkpoint (because of using HDFS). However, after crashing, Host B does not do any CPU consumption at all. As the node has crashed 100 seconds after the last checkpoint, the execution performed in this time has been wasted. Nevertheless, the system can recover the previous 1700 seconds of execution.

The duration of this task if executed at Host B without making checkpoints is around 3220 seconds. However, the task duration in this experiment (which includes checkpoints and recovering the task at Host A) has been 3460 seconds. From this time, 60 seconds has been used to make the checkpoints, 100 have been spent between the last checkpoint and the crash of Host B, and 40 are needed to notice the failure of Host B and start
the recover at Host A. The additional 40 seconds corresponds to the lower computing capacity of Host A with respect to Host B.

This evaluation demonstrates that it is an effective way to make faster checkpoints with low interference on task execution and efficient task recovery after a node failure.

2.8 Related work

Multiple works have exploited virtualization capabilities to build their solutions. In this section, we discuss the related work for each proposal.

**Managing virtual machines lifecycle.** Virtualization has been used to facilitate system administration and provide the users with dedicated and customized virtual working environments, making more comfortable their work. There is a bunch of works that deal with the management of virtual machines (Appendix A). However, these proposals follow a simplistic vision where they just instantiate previously created VMs and they require to store one image for every scenario. Following this approach, the user just selects a given image and thus, the customization of the image is provided by the creator of the image.

There exists works that try to overcome this limitation by providing mechanisms to customize the execution environment according to the user requirements. For example, OpenNebula [44] has the ability to create VMs that are customized once they are already boot up. To do so, it uses a customization file which is read at boot time and the virtual environment is modified according to the specification file provided by the user.

Another example is VMShop which is a virtual management system that provides execution environments for Grid Computing. It uses VMPlant [45] and provides automated configuration to meet application needs. VMPlant also allows the creation of flexible VMs that can be efficiently deployed (by implementing a caching-based deployment) on Grid resources.

To prevent the overhead of moving images to each node, Keahey et al. [11] proposes a model for provisioning VM images. This model takes into account the overhead resulting from instantiating a remote virtual resource and introduces a method for efficiently manage virtual data transfer to reduce the overhead. Following this work, Sotomayor et al. [36] propose the scheduling of the creation of VMs. However, this proposal requires lots of
space to store the images and does not provide full customized environments.

To overcome this problem, they have proposed contextualization [6] where some of the VMs parameters are customized but this is still limited to a reduced number of scenarios. We propose a solution where the images are created on demand by each node. To the best of our knowledge, there is no other work proposing dynamic creation of VMs images.

**Managing user data.** Aforementioned approaches include some solution for managing data among VMs, since this is one of the main challenges in virtualized providers.

For example, Globus Virtual Workspaces [11] uses a shared file system among all the nodes to store the VM images. In addition, this work can store VM images and data in the repository for future usage in the same way our proposal does. Another example is SoftUDC [35], where data is shared between nodes by providing a smart storage virtualization service that allows any machine accessing any data in the pool.

Regarding commercial solutions, Amazon S3 [46] is the most used. This service can store VM images that can be used by Amazon EC2. The S3 space can also be mounted inside their VMs. This storage system is also supported by our system. In fact, we are able to use S3 space to store data and use the images stored in this service.

Nevertheless, previously described works (and others such as SODA [47]) delegate to the user the responsibility of putting this data inside the VM before the task starts executing. Our system simplifies this process by decoupling the user and the VM on which the task runs. The user just uploads the data in a data repository and the system makes it accessible to the VM.

**Supporting migration.** Nelson et al. [7] proposed a system that supported moving a VM from a node to another. This work presented the initial mechanism that allowed VM migration by transferring the status in the background. Nevertheless, it requires the disk image to be in a central repository with the performance bottleneck and the scalability problems it implies. A work that tries to solve this limitation is SoftUDC [35] which adds an efficient shared storage between nodes located in different locations. This project provides the capability of sharing resources of different organizations and solves problems such as sharing data between separated clusters.

To allow migration, some other works send the required data among the nodes. An example of this approach is VIOLIN [48], which uses diff files for
transferring disk images to reduce the amount of sent data.

Migration is a well known technique but it is unknown if it is being used in commercial systems as this information is not public. For this reason, we do not know whether they support it or even if they implement it.

**Supporting checkpoint.** Virtualization have opened new paths for failure management in providers. Pausing, resuming, and migrating VMs [12, 49] are powerful mechanisms to handle failures in such environments.

This is not a new approach, checkpoint and rollback technique [50] has been widely used in distributed systems. For example, it is widely used in high-performance computing systems and clusters [51, 52, 53, 54]. Different characterization approaches of these failures have been made. For instance, Fu and Xu predict the failure occurrences in HPC systems through the spatial and time correlation among past failure events [55]. Gokhale and Tivedi [56] forecast the software reliability representing the system architecture using Markov chains.

With the appearance of virtualization, it is possible to checkpoint any VM with no extra effort for the user. However, just saving the whole state implies also storing the complete disk that can be tens of gigabytes. To avoid this limitation, Vallee et al. [8] propose using a union file system to save time to store VM checkpoints. Thus, the checkpoint time overhead comes mainly from the time needed to save the state and the memory of a VM. This time is determined by the virtualization technology. It also introduces a remote storage for checkpoints and saves VM disk during the checkpoint phase. However, it does not provide any implementation and just presents a theoretical approach.

An improvement in checkpoint storage is presented by Ta-Shma et al. [57] where they introduce a Continuous Data Protection with live-migration-based checkpoint mechanism that intercepts migration data flow. Similarly, Warfield et al. [58] present a storage subsystem for Xen to be used in cluster Xen Virtual Machines. This solution makes coupled checkpoints of both memory and disk using a Copy-on-Write mechanism (CoW) to maintain the remote images. In contrast with our approach, they use a central repository to create the checkpoint images.

Finally, Badrinath et al. [19] present a working architecture that makes SLURM [59] able to support virtualization capabilities such as checkpointing. Nevertheless, they only focus on memory checkpoints, neglecting disk.
2.9 Conclusions

This chapter has presented a new fabric that supports virtualization capabilities and offers them in a transparent way. The advantage it brings is it avoids any centralization and decentralizes the virtualization management load among the nodes of the provider. This makes it suitable for providers with lots of resources that want to scale.

Firstly, we have described a solution for supporting virtualization which provides application-specific virtual environments. We have presented experiments that show that application-specific VMs can be created in lower time than previous techniques and in totally distributed way.

Secondly, we have presented a data management system for virtualized service providers that allow taking full advantage of the data-related virtualization capabilities. For example, we propose a distributed file system that supports a distributed migration mechanism where nodes do not require centralized solutions. Thanks to this mechanism, the scheduler can freely decide to migrate a task without worrying about possible performance degradation in VMs.

And thirdly, we have proposed a smart checkpoint infrastructure for virtualized providers. We use special VMs that allows storing changes and reduce the time required to checkpoint and, as a consequence, the interference on task execution. The checkpoints are compressed and stored in a distributed and replicated way. In the evaluation, we demonstrate we are able to reduce the time needed to make a checkpoint and recover checkpoints efficiently. Thanks to this mechanism, upper layers can provide fault tolerant mechanism that use checkpointing.

All these mechanisms have decentralized the virtualization management. Their advantages will be exploited by upper layers to deal with virtualization capabilities in an easier way. Furthermore, these results will be used to model the behavior of a virtualized provider in Chapter 4.
Chapter 3

Managing a virtualized host

3.1 Introduction

Virtualization allows the consolidation of applications, multiplexing them onto physical resources while supporting isolation from other applications sharing the same physical resource. In this way, providers can consolidate VMs and reduce the number of required resources to run the same workload. In addition, virtualization allows agile and fine-grain dynamic resource provisioning by providing a mechanism for carefully controlling how and when the resources are used.

In this chapter, we exploit the features of virtualization for facilitating resource management in hosts. In addition, we consolidate VMs in fewer hosts to increase utilization while we keep meeting the QoS agreed with the customers for each VM.

Commercial proposals usually perform an static assignment of the resources [13] or just consider a re-assignment of non-used resources [14]. Taking into account applications usually do a dynamic usage of the resources over time, assigning resources to VMs in a static way moves the problem of underutilized hosts to underutilized VMs which is translated in underutilized resources again.

For this reason, we require a dynamic resource management as it has been previously proposed [15]. Nevertheless, it relies on resource allocation and there has been other works that started taking into account other requirements specified in the Service Level Agreements (SLAs) [16]. However, these SLAs are not intended for avoiding resource underutilization as they just state a minimum (e.g. amount of CPU). In this way,
providers have a limited leeway and they cannot do an optimal usage of their resources.

We propose a SLA-driven resource management cycle that is able to reduce resource underutilization and provides quality of service to the users. To achieve that, we support fine-grain dynamic resource distribution among VMs based on SLAs. Following this approach, providers can benefit from easier resource management, usage monitoring and fine-grain SLA enforcement, since these tasks are implemented by means of adaptive behaviors. This makes service providers able to adapt to changes in the environment conditions without any additional effort for the system administrator.

It guarantees to each application enough resources to meet the agreed performance goals and it provides the applications with supplementary resources, since free resources are also distributed among them depending on their priority and resource demand. The system continuously monitors if the SLAs of the applications running in the provider are being fulfilled. If any SLA violation is detected, an adaptation process for requesting more resources to the provider is started.

To overcome the limitation regarding the traditional metrics, we propose a new SLA metric which not only allows specifying a minimum requirement but also supports better utilization of idle resources.

Following the virtualized provider architecture, the management of resources in a virtualized host is performed on top of the virtualization fabrics, presented in Chapter 2, as it can be seen in Figure 3.1.

![Figure 3.1: Logic architecture of a virtualized provider: virtual machine manager](image)
3.2 Resource management

Virtual Machine Manager layer is responsible of distributing the provider’s physical resources among the VMs [26, 60]. The goal is to maximize physical resources utilization, while fulfilling the SLAs.

Each node is managed by a Virtualization Manager (VtM) and it hosts multiple VMs. For each VM, the VtM guarantees that the minimum resource requirements are met during the whole VM lifetime. Surplus resources that are not allocated to any VM are dynamically redistributed among the others. In this way, the applications can be provided with better QoS and the provider’s resources are fully exploited.

To accomplish this, the global Scheduler provides two parameters for each VM, namely the minimum resource requirements and the initial priority of this VM, to the VtM. These resource requirements would be derived from the SLA of the VM. Although the derivation of the resource requirements can be complex, specially for service-level metrics (e.g. response time, availability), this is out of the scope of this dissertation. We just focus on the requirements derivation from resource-level metrics. Regarding the priority, it corresponds to the priority of the customer that owns the VM (e.g. Gold, Silver, etc.).

We propose an approach where the VtM monitors the resource usage of the VMs and assigns the resources to each one according to the SLAs. To do so, it includes a SLA evaluation stage as it can be seen in Figure 3.2 which is further described in Section 3.3.1.

![Figure 3.2: Resource management cycle](image-url)
3.2.1 Resource monitoring

The provider has a monitoring subsystem that allows easily consulting the amount of CPU allocated to a VM and the amount that it is really using. The monitoring subsystem gets this information periodically from the virtualization hypervisor and the Resource Fabrics layer using the Resource Monitor architecture presented in Section A.5.2. From this monitoring system, we estimate the amount of resources that this task really needs. Figure 3.3 shows an application that is running in a VM managed by the VtM with no other tasks being executed in this node.

![Figure 3.3: Actual and predicted CPU usage of a VM over time](image)

In the figure, “current” usage is the resource usage that the application is doing during the time. This usage is obtained from outside the VM for two reasons: the first is that doing this, it will charge to the customer the cost (in CPU and memory) of this monitoring, and the second one is because measures taken inside the VM are not real measures (the only way to get real measures is asking to the hypervisor).

This problem is manifested, for instance, when having only one real CPU, two VMs try to consume 100% of the CPU. In this case, we get two different measures of the CPU consumption depending on where we measure. Measuring from the outside, we would see each VM uses a 50%. However, measuring inside the VM we get values between 20% to 100%.

Next, the “estimated” corresponds to the value inferred from the SLA agreed with the user before the execution. Finally, “calculated” is the amount of resources this application will really need to be executed. This value is based on the estimation done initially and varies depending on the actual usage of the application.
The calculation of this estimation varies according to the real usage of the application. Algorithm 1 shows how we calculate this value once it has enough usage values. Until this point, it uses the customer estimation.

**Algorithm 1 Calculate resource needs for a VM**

```
1: history = A list containing resource usages
2: calculated = last calculated value
3: decreasing = 0
4: if last(history) > 1.1 \cdot calculated then \{High increase\}
   5: \hspace{1em} calculated = \max(\text{mean}(history, 5), calculated)
   6: \hspace{1em} decreasing = 0
7: else if last(history) > calculated then \{Normal increase\}
   8: \hspace{1em} calculated = last
   9: \hspace{1em} decreasing = 0
10: else if calculated > mean(history, 5) then \{Checking decrease phase\}
   11: \hspace{1em} decreasing = 0.5 + \frac{calculated}{\text{mean}(history, 5)}
12: end if
13: if decreasing > 200 then \{Decrease\}
14: \hspace{1em} aux = \max(\text{mean}(history, 5), \text{mean}(history, 60), \text{mean}(history))
15: \hspace{1em} if required > aux then
16: \hspace{2em} aux = \text{mean}(required, aux)
17: end if
18: \hspace{1em} calculated = \text{mean}(calculated, aux)
19: \hspace{1em} decreasing/ = 2
20: end if
21: return calculated
```

This algorithm has three phases; the high increase phase avoids assigning too many resources to an application in a small period of time. The normal increase phase provides resources to the application immediately. Finally, the decrease phase is brought up slowly to not subtract too many resources immediately and wait until it confirms this.

Once it detects an application does not need that resources, it uses three different values to calculate the estimation: last 5 values mean, 1 minute mean and half and the total mean to reduce the assignment slowly.

### 3.2.2 Resource assignment

Once we have calculated the resources the VMs actually requires, these are assigned to the VM. In case that the amount of resources is lower than the node capacity, the system will have surplus resources not assigned to any
task. For this reason, the surplus resources are redistributed among the VMs according to a dynamic priority. This priority initially corresponds to the priority set by the user and it is dynamically modified to apply for more resources (by increasing the VM priority) if the internal SLA is violated. The amount of resources assigned to a VM is calculated using the next formula:

\[ R_{\text{assigned}}(i) = R_{\text{calculated}}(i) + \frac{p_i}{\sum_{j=0}^{N} p_j} \cdot R_{\text{surplus}} \]

where \( p_i \) is the priority of client \( i \).

Every time that the usage requirements of a VM change or dynamic priorities are changed, the \( VtM \) recalculates the resource assignment of the VMs. Finally, it binds these resources to each VM (Figure 3.4).

The resource binding in a node is a key issue in the local resource management; low layers provide different mechanisms for assigning resources to a VM. For example, CPU management is straightforward by using the Xen Scheduler credit policy. This is a proportional fair share CPU scheduler which assigns each domain a weight and a cap. A VM with a weight of 512 will get twice as much as CPU as a domain with a weight of 256 and the cap specifies a maximum CPU usage for a VM. Tweaking these scheduling parameters, we can specify the CPU assigned to a VM.

However, there are limitations to manage VM memory dynamically. In Linux systems, the mapping of physical memory is done at boot time. Once the guest system has booted, if the amount of memory allocated to the VM is reduced, the guest system is adapted to this reduction automatically. However, when assigning to the VM more memory, Linux does not handle it and it is necessary to restart the guest OS to make all this memory available. To overcome this limitation, \( VtM \) creates all the VMs with the maximum amount of memory possible and then it reduces the amount of allocated memory to the value indicated by the Scheduler.
3.3 SLA enforcement

As we have discussed, virtualized providers use resource management policies to increase the utilization of the resources. However, providers must supply fine-grain Quality of Service guarantees to customers and thus, they have to manage the resources taking into account these constraints.

Typically, a provider agrees the QoS with its customers through a Service Level Agreement, which is a bilateral contract between both stakeholders, i.e., the customer and the provider (or between providers). The provider uses this information to estimate the resources the user needs to fulfill its SLA. Using this information, the provider can perform a resource management driven by the SLA terms. To achieve this, we propose an SLA enforcement cycle that predicts possible SLA violations before they happen and using this information, it redistributes the resources to fulfill the agreement.

SLAs do not only state the conditions and constraints of a given service, but also characterizes the agreed QoS between them using a set of metrics. Service providers naturally offer service-level metrics (e.g. availability, service execution deadline [61], response time, throughput) to their customers for specifying the QoS.

Using service-level metrics has advantages for both the customer and the provider. The former does not need to provide detailed information about the resource requirements of the service to be executed (probably the customer does not know this exactly), but only a high-level performance goal (e.g. a deadline for executing the service or a given response time threshold). The latter can freely decide the allocated resources to the service whereas it guarantees that the service meets the agreed performance goals.

Being able to dynamically allocate resources to the different services is especially important for Cloud providers. Considering that, aiming for profitability, they tend to share their resources among multiple concurrent services owned by different customers.

On the other hand, resource providers in the Cloud must offer resource-level metrics that can be used to provide fine-grain QoS guarantees. First, the QoS agreement can be naturally expressed using resource-level metrics (e.g. number of processors, frequency of processors, FLOPS, etc.), since raw resources are the traded good.

Second, having fine-grain metrics, which guarantee a given resource allocation during a time period, is especially important for service providers...
that outsource resources to resource providers, as we have stated before. For instance, try to figure out how a service provider could guarantee that a service will finish within a given deadline if he does not receive fine-grain guarantees on the FLOPS supplied at every instant by the resource provider where the service is running.

Ultimately, service providers can also benefit in some situations from supporting resource-level metrics. First, those that do not support the inference of resource requirements of a service to fulfill a given service-level metric, and for this reason, cannot offer service-level metrics in the SLA. Second, those dealing with customers (typically coming from the HPC domain) who prefer to stay with resource-level metrics that they have been using for a long time, instead of moving to service-level metrics.

### 3.3.1 SLA evaluation

We propose a cycle for evaluating the fulfillment of SLAs in virtualized providers [27]. Figure 3.5 shows the SLA enforcement cycle proposed for virtualized providers. This cycle is executed periodically and it starts when the SLA Evaluator gets the values from the resource monitoring system using the presented resource management capabilities (Resource Monitor). This component controls the measurement intervals for each metric in the SLA and ensures the refresh of the measures on correct time.

![SLA evaluation cycle](image_url)

Figure 3.5: SLA evaluation cycle
When these values arrive at the *SLA Evaluator*, it checks if any metric has updated its value. In this case, the *SLA Evaluator* recalculates the top-level metric defined in the SLA using the corresponding formula (*Recalculate SLA*) and compares the result with the agreed value specified in the guarantee terms of the SLA (*Evaluate SLA*). If the SLA is fulfilled, the *SLA Evaluator* waits until the next iteration, otherwise the SLA violation protocol asks the resource management subsystem to reschedule the VM resources using the mechanism presented in Section 3.2.2.

### 3.3.2 SLA metric for resource-level guarantees

Current Cloud providers do not support fine-grain resource-level QoS guarantees on their SLAs. In fact, most of them only support SLAs with very simple metrics based on resource availability [62, 13, 63, 14]. For instance, whereas Amazon EC2 [9] offers different instances according to their computing capacity (which is measured in ECUs, EC2 compute units), there is not any guarantee in the SLA that this computing capacity will be supplied during the whole execution of the instance, as Amazon’s SLAs only provide availability guarantees [13].

According to this, one could think on using Amazon’s ECUs to support fine-grain resource-level guarantees on Cloud SLAs, or porting traditional resource-level metrics from the Grid environment to Cloud providers. However, this must be carried out carefully, since it can prevent providers from obtaining the maximum profit of their resources if done naively.

Metrics related to computing capacity (i.e., CPU) are especially susceptible to naive usage. For instance, if the agreed metric in the SLA establishes that the number of processors allocated to a given service must be greater or equal to some value, the provider must maintain this assignment during the whole execution of the service. This happens even if that service is not using all the allocated capacity during some phases of its execution (i.e., the provider is forced to statically overprovision processors to the service to avoid SLA violations). Notice that the unused resources could be temporarily allocated to another service, improving in this way the utilization and profit of the provider.

**Overview.** We derive a resource-level metric for specifying fine-grain QoS guarantees regarding the computing capacity of a Cloud provider by extending the Amazon’s approach [28]. Our metric overcomes the limitations
of traditional CPU-related metrics. By taking into account the customer’s resource usage, it allows the provider to implement dynamic resource provisioning and allocate to the different services only the amount of CPU they need. In this way, better resource utilization in the provider can be achieved.

There are situations where the SLA evaluator mechanism detects that an SLA is being violated, and the provider will be penalized for this, but the violation is not provoked by the provider’s resource allocation. In general, this occurs when the provider uses a poorly designed resource-level metric and the customer’s service does not use all the resources it has allocated.

The solution is to design a solid resource-level metrics that support this. This metric makes the customer free to use the amount of resources he wants, without provoking any SLA violation. Notice that the proposed metric can be used in heterogeneous environments, as it is based on Amazon’s ECUs.

**Metric unification among heterogeneous machines.** Cloud providers typically present very diverse architectures: architectures, speeds, etc. For this reason, a good resource-level metric has to be platform-independent so it can be used in all these architectures. In this section, we describe our approach for unifying computing capacity metrics among machines with heterogeneous architectures.

Commonly, **megahertz** have been used to measure the computing capacity of a machine. However, this does not directly measure the computing power of a machine, since noticeable differences can be observed depending on the processor architecture. For instance, using this measure a Intel Pentium III with 500 MHz would be 20 times slower than a Intel Xeon 4-core with 2.6 GHz ($\frac{4 \times 2600}{500} = 20$). However, simple tests demonstrate that it can be up to 85 times slower.

To consider the heterogeneity of processor architectures, Amazon uses EC2 compute units (ECU) in its services [9]. An ECU is equivalent in CPU power to a 1.0-1.2 GHz 2007-era AMD Opteron or Intel Xeon processor. This serves as a unified measure for the computing power, though it is not easily portable among different architectures. In this work, we use ECUs to unify CPU-related SLA metrics among heterogeneous machines, and additionally we extend the Amazon’s approach to set up SLAs that provide fine-grain CPU guarantees based on ECUs during a time period.

In our proposal, the maximum computing capacity of a given machine is
measured using its maximum amount of CPU\(^1\) (\(maxCPU\)) and the \(ECUs\) associated to the processor installed in that machine in the following way: 
\[ \frac{maxCPU}{100} \cdot ECUs. \]

**Derivation of CPU-based SLA metric** This section describes how our resource-level metric for establishing computing power guarantees is derived, and at the same time, discusses the limitations of alternative metrics.

All the metrics discussed in this section intent to establish a guarantee on the computing performance (in terms of CPU) of a service over a period of time in a provider using the idea presented in the previous section. This guarantee is a fixed value for each SLA, represented by the \(SLA_i\) term in the formulas, which results from the negotiation between the customer and the provider. The customer is only required to specify the computing performance he requires (in terms of ECUs), which will be accepted by the provider if it is able to provide this performance.

The main difference among the metrics is how the amount of CPU for a service is defined. The more natural approach is specifying CPU performance as a function of the allocated CPU to a service (Equation 3.1). This metric specifies that the resources for a service \(i\) at every time period \(t\) has to be at least the agreed value in the SLA (\(SLA_i\)), which depends on the amount of CPU that the provider assigns to the service in that time period (\(assig_i(t)\)). This assignment can vary over time, and it is periodically obtained by means of the monitoring subsystem.

\[
\frac{assig_i(t)}{100} \cdot ECUs \geq SLA_i 
\]  
(3.1)

Note that using this metric forces the provider to statically allocate to each customer at least the amount of CPU agreed in the SLA (he can assign more if he wants), because otherwise, the SLA will be violated. Note that there is not any control on whether the customer uses its allocated CPU or not. This is a quite restrictive approach, especially when customers’ services have a variable CPU usage over time. In this case, the provider will suffer from low resource utilization when the services do not use all the CPU they have allocated, since unused resources cannot be allocated to other services running in the provider.

As we have commented before, the provider aims to dynamically allocate

\(^1\)All CPU-related measures are quantified using the typical Linux CPU usage metric (i.e., for a computer with 4 CPUs, the maximum amount of CPU will be 400%)
its resources to the services depending on their demand, to improve resource utilization (and then increase profit). This requires an SLA metric that considers the CPU usage of services. However, as the CPU usage depends on the client’s task behavior, it must be carefully used as a CPU guarantee because it can provoke undesired effects.

Equation 3.2 shows an SLA metric where the computing power for a service $i$ at every time period $t$ depends on the amount of CPU that the service uses in that time period ($\text{used}_i(t)$). This metric assumes a provider able to predict the CPU usage of a given service during the next time period from previous data.

\[
\frac{\text{used}_i(t)}{100} \cdot \text{ECUs} \geq \text{SLA}_i
\]

This is achieved with reasonable accuracy using techniques such as Exponential Weighted Moving Average (EWMA). The provider will assign CPU to the service according to its CPU usage prediction. It allows the provider to dynamically allocate the CPU among the services whereas it assigns each service at least with the CPU required to fulfill the SLA.

When using this metric, an SLA can be violated in two situations. First, when the provider assigns to the service an amount of CPU that is not enough to fulfill the SLA. This is a real violation, the provider is responsible for it, and must pay the corresponding penalty.

Second, when the provider assigns to the service an amount of CPU that should be enough to fulfill the SLA, but the service does not use all the assigned CPU. This is what we have defined as fake violation, since the provider is not causing it, and for this reason, he should not pay any penalty. However, Equation 3.2, as currently defined, provokes this to be considered as a real violation, thus penalizing the provider for it.

Of course, the service should be able to use only a part of the CPU it has assigned without incurring on SLA violations. To allow this, we introduce our metric, in which we introduce a factor that represents the percentage of CPU used by the service with respect to its total amount of allocated CPU. As shown in Equation 3.3, when the service is using all the assigned resources, Equation 3.1 applies, so an SLA violation will only arise when the assigned resources are not enough. When the service is not using all the allocated resources, the SLA is considered to be fulfilled, since the provider is not responsible that the service does not exploit its allocated resources.
\[
\frac{\text{assig}_i(t)}{100} \cdot \text{ECUs} \geq \text{SLA}_i \cdot \left\lfloor \frac{\text{used}_i(t)}{\text{assig}_i(t)} \right\rfloor
\] (3.3)

However, some services, even being CPU-intensive, do not use the 100% of their assigned CPU during their whole execution. For these services, Equation 3.3 does not work. To overcome this limitation, we introduce an \( \alpha \) factor as shown in Equation 3.4. This factor acts as a threshold to choose when the service is considered to be using all its assigned resources. For instance, if we consider that a service is using all its allocated resources when it reaches a 90% utilization, \( \alpha \) should be set to 0.1.

\[
\frac{\text{assig}_i(t)}{100} \cdot \text{ECUs} \geq \text{SLA}_i \cdot \left\lfloor \frac{\text{used}_i(t)}{\text{assig}_i(t)} + \alpha \right\rfloor
\] (3.4)

Operating on Equation 3.4, we obtain the version of our metric ready to be used in an SLA:

\[
\frac{\text{assig}_i(t)}{100} \cdot \text{ECUs} \geq \text{SLA}_i \cdot \frac{\text{used}_i(t)}{\text{assig}_i(t)} + \alpha \]
(3.5)

The equation in this form can have an undefined value when the denominator is zero, which happens when the service is not considered to use all its allocated resources. We avoid this by defining \( \text{SLA}'_i \) as \( \frac{1}{\text{SLA}_i} \) and operating the equation. Notice that, when using this metric, the value specified in the SLA is \( \text{SLA}'_i \) instead of \( \text{SLA}_i \). The final version of our metric is:

\[
\frac{\text{used}_i(t)}{\text{assig}_i(t)} + \alpha \leq \frac{1}{\text{SLA}_i} = \text{SLA}'_i
\] (3.6)

### 3.3.3 SLA description

The SLA framework allows assigning its own SLA to each service by using a XML description that combines both WS-Agreement [64] and WSLA [65] specifications. Using these specifications, we can accurately define the metrics derived in the previous sections. In addition, we use some features of the above-mentioned SLA specifications to define the window size (i.e., 10 measures) and the interval between two measures (i.e., 2 seconds).

Each SLA \( S_i \) specifies the revenue that the customer will pay if the SLA is fulfilled \( (\text{Rev}(S_i)) \), and the penalty that the provider will pay other-
wise \( (Pen(S_i)) \). According to this, the provider’s profit for running a given application \( (Prof(S_i)) \) results from the revenue paid by the customer minus the penalties that the provider has to pay due to SLA violations, i.e., \( Prof(S_i) = Rev(S_i) - Pen(S_i) \).

Each penalty \( Pen(S_i) \) is calculated as a percentage of the revenue obtained when fulfilling the corresponding SLA in the following way: \( Pen(S_i) = Rev(S_i) \cdot \frac{Gom(\sigma(S_i))}{100} \). This percentage is calculated using a Gompertz function, which is a kind of sigmoid function. Its basic form is \( y(t) = a e^{b e^{c t}} \), where \( a \) is the upper asymptote, \( c \) is the growth rate, and \( b, c \) are negative.

![Figure 3.6: Gompertz Function for SLA Penalties](image)

For our purposes, we have adapted this function as shown in Figure 3.6, which displays the penalty percentage depending on a \( \sigma(S_i) \) function that represents the SLA violation ratio. In particular, as shown in Equation 3.7, when this ratio is zero or less, the penalty percentage is 0. When this ratio is one, the percentage tends to \( MAX_P\% \) of the price that the client pays for SLA \( S_i \). Notice that this upper limit (\( MAX_P\% \) of the price) can be agreed with the client during the SLA negotiation process.

\[
Gom(\sigma(S_i)) = \begin{cases} 
0 & \text{if } \sigma(S_i) \leq 0 \\
MAX_P \cdot e^{-e^{-\sigma(S_i)+1}} & \text{otherwise}
\end{cases} \tag{3.7}
\]

As shown in Equation 3.8, \( \sigma(S_i) \) function depends on the number of violations occurred \( (V_i) \) and it is parameterized with two thresholds, \( \lambda_{\text{min}} \) and \( \lambda_{\text{max}} \). These thresholds indicate the minimum number of SLA violations to start paying penalties and the maximum number of SLA violations that could occur during the execution of the application, respectively.

\[
\sigma(S_i) = \frac{V_i}{\lambda_{\text{max}} - \lambda_{\text{min}}} - \frac{\lambda_{\text{min}}}{\lambda_{\text{max}} - \lambda_{\text{min}}} \tag{3.8}
\]
3.4 Experimental environment

The experiments to test our proposal to manage a virtualized host will be run on top of the hosts described in Table 3.1. The tasks executed to evaluate the performance of the proposal are summarized in Table 3.2.

<table>
<thead>
<tr>
<th>Host</th>
<th>CPU</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>4@3.0 GHz</td>
<td>16 GB</td>
</tr>
<tr>
<td>B</td>
<td>4@2.6 GHz</td>
<td>16 GB</td>
</tr>
<tr>
<td>D</td>
<td>8@3.8 GHz</td>
<td>16 GB</td>
</tr>
</tbody>
</table>

Table 3.1: Testbed machines features

<table>
<thead>
<tr>
<th>Task</th>
<th>Description</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NPB</td>
<td>CPU intensive</td>
</tr>
<tr>
<td>B</td>
<td>mencoder</td>
<td>CPU and Disk intensive</td>
</tr>
<tr>
<td>C</td>
<td>Tomcat</td>
<td>Application server</td>
</tr>
</tbody>
</table>

Table 3.2: Used tasks for evaluation

In addition to the local experiments, we execute a large workload but the size of our testbed is not enough to execute it. For this reason, we have built a simulated provider.

This simulator mimics the behavior of a virtualized provider and considers the times to create VMs and start running tasks. It reads the workload and starts tasks when the time to do it comes. When the task is started, the XML describing the task SLA is sent to the SLA framework. At this point, the simulator interacts with the actual SLA framework as it was the actual system providing the information of the system status periodically.

3.5 Evaluation

This section evaluates the capabilities of the SLA-driven resource management proposed for virtualized service providers. The first experiment tests the monitoring of the resources and its capability to estimate the actual resource requirement of applications that run on top. In the second experiment, we demonstrate how our proposal distributes the resources among the VMs running in a host according to their resource usage.

In the third experiment, we set up the SLA framework in the implementation of the virtualized provider already presented in Chapter A. Using
this environment, we evaluate the different metrics using real executions with a small proof-of-concept workload composed of three tasks to show in detail the consequences of using each metric. In the final experiment, we use a simulator that mimics the operation of the resource provider (including the same SLA framework) and we evaluate the whole system with a real workload in a large scale environment.

### 3.5.1 Resource monitoring

The key issue when allocating resources to a task inside a VM is estimating the resources that it really needs. This is mainly done using the calculated resource usage, and the redistribution of the surplus resources.

The calculation of the resources that a VM needs is tested by executing a single task in a machine and obtaining the amount of resources that it actually uses. This calculation is straightforward if the application consumes more or less the same amount of resources during its execution. This is the case of `mencoder` (type Task B) and can be seen in Figure 3.7. This figure shows how the user initially estimates that the application needs 50% of the CPU but the VtM detects it actually requires one CPU.

![Figure 3.7: Calculated resources of a mencoder with requirement of 50%](image)

The algorithm for estimating the resources a task needs becomes totally meaningful if the executed task has a variable resource usage like an Application Server (type Task C). This application is stressed with a variable load during the day as it is shown in Figure 3.8, which shows the performance of a Tomcat during an hour.

![Figure 3.8: Performance of a Tomcat during an hour](image)
requirement and leaves all the responsibility to the VtM by requiring no resources, the estimation is very close to the actual usage. In the middle one, the user specifies that his application needs one CPU and the VtM estimates that this job needs most of the time, more or less a 100% of the CPU and if the usage becomes more than this, it assigns this extra resources during the needed period. Finally, in the last one, where the user specifies that it needs two CPUs, it reduces the estimation in comparison with the one specified by the user.

This approach tries to give as much resources as the application needs and reduces this slowly to prevent a possible increase in its needs. According to this experiment, we can observe the system uses 200 seconds to detect a decrease in the resource needs. Notice that this time depends on the parameters of the algorithm and it can be tweaked. Finally, it uses various means values to detect increases or decreases, but it always tends to satisfy the user requirements.

### 3.5.2 Resource assignment

Once we have seen how the VtM estimates the resources that a task needs to be executed, we show how resource assignment works. This experiment demonstrates how resources are dynamically reallocated among applications
and how the system is able to detect and resolve an SLA violation from one of the applications and reallocate resources. The experiment consists of running three tasks which request the same amount of CPU (type Task A) but have different SLAs. Table 3.3 describes them: requested CPU (i.e., the value provided by the upper levels), and the agreed CPU metric in the SLA. For simplicity, all the measures and figures are referred only to CPU, but analogous results could be obtained with memory.

<table>
<thead>
<tr>
<th>Task</th>
<th>Requested CPU</th>
<th>Agreed CPU in SLA</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>100</td>
<td>190</td>
</tr>
<tr>
<td>3</td>
<td>100</td>
<td>100</td>
</tr>
</tbody>
</table>

Table 3.3: Tasks description for the resource assignment experiment

Figure 3.9 displays the execution of the three tasks. The first three plots show the allocated CPU for each particular VM, the actually consumed CPU and a horizontal line that indicates the SLA threshold. If at any moment the consumed CPU is the same as the allocated CPU and this value is lower than the SLA line, it means that the SLA is being violated.

Notice that, SLA violations occur only if the task is using all the allocated resources, and despite this, the SLA threshold is not respected. Finally, the fourth plot corresponds to the CPU consumed by the Xen Domain-0, and it shows the CPU costs of creating and destroying VMs and of reallocating resources among them.

Zone A. Task 1 requests 100% of CPU and arrives at the system. Since this is the only Task, it receives the whole machine (400% of CPU = 100% requested + 300% surplus).

The required resources for Task 2 are underestimated, so that they are not enough to fulfill its SLA. In this case, the SLA needs 190% of CPU to be fulfilled, but the CPU requested for this VM is only 100%.

When Task 2 is submitted, the total amount of CPU required to satisfy the two tasks is 200% (100% to Task 1 and 100% to Task 2). The surplus 200% is distributed among the two tasks as described in Section 3.2.2 (100% for each). Thus, no SLA is violated, as both tasks have 200% of CPU.

Zone B. Task 3, which requests 100% of CPU, is started, so the total amount of CPU needed to satisfy the requests of the three tasks is now 300% (100% for each one). The remaining 100% is equally shared among
all the VMs, and all the tasks receive 133% of CPU.

Having only 133% of CPU, Task 2 violates its SLA (at the center of Zone B, its assigned CPU is under the SLA threshold, i.e., 190% of CPU). This starts the SLA violation protocol, which reallocates the CPU progressively until all the SLAs are fulfilled. We can see how the surplus CPU assigned to Task 1 and Task 3 is progressively moved to Task 2 until its allocated CPU is over the SLA threshold.

Zone C. All the SLAs are again within their thresholds and when a task finalizes its execution, its allocated resources are freed and redistributed among the other tasks.

If a fourth application would arrive requiring all the surplus resources, the SLA for the Task 2 would be violated. In this case, there would be no enough free resources to solve this violation locally. In this situation, the upper levels would be notified to attempt a global solution for the SLA violation (e.g. modify the minimum allocated resources of tasks, migrate one of them to another node, etc.).

Figure 3.9: Host running three tasks to demonstrate the resource assignment

3.5.3 SLA enforcement

Once we have observed how the VtM estimates the resources and reallocates them, we show how the SLA-driven resource management can solve some problems that simple resource management based on estimation cannot.
The proposed system can overcome SLA violations regarding resource assignment, such as CPU or memory, by assigning as many resources as the task needs. However, when managing metrics hard to evaluate, such as replies per second or execution time, the only component able to detect these violations is the SLA evaluator. To solve this SLA violations, the evaluator requests more resources for the VtM (Section 3.2.2).

SLA-driven resource management is also useful in other cases that would imply economic parameters. Nevertheless, it implies new policies that take into account the benefits and the penalties of executing a task or not and this is supported by upper levels.

A limitation of estimating resources requirement is when tasks have no enough assigned resources. In this case, the system cannot detect if it needs more resources. This experiment demonstrates how the SLA evaluator can solve this situation. In addition, we show how the proposed system is able to adapt to changes in the behavior of the applications.

![Figure 3.10: Host running four tasks to demonstrate the SLA enforcement](image)

The experiment consists of submitting a task which is a service (type Task C) and three tasks that use one CPU each one (type Task A). Task 1 does not use the assigned resource during a large period of time and the others use one CPU each one. Figure 3.10 shows the execution of these four tasks in Host A, which represents a node of a virtualized service provider.

Firstly, the first three tasks are created sequentially and they start exe-
Table 3.4: Tasks description for the SLA enforcement experiment

<table>
<thead>
<tr>
<th>Task</th>
<th>Requested CPU</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
<td>Task C</td>
</tr>
<tr>
<td>2</td>
<td>100</td>
<td>Task A</td>
</tr>
<tr>
<td>3</td>
<td>100</td>
<td>Task A</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>Task A</td>
</tr>
</tbody>
</table>

cutting. However, Task 4 cannot be submitted because the VtM estimates that Task 2 and Task 3 need more than one CPU because the applications consume almost one CPU but sometimes they consume a little more due to the OS. Hence, there is no place for Task 4.

Around the second 500, the resource monitoring detects that Task 1 is not making use of the assigned resources and assigns less resources to this task what implies that resources that were assigned to this task become surplus. At this moment, there are enough free resources for executing Task 4 (it requires one CPU). At this point, the surplus (around 45% of CPU) is equally distributed among the four applications because all have the same priority for sharing surplus resources.

Around second 600, Task 1 wants to make use of the CPU that it has originally requested but it has not enough resources and violates the SLA. However, the resource management system cannot detect that it needs more resources. In this moment, the SLA Evaluator detects this situation and requires more resources by increasing the dynamic priority that controls the surplus resources, and it assigns more surplus resources to this task. Finally, the SLA of Task 1 is accomplished and it has assigned one CPU.

This would not be possible if there would not be the SLA evaluator, because the VtM cannot estimate that an application would need more resources than the ones that it has been assigned with.

### 3.5.4 SLA metric

To evaluate our metric, we have conducted three experiments that consist on running three tasks of type Task C, which CPU consumption varies over time, in a single node during 400 seconds. This makes the system able to dynamically change the resources allocated to each virtual machine and allows demonstrating the benefits of our metric in such a scenario. This node is Host B, runs Xen 3.3.1 [66], and its processor has 5322.20 BogoMips,
which roughly corresponds to 10.4 ECUs. Task submission simulates what happens in real scenarios where users submit their VMs at different times.

Each task has its own SLA, which describes the agreed QoS with the provider. For each experiment, a different SLA metric is used to specify this QoS. In particular, in the first experiment we use the metric referred in Equation 3.1 (from now on denoted as SLA Metric A), in the second one we use the metric referred in Equation 3.2 (SLA Metric B), and in the last one we use the metric referred in Equation 3.6 (SLA Metric C).

We have monitored the allocated CPU to each task, its real CPU usage, and the value of the SLA metric at each moment in time (Figure 3.11). Comparing this value with the guarantee agreed in the SLA, we can know if the SLA is being violated or not.

Top graphic in this figure shows the CPU assignment and usage for each task over time. These values will be the same independently of the metric used in the SLA. We have forced the provider to execute the three tasks, but there are not enough resources to fulfill all the SLAs. This allows us evaluating how each metric deal with real SLA violations.

The provider distributes the resources in such a way that the SLAs of Task 2 and Task 3 are never violated in these experiments, while Task 1 behavior shows all the possible situations regarding SLA enforcement. For this reason, we focus the explanation only on Task 1, which has negotiated an SLA with a CPU requirement of 6 ECUs (1/6 for Metric C).

Figure shows that at the beginning Task 1 is assigned with the whole machine, since it is the only task running in the provider. When Task 2 is submitted at second 150, the CPU is redistributed among Task 1 and Task 2 according to their requirements. Finally, at second 250, when Task 3 arrives, the host redistributes the CPU among the three tasks.

Regarding the CPU usage of Task 1, it starts consuming almost 400% of CPU. After 100 seconds of execution, it reduces its CPU consumption and starts using just 20% of CPU for the next 40 seconds. Then, it begins increasing its CPU usage until it consumes around 150% of CPU.

Next three graphics in Figure 3.11 show the value of the SLA metric over time and its agreed value in the SLA, for Metric A, B and C, respectively.

**SLA Metric A.** It only takes into account the amount of resources assigned to the task. While this value is above of the agreed value in the SLA, this is fulfilled; otherwise, an SLA violation arises.
As shown in the second graphic, due to the redistribution of CPU occurred when Task 3 arrives at the provider at second 250, the value of the SLA Metric A for Task 1 falls below the agreed value in the SLA, thus violating it. The shaded zone identifies the interval where Task 1 is violating the SLA (from second 250 to 400). To avoid this, the provider is forced to refuse executing Task 3, even if this was initially possible since the other tasks were not using all their allocated CPU.

Using this metric, the provider is unable to exploit adequately its resources, as it always assign enough resources to fulfill the SLA, despite that tasks use them or not.

SLA Metric B. This one considers the resource usage of the tasks. Using this metric allows the provider moving freely the CPU assignment if tasks do not use all their allocated resources.

As shown in the third graphic, this metric detects SLA violations that are no responsibility of the provider. In particular, when the CPU usage of Task 1 goes below the agreed value in the SLA (from second 115 to 400), the SLA is being violated. Between second 115 and 250 (light gray zone in the graphic), the SLA is being violated because although the provider is giving enough resources to guarantee the SLA, the task does not use all its assigned CPU. As we have described before, this is a fake violation.
From second 250 to 400, the SLA is being violated (dark gray zone in the graphic) because the provider has not allocated enough resources to the task to fulfill its SLA, and consequently, even if it used all of them, its CPU usage would be under the SLA threshold. This is then a real violation.

Notice that the value of *SLA Metric B* has some start delay with respect to the CPU usage. This is because of the different sampling rates between the SLA monitoring system and the CPU usage monitoring system.

**SLA Metric C.** (the one proposed in this thesis) gets the best from the two other ones, since it considers both the CPU assigned to a task and its real CPU usage. This metric specifies the inverse value in the SLA with respect to the other metrics. For this reason, when using this metric, an SLA violation will happen when the measured value of the metric is greater than the agreed one (1/6 in this particular experiment).

As shown in the fourth graphic, this only occurs in the shaded zone, that is, from second 290 to 400. As commented before, this is the only zone where the task is using all its allocated resources and the provider has not allocated enough resources to the task to fulfill its SLA. On the other side, notice that when using this metric fake SLA violations are avoided, even when the task is not using all its allocated resources (as occurs from second 115 to 290). In these experiments, we have considered that a task is using all its allocated resources when it uses more than 70% of them (this corresponds to a 0.3 value for the $\alpha$ factor in Equation 3.6).

**Summary.** Three well-differentiated phases can be appreciated in the figure. In the first one, the assigned CPU is enough to fulfill the SLA and Task 1 uses all its allocated resources. Consequently, the SLA is not violated with none of the metrics.

In the second one, CPU allocated to Task 1 has been reduced, but the task is not using all its allocated resources. In this case, using the *SLA Metric A* causes low resource utilization because unused resources cannot be assigned to any other task without violating the SLA, while using the *SLA Metric B* causes fake violations.

Finally, in the last phase, the resources assignment is not enough to fulfill the SLA, and Task 1 uses all its assigned resources. In this case, all the metrics violate the SLA.

Table 3.5 summarizes the time while the SLA is violated. It shows the time (in seconds) with real SLA violations, with fake violations, and the
<table>
<thead>
<tr>
<th>SLA Metric</th>
<th>Real</th>
<th>Fake</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>150</td>
<td>0</td>
<td>150</td>
</tr>
<tr>
<td>B</td>
<td>150</td>
<td>135</td>
<td>285</td>
</tr>
<tr>
<td>C</td>
<td>110</td>
<td>0</td>
<td>110</td>
</tr>
</tbody>
</table>

Table 3.5: Duration of SLA violations (proof-of-concept workload)

total time violating the SLA. Notice that, when using \textit{SLA Metric C}, the time with real SLA violations is lower than when using the other metrics. This happens because \textit{SLA Metric C} realizes \textit{Task 1} does not use all its allocated resources between second 250 and second 290. In addition, \textit{SLA Metric B} is the only metric causing fake SLA violations.

### 3.5.5 Real workload

We evaluate the metrics using a real workload, but our testbed is not enough to execute this workload. For this reason, we use a simulated provider composed by 24 nodes. The number of nodes has been chosen to make the provider able to execute the maximum number of concurrent jobs of the workload, while still violating some SLAs to demonstrate the behavior of the different metrics.

We set up three types of nodes in the simulated provider to demonstrate how our proposal can deal with heterogeneous machines. The first type is the one already used in previous experiments (\textit{Host B}) and we use 8 of them. The other 16 nodes are of type \textit{Host A}, and \textit{Host D}, which respectively has 12 and 22.4 ECUs.

The workload corresponds a whole day load obtained from Grid5000 [67] corresponding to Monday first of October of 2007. It is composed by 215 jobs of 10 different applications of type \textit{Task C} which has a variable CPU consumption over time. For each task, it specifies a static requirement of resources, and the dynamic resource usage it performs during its execution. Each task is executed within a VM with 2 VCPUs, since this is enough to fulfill the CPU requirements of all the applications.

To demonstrate the impact on each metric of having dynamic CPU usage pattern over time, we have forced two of the 10 applications to use more CPU than the requested during some parts of their execution. Resource requirements and usages constitute the input of the simulator, which imple-
ments a backfilling policy taking into account these requirements to decide the allocated resources to each task and simulates the specified resource consumption for each task in a given time slot.

Each task has a different SLA. All of them use the same metric for specifying the CPU guarantee, but with a different agreed value, which depends on the task CPU requirement. In addition, the SLA specifies a revenue of $0.38 per hour (we use the pricing of an Amazon EC2 medium instance [9], which has similar features to the VMs used in the experiment).

The SLA is evaluated every 100 seconds, and the penalty is calculated following Equation 3.7 using a maximum penalty ($MAX_P$) of 125%. As the SLA is evaluated every 100 seconds, the maximum number of SLA violations ($\lambda_{max}$) occurs when the SLA is violated every sample. Hence, $\lambda_{max}$ is equal to the number of samples, which is calculated by dividing the application execution time ($T_{exec}$) by the sampling period ($T_{sample}$). Using this information, and considering that $\lambda_{min}$ is zero in our case, we can operate on Equation 3.8 and calculate $\sigma(S_i)$ as follows:

$$\sigma(S_i) = \frac{V_i \cdot T_{sample}}{T_{exec}}$$

Table 3.6 shows the results for each metric, including the number of violations (real, fake, and total), and the final profit. For SLA Metric C, we have also evaluated the impact of the $\alpha$ factor. These results demonstrate that using our metric the provider can get the highest profit, since it avoids all the fake SLA violations and reduces the real ones when the application is not using all the allocated resources.

In particular, we have reduced the incurred penalties in more than a 58% regarding SLA Metric A and more than a 72% compared to SLA Metric B when $\alpha = 0.1$. In addition, note that the lower the alpha factor is, the lower the number of SLA violations is, though the difference is not very noticeable. However, this difference highly depends on the CPU usage of the applications. This occurs because for higher alpha values the task is considered to be using all its allocated resources during more time, which increases the probability of SLA violations imputable to the provider.

We conclude that our metric detects less SLA violations than the others, it is able to avoid fake SLA violations, and it allows the provider to freely redistribute the resources when the tasks are not using them.
### 3.6 Related Work

Several works use virtualization to enable fine-grain dynamic resource distribution among VMs in a single node. For instance, Song et al. [68] develop an adaptive and dynamic resource flowing manager among VMs, which uses dynamic priorities for adjusting resource assignment between VMs in a single host to optimize its performance. Padala et al. [69] introduce an adaptive resource control system (implemented using classical control theory) that dynamically adjusts the resource shares to VMs, which contain individual components of multi-tier enterprise applications in a shared hosting environment, to meet application-level QoS goals.

Steinder et al. [70] take advantage of virtualization features to collocate heterogeneous workloads on any server machine, thus reducing the granularity of resource allocation. Finally, Govindan et al. [71] developed a new communication-aware CPU scheduling algorithm that improves the performance of a default Xen monitor by enabling the underlying scheduler being aware about the behavior of hosted applications.

Our work proposes a more general and extensive solution for managing service providers by joining in a single framework the creation of application-specific VMs on demand, global resource allocation among nodes, and SLA-driven dynamic resource redistribution at node level (based on the redistribution of surplus resources).

Other works combine some of these functionalities, albeit none of them provides all our facilities. In particular, Abrahao et al. [16] propose a dynamic capacity management framework for virtualized hosting services, which is based on an optimization that links a cost model based on SLA contracts with an analytical queuing-based performance model. However, this work does not support the two-level resource allocation. In addition,

<table>
<thead>
<tr>
<th>SLA Metric ($\alpha$)</th>
<th>Real</th>
<th>Fake</th>
<th>Total</th>
<th>Profit ($)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3891</td>
<td>0</td>
<td>3891</td>
<td>231.0</td>
</tr>
<tr>
<td>B</td>
<td>3891</td>
<td>2857</td>
<td>6748</td>
<td>176.6</td>
</tr>
<tr>
<td>C 0.05</td>
<td>1674</td>
<td>0</td>
<td>1674</td>
<td>277.5</td>
</tr>
<tr>
<td>C 0.1</td>
<td>1782</td>
<td>0</td>
<td>1782</td>
<td>271.3</td>
</tr>
<tr>
<td>C 0.2</td>
<td>1877</td>
<td>0</td>
<td>1877</td>
<td>266.9</td>
</tr>
<tr>
<td>C 0.4</td>
<td>1884</td>
<td>0</td>
<td>1884</td>
<td>266.6</td>
</tr>
</tbody>
</table>

Table 3.6: Number of SLA Violations and Profit (Real Workload)
they use a discrete event simulation to evaluate their proposal and not a working implementation of the proposed system.

Similarly, Xu et al. [72] present a two-level autonomic resource management system for virtualized data centers that enables automatic and adaptive resource provisioning in accordance with SLAs specifying dynamic trade-offs of service quality and cost. A novelty of this approach is the use of fuzzy logic to characterize the relationship between application workload and resource demand.

**SLA enforcement.** SLA-driven resource management is still an open topic in the Cloud. Current providers only support the specification of QoS guarantees by means of simple SLAs. For instance, the SLA supported by Amazon EC2 [9] only consider availability, so-called *annual uptime percentage* [13]. In addition, these violations must be monitored and claimed by the user. Furthermore, whereas Amazon EC2 offers different instances according to their ECUs, there is not any guarantee that this computing capacity will be supplied during the whole execution of the instance.

Similarly, GoGrid also considers availability metric [63], including the availability of the storage and the primary DNS. It also offers some metrics related with network performance such as jitter, latency, and packet loss rate. Analogously, 3Tera also provides availability guarantees on its Virtual Private Datacenter [62] in the same way RackSpace does [14]. However, none of the Cloud approaches considers CPU-related metrics as they implement static provisioning of resources. As this can lead to low resource utilization, moving to dynamic provisioning is recommended to increase profit.

Conventional Utility computing systems, such as Autopilot [73], implemented a set of resource management policies that acted in conjunction with distributed sensors that provide real-time performance data. Based on this, a set of actuators implement policies to increase throughput for heterogeneous workloads. However, virtualization brings features that can simplify the process of server consolidation, such as live migration of VMs.

Some work has been carried out for traditional e-business environments. For example, Bennani et al. [74] combine the use of analytic predictive multi class queuing network models and combinatorial search techniques to design a controller for determining the required number of servers for each application environment to continuously meet the SLA goals under a dynamically varying workload.
Another example is Oceano [75], which is a SLA-driven prototype for server farms that enables moving servers across clusters depending on the customer needs. The addition and removal of servers from clusters is triggered by SLA violations. Lodi et al. [76] present a middleware for J2EE clusters that optimizes the resource usage to allow application servers fulfilling their SLA without incurring in resource over-provisioning costs. This resource allocation is done adding or removing nodes from the cluster.

Until now, SLAs have been primarily used to provide QoS guarantees on e-business computing utilities. Due to this, most of the works focusing in this area have used metrics of interest in these environments such as throughput, response time, and availability [16, 72]. None of them has included resource-level metrics as part of its solution, as we suggest in this chapter.

Recent contributions like Dickman et al. [77] are focused on business-level SLAs, where they state that security measures are needed to establish trust between clients and providers. Tserpes et al. [78] exploit a high-level metric, i.e., Quality of Experience (QoE), to allow customers to evaluate providers’ services. Ul et al. [79] claim that SLA offered in Cloud environments must be oriented on business parameters, such as trust, privacy, or security requirements. However, none of these works considers what should be the basis of these high-level SLAs, which is the QoS guarantee based on accurate fine-grain metrics, such as CPU usage, to meet users’ expectations.

Some effort to provide QoS guarantees using SLAs has been carried out in the Grid environment. For instance, Ching et al. [80] and Sashai et al. [81] propose a SLA management system for Grid environments which use the expected completion time of the jobs for defining SLA metrics. Similarly, Menascé et al. [82] define latency, throughput, and availability metrics to minimize execution times and costs in a Grid provider. Kyriazis et al. [83] present an innovative QoS-aware workflow where they evaluate the offered QoS level (i.e., availability and execution time).

Whereas these works neglect resource-level metrics, other works define metrics for providing CPU guarantees in the SLA, but usually in a very limiting way for the resource provider. For instance, Skital et al. [84] allow defining the CPU number, its architecture, and its performance to run a physics application on the Grid. Similarly, Raj et al. [85] use metrics such as the number of processors, the CPU share, and the amount of memory. As we have discussed, when used incorrectly, some of these metrics force the
provider to statically overprovision, while others induce to fake SLA violations. As demonstrated in our evaluation, we overcome these limitations.

3.7 Conclusion

We have presented a resource management cycle that supports fine-grain dynamic resource distribution among virtual environments based on SLAs. The system implements a self-adaptive behavior: each application receives enough resources to meet the agreed QoS, and free resources can be dynamically redistributed among applications.

The evaluation of this resource management policy demonstrates that our system is able to adapt the resource allocations in very short time under changing conditions while fulfilling the agreed performance metrics and solve SLA violations by rescheduling efficiently the resources.

We have proposed a resource-level SLA metric for specifying fine-grain QoS guarantees regarding the computing capacity of a Cloud provider. This metric overcomes the limitations of traditional CPU-related metrics. In particular, it takes into account if services are really using their allocated resources. This allows the provider to implement dynamic resource provisioning, allocating to the different services only the amount of CPU they need. In this way, better resource utilization in the provider can be achieved.

Furthermore, dynamic resource provisioning can be accomplished while avoiding fake SLA violations that could happen with other metrics when customers’ services do not use all the resources they have allocated. Finally, it can be used in heterogeneous machines, since it is based on Amazon’s ECUs and it has been encoded using a real SLA specification.

As demonstrated in the evaluation, using this metric we have reduced the number of SLA violations, since it only detects those that are a real responsibility of the provider. We have described a use case for implementing this metric in a virtualized service provider, which demonstrates that it is feasible to implement it and that it really supports the expected capabilities.

As demonstrated in the evaluation, using this metric reduces the number of SLA violations, since it only detects those that are a real responsibility of the provider. Thus, we are able to increase the provider utilization by running more VMs in the same resources.
Chapter 4

Managing a virtualized provider

4.1 Introduction

Providers have undergone a metamorphosis during the last years because of virtualization. The problem of this new scenario is it brings another layer of abstraction which prevents conventional policies from performing efficiently or correctly. Moreover, this technology brings new capabilities, such as migration, which need to be exploited [86], and thus open many paths in IT resource management.

First of all, they must be able to deal efficiently with the new virtualization capabilities [18, 86]. Secondly, provider management is receiving economical and social pressure to reduce their energy consumption [17]. And thirdly, providers must offer high availability [19] and performance to their users, bound to Service Level Agreements (SLAs).

For this reason, we propose a novel policy for managing a virtualized provider. It mainly focuses on the allocation of VMs in provider nodes according to multiple facets looking for optimizing the provider’s profit. The final profit for the provider is taken into account as a reference value for all the allocation decisions. The revenue obtained by executing a job and hosting a Web-based application is taken into consideration, and different economic costs are assigned to the multiple factors of any VM operation, namely the energy consumption, the incurred SLA violation penalties, the capabilities on fault tolerance, etc.
Chapter 4. Managing a virtualized provider

It is noteworthy that some of these facets have been brought to a new dimension due to the appearance of both Cloud computing paradigm and virtualization technology, which have influence on how they have to be accomplished. One example is scheduling with energy efficiency in mind, which is not longer limited to statically consolidate and power off un-used nodes [87].

Nowadays, thanks to the VM migration technique, a long running job can be moved dynamically from an underused node to another mostly full [17]. In addition of consolidation, the encapsulation that comes with virtualization permits checkpointing any application in a transparent way, which helps to improve provider’s fault tolerance. Nonetheless, application nature needs to be considered at this stage since there is no need to checkpoint stateless applications.

Apart from enabling these new approaches to well-known problems, virtualization also incurs some overheads, such as VM creation and instantiation (which can take minutes), migration, or checkpointing, the extra overhead added by the virtualization hypervisor. Of course, a good management policy should also consider them when making decisions, not only as overheads for that specific VM but also as SLA penalties for the other VMs running in the same node.

Keeping in mind these problems, in this chapter we propose a holistic VM scheduling policy towards nodes of a provider. Our policy analyzes and manages the presented issues, facets and overheads, in a unified way by modeling them as costs or revenues, depending on their nature. Based on this model, the policy tries to find the best VM allocation focusing towards maximizing the provider’s benefit. This Virtual Machine Scheduler manages all the hosts of a provider as it is highlighted in Figure 4.1.

Figure 4.1: Logic architecture of a virtualized provider: virtual machine scheduler
4.2 Modeling a virtualized provider

The allocation of VMs in a provider involves a set of factors that must be considered to perform a proper scheduling. For this purpose, we model these factors related to the operation and profitability of a virtualized provider.

This model calculates all the costs and revenues for a given schedule, deciding whether an allocation of a VM in a given execution platform will provide a benefit or not. This platform is usually a local host of the provider which can execute VMs, but it can refer to assets of an external provider (i.e. outsourcing) or any other kind of resources. However, from the model point of view, all the execution platforms will be seen as a host with different costs and features depending on its nature (local, outsourcing, ...).

A VM is an execution environment where customers can run different kinds of application, paying for them according to a Service Level Agreement (SLA). The benefit per each VM is calculated as the revenue for running it \( R \), and subtracting the costs \( C \) related with the execution. Estimating the costs for a given allocation, the benefit can be calculated as \( B = R - \sum C_i \). Using these values, the model obtains the best scheduling, keeping in mind the final goal: maximizing the overall benefit.

To model a virtualized provider, we have to solve some important issues like: merging multiple units, what to do with running and not running VMs, what time references must be taken, what heterogeneity factors might affect the system, and how SLA terms compute revenues and penalties. All these issues are tackled in detail in next subsections.

4.2.1 Unifying Units

The proposed model uses the associated cost to each factor including incomes, operational costs, and energy consumption. However, each of them has different units, measures, or meanings. For instance, operational costs can be measured in time, while energy is measured in watts hours. For this reason, we need to unify all those parameters in a common way. To achieve this, we define everything as economic revenues or costs, depending on their nature. Following the previous example, operational costs and power consumption can be related to “price of CPU time unit” or “bill for watt hour” respectively. Unifying the units of the different factors, we can merge them and obtain a final score which makes optimization problem easier to solve.
4.2.2 Time reference

Modeling a virtualized provider for scheduling implies having a temporal reference to estimate in which point of its life each VM is. Different works state that user estimations of tasks run time are inaccurate [88]. In our work, it is only used as a reference time to estimate the execution state and deal with queue times, virtualization overheads, low performance, and SLA penalties incurred by these overheads.

For this purpose, the execution time for a task in a dedicated machine ($T_d(vm)$) is used as a reference for the scheduler, and it will be the one used for billing and calculating the final revenue according to the SLA. Therefore, the user will pay for the execution, and the extra time used by the provider to accomplish the VM execution will be charged to this according to the SLA penalty terms.

In addition, different time estimations can be used for a given VM: the extra time added due to operations or virtualization overheads ($T_{extra}(vm)$), the elapsed time ($t(vm)$) on the execution, an estimation of the remaining time ($T_{rem}(vm)$), and an estimation of the remaining time including the extra time due to virtualization overheads ($T_r(vm)$). All these values will be used to estimate future costs for a virtual machine.

\[
\begin{align*}
T_d(vm) &= \text{vm execution time in a dedicated machine} \\
T_{extra}(vm) &= \text{extra time added to vm} \\
t(vm) &= \text{time since vm submission} \\
T_{rem}(vm) &= \text{vm remaining time} \\
&= T_d(vm) - t(vm) \\
T_r(vm) &= \text{vm remaining time including virtualization overheads} \\
&= T_{rem}(vm) + T_{extra}(vm)
\end{align*}
\]

4.2.3 Dealing with non-running VMs

One of the keys of the approach is having a queue holding those VMs which are not running in any host, including those that have not been yet executed or previous execution has failed. For this purpose, it is used the concept of a virtual host with special features, which holds the description of that VMs not allocated in any physical machine.

As the provider wants to run as many VMs as possible in the shortest time, the stay of VMs in the queue has to be minimized. This is achieved by
assigning the maximum cost possible to the allocation of a VM in the queue, which corresponds to no revenue and a maximum SLA penalty. This makes any other allocation (into a physical machine) better than not running the VM. Hence, in a scheduling round, the operations with maximum benefit will be those involving the allocation of a new VM or failed ones into a real host able to handle it.

However, this approach also considers that not executing or waiting to execute a VM can be a desirable solution for some situations. In this case, when a VM is moved from a running status to the queue again, it means executing that task will not be profitable and thus, this VM will be destroyed.

4.2.4 Service Level Agreement terms

The proposed approach considers heterogeneous workloads composed by two kinds of tasks with different goals and functionalities: HPC jobs and Web-based services and applications. These are supported by the provider by offering different SLA terms for each one. The common part of both SLAs is that the customer must pay \( R(vm) \) for executing a VM during an amount of time \( T_d(vm) \) at an agreed pricing \( Pr_{hour}(vm) \), in a similar way Amazon EC2 [9] does:

\[
R(vm) = T_d(vm) \cdot Pr_{hour}(vm)
\]

Furthermore, these SLAs specify the penalties the provider must pay for not offering the desired quality of service, which typically depends on the application and its performance.

On the one hand, the performance of HPC jobs is measured taking into account an execution deadline and it has different violations degrees depending on the amount of elapsed time. This is determined by a soft deadline where time starts to penalize and a hard deadline where it reaches the maximum penalty. To specify these deadlines, the execution time in a dedicated machine is used as the soft deadline and doubling this base time is the hard deadline. Figure 4.2(a) shows an example of the SLA fulfillment outcome for a task according with its elapsed time to execute.

On the other hand, penalties in Web-based services depend on the performance they get which is measured in terms, such as response time or throughput. For example, response time will depend on the assigned CPU and the received requests. Thus, a service with no enough resources to satisfy all the requests will increase its response time and violate its SLA.
In this case, the customer agrees a maximum response time and the SLA is violated if the instantaneous response time goes beyond this. The overall fulfillment depends on the time the instantaneous SLA has been fulfilled (Figure 4.2(b)). For example, if a one hour service has a response time higher than the maximum for 15 minutes, its SLA fulfillment will be 75%.

4.2.5 Dealing with resource heterogeneity

Usually providers own hosts with different capabilities and speeds. Thus, resource management policies must be aware of this fact and try to take profit of this resource heterogeneity. The performance and the speed of applications are highly variable depending on the speed of the nodes and this influences the levels of SLA fulfillment.

Our model currently focuses on CPU speed as the provider targets HPC applications, which directly depend on CPU, and Web-based applications, which bottleneck is also, in part, the CPU. However, it can be easily extended if the applications would be memory or I/O intensive.

As commented before, the model uses some data (e.g. $T_d(vm)$) that it is calculated in a reference dedicated host ($H_{ref}$). We define the speed of this machine as $Speed(H_{ref})$. Of course, a given VM can be executed in a host with different speed regarding the reference machine. To consider this situation, we define a performance factor for a $vm$ running in a host $h$:

$$Perf(h, vm) = \frac{Speed(H_{ref}(vm))}{Speed(h)}$$

This performance factor allows to extrapolate time estimations obtained in the reference host to any other host. For example, given an HPC task with
an execution time $T_d(vm)$ in the reference cost, the model can estimate how long it would take in another host $h$ as $T_d(vm, h) = T_d(vm) \cdot \text{Perf}(h, vm)$.

### 4.3 Cost-benefit analysis

Once users and providers have an agreement about the payment (provider’s revenue) for running a VM, the provider must take care of the execution of that VM and calculate, for each scheduling option, the costs and handicaps that each option implies. For instance, allocating a VM in a host that does not accomplish the VM requirements, or running it in a host whose CPU speed is not enough to meet the task deadline. All these factors can be treated as costs, and these will be subtracted of the revenue, finding the resulting benefit for that VM.

#### 4.3.1 Task requirements

The first issue to be addressed by the scheduling is checking the capability of a host for holding a given VM. This is performed by evaluating VM requirements $\text{Req}(vm)$, which include the hardware (e.g. the required system architecture, type and number of CPUs . . . ), and the software (the libraries needed to execute an application, the hypervisor, e.g. Xen or KVM).

In case the host is not available to execute that VM, the cost of placing this VM into that host can be considered as the maximum penalty for that task as specified in the SLA and the allocations in that host will not be performed. If the maximum penalty is not available, the cost would be infinity. Hence, it will act as a conditional statement which will avoid the execution of that VM into that host. The unfeasible situations are discarded considering it as a boolean function: renting available resources costs zero and maximum penalty whether the resources in the tentative hosts are nonexistent or unavailable. From this description, $C_{\text{req}}(h, vm)$ is derived.

$$C_{\text{req}}(h, vm) = \begin{cases} \infty & \text{if } h \text{ cannot fulfill } \text{Req}(vm) \\ 0.0 & \text{otherwise} \end{cases}$$

#### 4.3.2 Service Level Agreement penalties

As explained in previous section, one of the most important costs comes with the penalties incurred by violating SLA terms. Overcrowding a host
can degrade the performance of the VMs running on it, so we have defined a health heuristic to estimate how the impact of overloading can result. In particular, $He(h)$ function, which depends on the total CPU capacity in a host $h$ ($CPU_{avail}(h)$) and the CPU demand of each VM running on that host ($CPU_{req}(vm_i)$), can be defined as follows:

$$He(h) = \min \left( 1, \frac{CPU_{avail}(h)}{\sum_{vm_i} CPU_{req}(vm_i)} \right)$$

This function follows a behavior already observed in previous works like [29], where the performance is directly related to the ratio of offer/demand of CPU and its sharing consequences. This health function brings a factor of how much the host is able to handle all its load, and in case of overwhelming how much $Perf(h, vm)$ is degraded, so it can be used as heuristic for estimating the execution delays and thus the SLA penalties.

Regarding deadline-based SLAs, we consider two components: the extra time added and the slowdown because of host performance. The first one uses the estimated SLA penalty for a $vm$ running at host $h$ at a given time, which depends on the extra time added because of virtualization operations and overheads ($T_{extra}(vm)$). This is accumulated during the execution to know how much the base execution time is exceeded.

The second part considers the estimated extra time that will be added for this VM in the future, which is derived from $T_{rem}(vm)$ and adjusted according to the host health ($He(h)$) and its performance factor ($Perf(h, vm)$). Notice that we are trying to figure out how much this VM will be delayed beyond its deadline. This penalty is calculated as follows:

$$Pen_{DL}(h, vm) = \max \left( 0, \frac{Pr_{hour}(vm)}{3600} \cdot \left( T_{extra}(vm) + \frac{T_{rem}(vm) \cdot Perf(h, vm)}{He(h)} - T_{rem}(vm) \right) \right)$$

Managing violations for performance-based SLAs (e.g., web applications), which may depend on response time and thus the amount of CPU, is a matter of considering the extra CPU load each operation incurs. During those periods, the response time will be increased and depending on the amount of resources assigned, it might violate the SLA. According to this, we can estimate the penalty for a web application trying to figure out the amount of time it will receive less CPU than required during its remain-
ing execution time ($T_{rem}(vm)$), which basically depends on the host health ($He(h)$) and its performance factor ($Perf(h, vm)$).

\[
Pen_{perf}(h, vm) = \max \left( 0, \frac{Pr_{hour}(vm)}{3600} \cdot \left( \frac{T_{rem}(vm) \cdot Perf(h, vm)}{He(h)} - T_{rem}(vm) \right) \right)
\]

Both penalty estimation formulas are conceptually different, but we can generalize a single formula for both SLA types by taking into account that $T_{extra}(vm) = 0$ for web applications. According to this, we can conclude that the cost associated to SLA penalties can be calculated as follows:

\[
C_{pen}(h, vm) = Pen_{DL}(h, vm)
\]

Estimated SLA penalties are not only used for calculating the current penalties. We can extend this idea to assess estimated penalties of potential situations. For instance, we can estimate the SLA penalty of a VM ($vm_1$) in a given host ($h$) when adding a new VM ($vm_2$): $\hat{Pen}(h, vm_1, vm_2)$. This will be useful to calculate the cost of a new operation (see next subsection for details on how this estimation is used to calculate the cost of virtualization operations). This estimation uses the tentative health, $\hat{He}(h, vm_2)$, which evaluates the status of the host when adding $vm_2$. The general penalty formula which takes into account a tentative allocation is:

\[
\hat{Pen}(h, vm_1, vm_2) = \max \left( 0, \frac{Pr_{hour}(vm_1)}{3600} \cdot \left( \frac{T_{rem}(vm_1) \cdot Perf(h, vm_1)}{\hat{He}(h, vm_2)} - T_{rem}(vm_1) \right) \right)
\]

Finally, notice that all those estimations are forcing all penalties to be positive, since there is no reward if the task finishes earlier or the web service gets a better performance by running in a faster machine. Notice also that scaling these penalties by the price per VM per hour, the model is obtaining economic values for every risk of delay.

4.3.3 Infrastructure

Traditionally, enterprises who widely rely on IT systems had to make great investments at their first steps for building the needed IT infrastructure, which are considered as capital expenditures (CAPEX). This initial investment was often a significant problem, specially for small and medium
sized businesses. Nowadays, this fact has changed a lot thanks to the well-established Cloud billing model (i.e. pay-as-you-go). Actually, investments on Cloud infrastructures are smoothed over time versus a large lump sum, thus converting the aforementioned capital expenditures into operating expenses (OPEX) [20].

In this sense, the cost of the Cloud infrastructure itself includes the nodes (taking into account their amortization), the space required to deploy them, etc. This cost is distributed among the VM running on that host:

\[ C_{fix}(h, vm) = \frac{Pr_{hour}(h) \cdot T_d(vm)}{\#VM(h)} \cdot T_r(vm) \]

### 4.3.4 Energy consumption

One of the costs that varies with the utilization of the system is the energy consumption and this needs to be modeled to predict the energy consumption associated with the execution of each VM. This modeling is obtained by measuring the power consumption of a physical machine stressed with different loads and from this information a curve which models the power consumption can be extracted and used to obtain the energy consumption of a host with a given occupation, \( Power(h, o) \cdot T_r(vm) \).

Using this information, the model can estimate the cost caused by the energy consumption of each VM, \( C_{pwr}(h, vm) \).

\[
\begin{align*}
Power(h, o) &= \text{power consumption of } h \text{ at occupation } o \\
O(h, vm) &= \text{occupation of } h \text{ where } vm \text{ going to run} \\
C_{pwr}(h, vm) &= \frac{CPU_{req(vm)}}{\sum_{vm_i} CPU_{req(vm_i)}} \cdot Power(h, O(h, vm)) \cdot Pr_{KW} \cdot T_r(vm)
\end{align*}
\]

The power cost is also expressed in economic units by calculating the cost of the energy consumed by the VM.

### 4.3.5 Virtualization overhead

As seen in Chapter 2, virtualization makes the overall system management more complicated and requires well-designed policies which take VM management problem into account. One of the strengths of this proposal is its capability to deal with virtualization management overheads.

One of these overheads is the VM creation overhead, which is the time required to create and start a new VM before it is ready to run tasks
When a new VM needs to be started in the system the time to create and boot up it in each host is considered as a cost.

The other one is the migration overhead $T_m(h, vm)$, which is the one incurred when moving a running VM from a node to another. The time required to migrate a VM is also taken into account for that new tentative allocation. This cost reduces the number of migrations and so, prevents the same VM from moving too often.

$$Host(vm) = \text{host where } vm \text{ is allocated}$$

$$T_c(h, vm) = \text{time of creating } vm \text{ in } h$$

$$T_m(h, vm) = \text{time of migrating } vm \text{ from } Host(vm) \text{ to } h$$

When migrating a VM, this cost must take into account the remaining execution of a VM since it is not worth to move a VM which will finish in small time. This is done using a migration penalty, which considers the estimated remaining execution time:

$$P_m(h, vm) = \begin{cases} 
2 \cdot T_m(h, vm) & T_r(vm) < T_m(h, vm) \\
\frac{T_m(h, vm)^2}{T_r(vm)} & T_r(vm) \geq T_m(h, vm)
\end{cases}$$

However, virtualized environments have other operational problems. For example, when we perform an action in a VM, another action can provoke non-desired situations like migrating VMs that are not ready yet, or trying to destroy a VM which is being migrated. For this reason, while the VM is being operated (created, migrated, checkpointed...) we avoid any other operation. This is achieved by setting an infinity penalty to actions to already operated VMs:

$$P_{virt}(h, vm) = \begin{cases} 
0.0 & \text{if } Host(vm) = h \\
\infty & \text{if action performed in } vm \\
T_c(h, vm) & \text{if } Host(vm) = \emptyset \\
P_m(h, vm) & \text{if } Host(vm) \neq h
\end{cases}$$

Another issue to be considered is performing more than one action at the same time in the same host, which will be referred as concurrency from now on. This situation can generate a race for the resources (e.g. disk, CPU) among the different VMs which will add an additional overhead. For this purpose, the model firstly calculate the concurrency for a given host
Chapter 4. Managing a virtual provider

4.4 Managing virtual machine placement

The proposed approach focuses on maximizing the provider’s benefit and it consists in taking profit of the capabilities that a virtualized environment
Managing virtual machine placement

offers. This is achieved using the model of a virtualized provider based on the revenues and costs associated with each process, including job execution, turn on or off nodes, and outsourcing; and other new features brought by virtualization such as customized environment creation, and migration.

4.4.1 Scheduling policy

The key for increasing the provider’s benefit is finding the best VM location at each moment, from a point of view of revenue opportunities and required costs. This is achieved by assigning an economic score to each possible VM allocation, which results from the sum of all the individual revenues and costs (conceptually negative) associated with the possible execution of a VM in the target host.

All these factors needs to be merged in a common unit and the chosen one is monetary units, as presented in Section 4.2. This unit is the more intuitive and understandable metric by both clients and providers.

Once the scheduling knows the revenue and costs of executing each VM $VM_i$ in each possible host $H_j$, it puts all this information together in a matrix $\#Hosts \times \#VMs$, with each cost value as $(host, VM)$. For example, after calculating the previously presented revenues and costs, we could obtain the following score matrix, where $VM_1$ is currently running in $H_M$, $VM_2$ in $H_3$, $VM_3$ in $H_5$, $VM_4$ in $H_1$, and $VM_N$ in $H_6$:

<table>
<thead>
<tr>
<th></th>
<th>$VM_1$</th>
<th>$VM_2$</th>
<th>$VM_3$</th>
<th>$VM_4$</th>
<th>...</th>
<th>$VM_N$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$H_1$</td>
<td>15.2</td>
<td>15.2</td>
<td>∞</td>
<td>15.2</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>$H_2$</td>
<td>∞</td>
<td>7.8</td>
<td>7.8</td>
<td>7.8</td>
<td>∞</td>
<td></td>
</tr>
<tr>
<td>$H_3$</td>
<td>10.3</td>
<td>10.3</td>
<td>∞</td>
<td>10.3</td>
<td>10.5</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$H_M$</td>
<td>11.0</td>
<td>∞</td>
<td>11.0</td>
<td>11.0</td>
<td>∞</td>
<td></td>
</tr>
<tr>
<td>$H_V$</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td></td>
</tr>
</tbody>
</table>

On the one hand, calculating the obtained revenue, which is specified in the SLA, takes into account the execution time of the application in a reference node. On the other hand, the final cost is an aggregation of the individual costs that checks the requirements ($C_{req}(h, vm)$); those associated with the VM execution, such as renting ($C_{rent}(h, vm)$), or power ($C_{pwr}(h, vm)$); those that take into account the virtualization overheads ($C_{virt}(h, vm)$); and
finally, the SLA penalties. Hence, the function to be maximized is:

$$B(h, vm) = R(vm) - \sum C_i(h, vm)$$

The SLA penalties calculation is different if the VM is already running or it needs to be allocated for first time. For the first one, when building the matrix, a VM will always have those costs associated with previous violations. For the second one, when an action needs to performed (creation or migration), the virtualization overhead takes into account the violations that operation will perform on the VM’s already allocated in that host.

**Scheduling example**

We illustrate this with an example. Let’s assume that we have a given host, $H_A$, which has 2 CPUs at 3.00 GHz each one and lasts 30 seconds to create a VM. $H_A$ is running two VMs with a requirement of 1 CPU: $VM_1$ and $VM_2$. The first one is running an HPC task, which deadline is due in one hour and already has 60 seconds of extra time added by previous virtualization operations. $VM_2$ is running a web service that will finish in one hour. Note that these time references are based on the applications performance on the reference host. However, to keep this example simple, we assume that $H_A$ has the same capabilities than the reference host, and thus $Perf(H_A) = 1$. In such an scenario, $He(H_A) = 1$ and, therefore, the current estimations of SLA penalties for these two VM are calculated as follows:

$$C_{pen}(H_A, VM_1) = \max \left(0, \frac{Pr_{hour}(VM_1)}{3600} \cdot \left(60 + \left(\frac{3600 \cdot 1}{1} - 3600\right)\right)\right) = \frac{Pr_{hour}(VM_1)}{3600} \cdot 60$$

$$C_{pen}(H_A, VM_2) = \max \left(0, \frac{Pr_{hour}(VM_2)}{3600} \cdot \left(\frac{3600 \cdot 1}{1} - 3600\right)\right) = 0$$

Let’s consider that the provider needs to schedule a new VM $VM_3$ in the system and it needs to check the profitability for placing this new VM in $H_A$. To evaluate the chance to start the new VM, $VM_3$, in that host, the model needs to calculate the cost for this creation operation. Considering that $VM_3$ will host an HPC job with a deadline of 2 hours in the SLA and requiring also 1 CPU, the estimated SLA penalty for the new VM would be (notice that the host health after adding this VM would be $\frac{2}{3} = 0.6$):
\[ C_{pen}(H_A, VM_3) = \max \left( 0, \frac{Pr_{hour}(VM_3)}{3600} \cdot \left( 30 + \left( \frac{7200 \cdot 1}{0.6} - 7200 \right) \right) \right) \]

Additionally, the virtualization cost for creating that VM (i.e. an extra time of 30 seconds) has also to consider an estimation of the SLA penalties incurred in the others already running in that host after adding \( VM_3 \):

\[ C_{op}(H_A, VM_3) = \left( \frac{Pr_{hour}(vm) + Pr_{KWh}}{3600} \right) \cdot 30 + \sum_{vm_i}^H \hat{Pen}(H_A, vm_i, VM_3) \]

\[ \sum_{vm_i}^H \hat{Pen}(H_A, vm_i, VM_3) = \hat{Pen}(H_A, VM_1, VM_3) + \hat{Pen}(H_A, VM_2, VM_3) \]

\[ \hat{Pen}(H_A, VM_1, VM_3) = \max \left( 0, \frac{Pr_{hour}(VM_1)}{3600} \cdot \left( 0 + \left( \frac{2690}{0.6} - 3600 \right) \right) \right) = \frac{Pr_{hour}(VM_1)}{2} \]

\[ \hat{Pen}(H_A, VM_2, VM_3) = \max \left( 0, \frac{Pr_{hour}(VM_2)}{3600} \cdot \left( 0 + \left( \frac{2690}{0.6} - 3600 \right) \right) \right) = \frac{Pr_{hour}(VM_2)}{2} \]

\[ \sum_{vm_i}^H \hat{Pen}(H_A, vm_i, VM_3) = \frac{Pr_{hour}(VM_1) + Pr_{hour}(VM_2)}{2} \]

\[ C_{op}(H_A, VM_3) = \left( \frac{Pr_{hour}(vm) + Pr_{KWh}}{3600} \right) \cdot 30 + \frac{Pr_{hour}(VM_1) + Pr_{hour}(VM_2)}{2} \]

**Solving scheduling**

Once the matrix is filled with the expected cost for each possible allocation, the solving algorithm tries to find those combinations with better benefit for the overall system. However, before taking decisions, the algorithm must prepare this matrix to compare values in a suitable way. First of all, values for each VM must be centered to the value of the current VM situation. From each cell, \( \langle H_j, VM_i \rangle \), the current benefit of maintaining \( VM_i \) in the current host is subtracted (i.e. this is the value of cell \( \langle H_{cur}, VM_i \rangle \) if \( VM_i \) is running in \( H_{cur} \)). From now on, each cell contains the improvement or degradation of moving a VM from its current host to the host corresponding to this cell. Positive scores mean improvement and negative scores mean degradation.

When the matrix has been already preprocessed, we start the optimization. This process selects on each iteration the highest value of the matrix,
which represents the best movement to be performed in the system. After moving the corresponding VM to the new host machine, the matrix is refreshed with the new values. Then, it iterates until the cost matrix has no negative values, or the algorithm considers it has performed enough iterations for the current round. However, there is always a chance of not converging and entering in a periodic movement cycle, so a limit number of movement per scheduling round is applied.

When the matrix reaches a stable state where all changes are negative or zero (no more improvements can be done), or the number of movements has reached a given limit, it is assumed a suboptimal solution for the current system configuration has been found. Algorithm 1 shows the matrix optimization algorithm.

**Algorithm 1** Algorithm for optimizing allocation matrix

\[
\begin{align*}
M & := \text{Matrix [hosts][VMs]} ; \\
- \text{Fill } M : \\
& \quad - \text{Add revenues for each VM;} \\
& \quad - \text{Substract task requirements to each Host,VM;} \\
& \quad - \text{Substract infrastrucyte cost to each Host;} \\
\text{While } M \text{ has positive values do:} \\
& \quad - \text{Update } M : \\
& \quad \quad - \text{Substract energy costs for each Host,VM;} \\
& \quad \quad - \text{Substract SLA penalties costs for each Host,VM;} \\
& \quad \quad - \text{Substract renting costs for each Host;} \\
& \quad \quad - \text{Substract operation costs for each Host,VM;} \\
<h,v> & := \text{biggest position on } M ; \\
o & := \text{current host for } v ;
\end{align*}
\]

\[
\begin{align*}
& \quad - \text{Re-schedule VM } v \text{ from Host } o \text{ to Host } h ; \\
& \text{If (iterations limit reached) then:} \\
& \quad \text{break;} \\
& \text{End If} \\
& \text{End While}
\end{align*}
\]

Note that this algorithm is based on the *Hill Climbing* one, which is greedy. Nonetheless, in this situation it finds a suboptimal solution much faster and cheaper than evaluating all possible configurations. Each step brings to a more optimal configuration until there are no better configurations or an iteration limit is reached. The algorithm complexity has an upper
boundary of $O(#Hosts\cdot #VMs) \cdot C$ since it iterates over the \langle host, VM \rangle matrix $C$ times. In addition, in the current study case, some of the constraints help to reduce the search space, i.e. the resource requirement constraint discards a great amount of combinations at the beginning of the algorithm.

4.4.2 Management procedures

The proposed approach performs a scheduling of the VM’s keeping in mind the benefits and the costs associated with each allocation. At this point, it is time to perform the actions the scheduler has selected. These actions include the simple VM management like the creation of new VM’s to run simple tasks, or the migration a VM between two nodes, but also more complex actuators that will be detailed in next subsections.

SLA enforcement

To provide SLA enforcement, we use the cycle presented in Chapter 3. As stated before, resource requirements of VMs ($CPU_{req}(vm)$) can vary along time. To handle this situation, the hosts monitors and reassigns resources to the VMs according to their actual requirement. These fluctuations have an effect on the host health ($He(h)$). In particular, when a VM requires more resources ($CPU_{req}(vm) = CPU_{req}(vm) + \delta$), the health of the host where this VM runs becomes worse. This could result in a SLA violation since some VMs running on that host could receive less resources than required.

According to this, the scheduler periodically assesses the hosts health, and when it detects that a host health becomes worse (i.e. some VMs in that host could violate their SLAs), it initiates a rescheduling aiming for the maximum profitability according to the new scenario of resource requirements. This is accomplished using the new health values in the scheduling policy described before.

Power efficiency

The power efficiency of the presented approach is based on the consolidation of VMs to reduce the amount of running nodes and save the energy they consume. This consolidation is achieved by the scheduling policy which evaluates the energy costs and tries to optimize the resource usage taking into the other costs.
However, consolidating is not enough to save energy. For this purpose, one of the key decisions is determining the amount of operative nodes are needed to execute the workload at each moment to save the power consumption of those idle nodes [89]. This is based on deciding when a non-used node can be turned off to save its idle power consumption, or turned on again (i.e. using Wake-on-LAN) to run VMs and try to fulfill the tasks SLAs in the same way it was done in previous work [29].

In this work, when to turn on or off a node is decided using a simple approach where a given amount of idle machines is always maintained. For example, if the provider is using 10 nodes and it has a margin of 2 nodes, it would require 12 nodes to be up and running while only 10 are used. More complex approaches like workload prediction could be also applied to make a more accurate policy.

Once the number of required nodes is known, the model is also used to select which nodes needs to be turn on or off. In this situation, the presented approach tries to schedule a fake VM to the system and calculates the costs this process would cost to the provider. Depending on the current target, turning on or off a node, the algorithm would select the one with biggest or lowest benefit respectively.

**Fault tolerance**

The proposed model is also able to support fault tolerance. Each host has a given availability factor ($Up(h)$) between 0 and 1 which will be zero if it is 90% or less (this is set as minimum uptime) and 1 if the node is always up, i.e. there are no failures. To take this into account, we multiply the final benefit of an allocation by this uptime factor, which can be expressed as $B(h, vm) = Up(h, vm) \cdot B'(h, vm)$. However, some VMs have permissiveness to failure, so no uptime factor is applied to them.

$$F_{tol}(vm) = \begin{cases} 
0 & \text{if } F_{tol}(vm) \\
1 & \text{otherwise}
\end{cases}$$

$$Up(h, vm) = \begin{cases} 
1 & \text{if } F_{tol}(vm) \\
Up(h) & \text{otherwise}
\end{cases}$$

In addition, our approach supports the capability to recover executions from a checkpoint. In this sense, the system periodically decides to perform a checkpoint according to the profitability of doing this checkpoint. First of all, it checks if the VM is stateless and in that case, it does not perform any
checkpoint since it is not useful. Otherwise, it checks the execution time elapsed after the last checkpoint and if it is worth taking into account the required time to perform a checkpoint, it will run the checkpoint process.

Finally, to complete the recovery mechanism, when a node fails, the VMs that were being executed on that host are moved to the virtual host with an infinity cost. Once a checkpointed VM has been scheduled to be executed again, it is recovered from the last checkpoint.

4.5 Experimental environment

The proposed approach manages a provider which is able to execute batch jobs and host web services on top of the virtualized provider presented in this thesis. Experiments aim to demonstrate the capability of the proposed scheduling policy to manage a virtualized provider and improve the provider’s benefit.

To evaluate our proposal, we use a simulated provider which supports different scheduling policies and is able to run test multiple workloads. Finally, this simulator provides multiple metrics like power consumption, node usage, performance, and benefit. Next subsections explain into detail each one of the components of the experimental environment.

4.5.1 Simulator

A virtualized provider is a variable environment and not flexible enough for testing policies with different configurations. In addition, results are difficult to obtain and reproduce because of the variability of the system. To solve this problem, we have developed a simulated environment which mimics the behavior of the provider described in Chapter 2, Chapter 3, and following the architecture presented in Appendix A.

This simulator is able to monitor performance and energy consumption, which is model using actual measures obtained with a power meter. Among other factors, it also takes into account the virtualization overheads (including creation, migration, checkpoint mechanisms presented in Chapter 2), the resource distribution behavior between VMs in a node (described in Chapter 3), and the ability to turn nodes on and off (including the ability to simulate node crashes). It is also able to use multiple scheduling policies and it obtains metrics like energy consumption, node usage, and performance.
An early stage of this simulator was already presented in [29], where we presented a framework for achieving energy-efficient datacenters by using machine learning techniques and power-aware consolidation algorithms. This power-aware simulator, based on OMNet++, is described in detail and validated in [90]. In addition, it was used to test initial versions of the current scheduling policy in our previous work [30].

**Validation.** To validate the simulator energy consumption, a real workload has been submitted to a single node which provokes different situations and we have measured its CPU usage and energy consumption. This validation process shows our simulator has an error of less than -0.43 Wh over 93.49 Wh of real energy consumption. Regarding the instantaneous error, it has less than 6.23 W of absolute error which represents relative error of 0.02%. Figure 4.3 shows this validation by comparing the measured consumption and the simulated one.

![Figure 4.3: Simulator power consumption validation.](image)

4.5.2 Scheduling policies

To evaluate our proposal, we have developed an implementation of the scheduling policy presented in this chapter. It is called “Cost-driven Scheduling Policy” (CDSP) and supports different variations of the presented scheduling by adjusting some parameters. For example, we can take into account the virtualization penalty, add outsourcing capabilities, etc.

To give a lower boundaries in terms of SLA fulfillment and power consumption, we use a Random policy, which decides the placement randomly without taking into account any factor, and a Round-Robin policy. The latter policy gets the lowest consolidation possible as it always place the VMs in the host with the lower usage.

We also use a more plausible approach which is consolidation-aware like
Experimental environment

backfilling (BF). We use a variation of the classical EASY-Backfilling policy for job scheduling [91]. This variation follows the same idea that regular backfilling but with VMs instead of jobs. Thus, it fills nodes with idle resources with small VMs to increase consolidation. In addition, we use an extension of this approach which supports migration (BF+M), which increases global system consolidation by migrating VMs from mostly unused machines to those close to be full.

Notice that, CDSP policy follows the backfilling basis but in this case, it takes into account the virtualization management overhead. In addition, it is driven by the proposed economical model and thus, it can deal with the tradeoff between SLA penalties and energy consumption.

Another good policy to compare, is the one just based in Scores which does not consider economic factors [30]. As it does not support the economic model it cannot automatically deal with the tradeoff between performance and energy and depends on the administrator setup.

Finally, we give an upper boundary in terms of SLA by terms of the Perfect policy. This policy gives the minimum energy consumption possible to execute the workload fulfilling the 100% of the SLA. This is an analytical policy which implementation is NP complete and thus, it is not feasible for an on-line model.

4.5.3 Provider’s configuration

The experiments consist of the simulation of a whole virtualized provider with a different amount of local nodes with different features like architecture, speed and size, and it is also able to outsource to other providers.

Provider behaves as a IaaS provider similar to Amazon EC2 where users will rent some VMs to run their tasks. In particular, we will use the EC2 pricing for medium instances with high CPU load, which have a cost of 0.17€/hour (EC2 pricing in Europe). However, the accounting will be performed in a more accurate way and the user will pay per each second and not each hour.

The electricity pricing used is the Spanish one, that is 0.09€/KWh [92]. Furthermore, to calculate the cost of the nodes, we also take into account the amortization of the servers (in 4 years) and the space (in 10 years) required to deploy them using a price of 2000€/m². As external providers, two Amazon EC2 datacenters are considered: US and Europe. Both have an average
creation time of 300 seconds and a cost of 0.19 and 0.17 € respectively.

Regarding the local resources, provider will have local nodes with different speeds and features according to the experiment and its goal. These nodes can have two different architectures Xeon or Atom. The first two types have a good performance and a high power consumption while C type is old machines with medium performance and high consumption. Atom machines are cheaper and have poor performance but very low power consumption. Table 4.1 presents the features of the different nodes including: architecture, CPU features, memory, power consumption (obtained using a power meter), a reference pricing, and the space ( racked machines).

<table>
<thead>
<tr>
<th>Power (W)</th>
<th>Arch</th>
<th>CPU,(@MHz)</th>
<th>Mem,(GB)</th>
<th>Min.</th>
<th>Max.</th>
<th>Price (€)</th>
<th>m^2/node</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Xeon</td>
<td>4@3000</td>
<td>16</td>
<td>230.0</td>
<td>317.9</td>
<td>1000.0</td>
<td>0.1</td>
</tr>
<tr>
<td>B</td>
<td>Xeon</td>
<td>4@2660</td>
<td>16</td>
<td>228.6</td>
<td>316.1</td>
<td>1000.0</td>
<td>0.1</td>
</tr>
<tr>
<td>C</td>
<td>Xeon</td>
<td>2@2392</td>
<td>2</td>
<td>228.1</td>
<td>315.3</td>
<td>1000.0</td>
<td>0.1</td>
</tr>
<tr>
<td>D</td>
<td>Atom</td>
<td>2@1600</td>
<td>4</td>
<td>37.2</td>
<td>39.8</td>
<td>500.0</td>
<td>0.05</td>
</tr>
</tbody>
</table>

Table 4.1: Node features according to its type

4.5.4 Provider workload

Our approach is able to deal with heterogeneous workloads composed by batch HPC jobs and Web-based services and has a similar amount VMs of both types running at the same time. For testing purposes, the former is a Grid workload obtained from Grid5000 [67] on the week that starts on Monday first of October of 2007. The latter workload was obtained from an anonymous European ISP and was collected during 2009. It results from the aggregation of several services and the used profile is of a whole week, thus representing different classical increases of load of each weekday and the decreases of load of weekends.

The workload is composed by HPC jobs and web-applications (supported by our model). These applications need to be modeled and introduced in the framework which will simulate the provider’s behavior. Table 4.2 shows the details of the heterogeneous workload. Table firstly shows there are a total of 1973 HPC tasks which are CPU intensive (with several CPU consumptions) and behave in a linear way depending on the speed of the node and it has been introduced in the simulator. In addition, the tasks have
different durations: short (less than 600 seconds), medium, and long (more than 20000 seconds); and they are distributed among time and a mean of 35 and a maximum of 204 tasks are running concurrently.

<table>
<thead>
<tr>
<th>Type</th>
<th>SLA type</th>
<th>#VMs</th>
<th>Description</th>
<th>Average duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>HPC</td>
<td>Deadline</td>
<td>12</td>
<td>1 CPU</td>
<td>66213s</td>
</tr>
<tr>
<td>HPC</td>
<td>Deadline</td>
<td>18</td>
<td>1 CPU</td>
<td>57913s</td>
</tr>
<tr>
<td>HPC</td>
<td>Deadline</td>
<td>89</td>
<td>1 CPU</td>
<td>8456s</td>
</tr>
<tr>
<td>HPC</td>
<td>Deadline</td>
<td>854</td>
<td>1 CPU</td>
<td>5346s</td>
</tr>
<tr>
<td>HPC</td>
<td>Deadline</td>
<td>27</td>
<td>2 CPU</td>
<td>206s</td>
</tr>
<tr>
<td>HPC</td>
<td>Deadline</td>
<td>49</td>
<td>2 CPU</td>
<td>594s</td>
</tr>
<tr>
<td>HPC</td>
<td>Deadline</td>
<td>82</td>
<td>2 CPU</td>
<td>18485s</td>
</tr>
<tr>
<td>HPC</td>
<td>Deadline</td>
<td>725</td>
<td>2 CPU</td>
<td>101s</td>
</tr>
<tr>
<td>HPC</td>
<td>Deadline</td>
<td>29</td>
<td>3 CPU</td>
<td>15380s</td>
</tr>
<tr>
<td>HPC</td>
<td>Deadline</td>
<td>88</td>
<td>4 CPU</td>
<td>19911s</td>
</tr>
<tr>
<td>Web</td>
<td>Performance</td>
<td>6</td>
<td>Monday</td>
<td>86400s</td>
</tr>
<tr>
<td>Web</td>
<td>Performance</td>
<td>6</td>
<td>Tuesday</td>
<td>86400s</td>
</tr>
<tr>
<td>Web</td>
<td>Performance</td>
<td>6</td>
<td>Wednesday</td>
<td>86400s</td>
</tr>
<tr>
<td>Web</td>
<td>Performance</td>
<td>6</td>
<td>Thursday</td>
<td>86400s</td>
</tr>
<tr>
<td>Web</td>
<td>Performance</td>
<td>6</td>
<td>Friday</td>
<td>86400s</td>
</tr>
<tr>
<td>Web</td>
<td>Performance</td>
<td>5</td>
<td>Saturday</td>
<td>86400s</td>
</tr>
<tr>
<td>Web</td>
<td>Performance</td>
<td>5</td>
<td>Sunday</td>
<td>86400s</td>
</tr>
<tr>
<td>Web</td>
<td>Performance</td>
<td>20</td>
<td>One week</td>
<td>604800s</td>
</tr>
</tbody>
</table>

| Total HPC | 1973 | 10979339s |
| Total Web | 65   | 15552000s |

Table 4.2: Workload details

On the other hand, the web-applications are customers who want to run their services on our provider. We use SPECweb2009 e-Commerce application [93] as web-based application. The model for this application has been obtained by stressing it (deployed on a Tomcat v5.5) with different input loads and with different processing units. This model focuses on relating the response time with both incoming users’ load and CPU usage of the server (note that the application used is CPU-bounded). The behavior of the application can be divided in three groups: an stationary response time when the incoming load causes a CPU utilization less than 60%; slightly increased when this CPU utilization is between 60% and 80%; and polynomial when the server is overloaded (i.e. CPU utilization greater than 80%).

Finally, the number of VMs concurrently running in the provider and the features of the local nodes will be the parameters to dimension the provider. Depending on the experiment and its purpose, we will use providers with different setups.
4.5.5 Service Level Agreements

The key innovation regarding typical providers is the addition of SLA management and as it has been already presented, our propose supports applications with two different SLAs. To demonstrate the capability of the SLA management capabilities of the presented approach, a very restrictive service level will be used and all the extra costs will be taken into account by the provider. Nevertheless, real providers can use a more relaxed approach.

The provider will monitor and account the execution of its tasks and it will calculate the percentage of time in which each SLA has been violated, \( \sigma(\text{App}_i) \), which will be used to measure the system performance. Moreover, the provider uses this violation ratio as the input for determining the SLA penalty \( \text{Pen}(\text{App}_i) \). This will be the final penalty the provider must pay to the customer for non accomplishing the agreed SLA and it is calculated as a percentage of the revenue acquired \( \text{Rev}(\text{App}_i) \) when fulfilling the corresponding SLA. It is measured by using a Gompertz function \( \text{Gom}(\sigma(\text{App}_i)) \), which is a kind of sigmoid function where growth is slowest at the start and end of a time period and it allows reducing the penalties for low SLA violations. Finally, the penalty for \( \text{App}_i \) is defined as:

\[
\text{Pen}(\text{App}_i) = \text{Rev}(\text{App}_i) \cdot \frac{\text{Gom}(\sigma(\text{App}_i))}{100}
\]

As it has been previously mentioned, a deadline based SLA (\( DL \)) will be used for HPC jobs and it will use the execution time in a dedicated machine as base speed and the final execution time. On the other hand, the SLA for web-applications will be performance based (\( P \)) and the metric will be the response time. The threshold will be 8 seconds, which is the limit a user can wait for a web page as stated in [94].
4.6 Evaluation

To evaluate our proposal, we compare its behavior against other policies. In particular, we analyze several metrics like power consumption and system performance, but focusing in the final goal: provider’s benefit.

4.6.1 Scheduling algorithm scalability

First experiment consists in evaluating the scalability of the scheduling algorithm. For this purpose, a new VM will be submitted every 100 seconds. Every VM fits in every host and any restriction apply, which will be the worst case and will give an upper boundary for the efficiency of the algorithm.

![Figure 4.5: Algorithm scalability according to the number of hosts and VMs](image)

Figure 4.5 shows the required number of operations depending on the number of VMs and the number of nodes. The algorithm powers new hosts on as soon as it needs them (following the policy presented in Section 4.4.2). Until this point, the algorithm fills those nodes and once all of them are full, the model decides there is no more room for VMs and starts keeping the new requests in the queue.

Results show the scheduling policy has a linear cost (as seen in Section 4.4.1. In addition, it takes less than 2 seconds to schedule a VM in a provider of 1000 nodes which is executing more than 4000 VMs. Moreover, the overhead when executing the already presented workload during one week is less than 0.05%.

The upper boundary of the scheduler’s complexity has demonstrated to be linear as it follows $O(\#\text{Host} \cdot \#\text{VM}) \cdot C$. In addition, this reasoning can be easily distributed defining a hierarchical architecture, where each scheduler controls a set of hosts.
4.6.2 Energy consumption vs. SLA fulfillment tradeoff

As it has been already presented, consolidation is applied to be able to get idle nodes to be turned off. Nevertheless, a too aggressive node turning off policy will incur in not enough resources to execute tasks while a passive one will have bigger energy consumption.

To evaluate this tradeoff, we develop a policy that uses two thresholds to decide whether to turn on or off nodes. These two thresholds are: the minimum Working hosts threshold $\lambda_{\text{min}}$ and the maximum Working hosts threshold $\lambda_{\text{max}}$. When the ratio of working nodes goes over $\lambda_{\text{max}}$, the scheduler must start turning on stopped nodes. The nodes to be turned on are selected according to a number of parameters, including its reliability, boot time, etc. On the other hand, when the ratio of working nodes goes below $\lambda_{\text{min}}$, the scheduler can start turning nodes off. The scheduler selects those idle machines according to its cost. This cost results from the aggregation of benefits (matrix row) and taking into account the number of negative infinities. Those nodes with a lower global benefit are selected to be turned off. Finally, to define a minimum set of operative machines, the scheduler can use the min$_{\text{exec}}$ parameter.

The effect of these two thresholds has been tested by executing the Grid workload on top of the simulated provider using our policy, which is the one that makes a more aggressive consolidation. This allows evaluating the influence of the turning on/off thresholds by showing the client satisfaction and the energy consumption respectively.

![Figure 4.6: Energy consumption using different turn on/off thresholds](image)

Figure 4.6 shows that waiting the nodes to reach a high utilization before adding new nodes (high $\lambda_{\text{max}}$) makes the energy consumption smaller. In
the same manner, the earlier the system shuts down a machine (high $\lambda_{\text{min}}$),
the smaller the energy consumption is. It demonstrates how turning on and
off machines in a dynamic way can be used to dramatically increase the
energy efficiency in a consolidated datacenter.

On the other hand, client satisfaction decreases, as shown in Figure 4.7,
when the turn on/off mechanism is more aggressive and it shuts down more
machines (to increase energy efficiency). Therefore, this is a tradeoff between
the fulfillment of the SLAs and the reduction of the energy consumption,
whose resolution will eventually depend on the provider's interests. For
instance, if the provider is having a high client satisfaction, it could decide
to reduce it slightly to provide more energy efficiency (by shutting down
more nodes). In addition, this experiment has shown the capability of the
presented scheduling policy to consolidate the load and reduce the amount
of used nodes. Fortunately, average threshold values give a balanced tradeoff
between energy and QoS.

![Figure 4.7: Client satisfaction using different turn on/off thresholds](image)

4.6.3 Scheduling policy performance

Presented approach is different from others because in addition to consoli-
dation, it considers virtualization management. Thus, it improves the pro-
vider’s benefit by reducing this overhead and it is also aware of economical
issues. This can be seen by submitting our workload to a provider with dif-
ferent schedulers and compare their results (Table 4.3). This provider is
configured to have 65 nodes as it is the maximum amount of nodes needed
in the workload peak load.
Table 4.3: Scheduling results of policies

<table>
<thead>
<tr>
<th>Policy</th>
<th>Work/ON</th>
<th>Energy (kWh)</th>
<th>DL (%)</th>
<th>P (%)</th>
<th>B (€)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perfect</td>
<td>22.5 / 22.5</td>
<td>1229.5</td>
<td>100.0</td>
<td>100.0</td>
<td>1391.021</td>
</tr>
<tr>
<td>RD</td>
<td>22.0 / 23.9</td>
<td>1328.9</td>
<td>9.7</td>
<td>69.3</td>
<td>654.4</td>
</tr>
<tr>
<td>RR</td>
<td>28.1 / 30.1</td>
<td>1556.3</td>
<td>45.4</td>
<td>82.5</td>
<td>1030.5</td>
</tr>
<tr>
<td>BF</td>
<td>25.1 / 27.1</td>
<td>1495.7</td>
<td>59.8</td>
<td>84.2</td>
<td>948.3</td>
</tr>
<tr>
<td>BF+M</td>
<td>24.2 / 26.2</td>
<td>1420.0</td>
<td>61.5</td>
<td>84.3</td>
<td>967.8</td>
</tr>
<tr>
<td>Score</td>
<td>24.7 / 26.7</td>
<td>1464.6</td>
<td>66.9</td>
<td>84.4</td>
<td>1053.2</td>
</tr>
<tr>
<td>CDSP no SLA</td>
<td>24.0 / 26.0</td>
<td>1413.8</td>
<td>59.8</td>
<td>84.2</td>
<td>965.8</td>
</tr>
<tr>
<td>CDSP no mig</td>
<td>13.9 / 16.0</td>
<td>831.4</td>
<td>77.5</td>
<td>82.1</td>
<td>1401.7</td>
</tr>
<tr>
<td>CDSP</td>
<td>13.3 / 15.3</td>
<td>838.0</td>
<td>79.5</td>
<td>81.9</td>
<td>1412.3</td>
</tr>
</tbody>
</table>

Firstly, we evaluate an ideal scheduling which fulfills all the SLAs and uses the minimum optimal number of resources to fulfill all the SLA (Perfect policy). This approach gives us a reference of what is the best value we can get with this testbed and this workload. It consumes 1229.6 kWh and obtains a 100% SLA fulfillment which implies a benefit of 1391€. Notice that this policy is not realistic as it has a complexity NP but it gives a theoretical upper boundary.

After finding an upper boundary, we test some simple approaches like Random (RD) and Round Robin (RR) policies which neither take care of consolidation issues nor performance. As it is expected, the first one performs very badly in terms of SLA as it can be seen in the fulfillment of Deadline (DL) and Performance (P), and the second one gets a very bad consolidation level requiring a mean of 28 nodes working and 30 nodes on (Work/ON). These policies establish a lower boundary.

We introduce then a more plausible approach, which is consolidation-aware like backfilling (BF). It fills nodes with not assigned resources with small VMs to increase consolidation. In addition, the dynamic extension of this approach (BF+M), which supports migration of VMs to improve consolidation. However, both techniques cannot consider the overheads of managing virtualization and thus, they get a medium satisfaction level.

Another good policy to compare is the one just based in scores which does not consider economic factors (Score). Thanks to this score-based model, we are able to improve the satisfaction but this must manually deal with the different tradeoffs between performance and consolidation, which makes increasing the benefit (B) harder.

On the other hand, our scheduling policy considers multiple variants.
The first one, which does not consider SLA ($CDSP$ no SLA), gets results similar to the score-based policy. Adding SLA costs ($CDSP$ no mig), it is able to increase provider consolidation while getting better SLA fulfillment. Finally, the extension of our policy which supports migration ($CDSP$) gets a better consolidation and gets the higher benefit.

These results demonstrate our model is able to handle virtualization overheads and thanks to this, improve the provider benefit. In particular, it gets a 30\% more benefit than simple policies like backfilling and its variants. It is also better than the Perfect policy as it sacrifices some SLAs to dramatically reduce the energy consumption. The low energy consumption shown in Figure 4.8 is mainly because of a higher consolidation.

![Power consumption comparative](image)

**Figure 4.8:** Power consumption comparative

In addition, it increases the SLA fulfillment which reduces the SLA penalties. This can be seen in Figure 4.9 which shows our approach gets most of the tasks with a big SLA fulfillment while $BF+M$ has many tasks with less than a 0.5 fulfillment (50\%).

![SLA fulfillment values](image)

(a) Backfilling + Migration  
(b) CDSP

**Figure 4.9:** Frequency histogram of SLA fulfillment
4.6.4 Resource heterogeneity

An advantage of the proposed scheduling policy is able to take advantage of nodes heterogeneity. To evaluate this, we run the scheduling policy on providers with the same amount of resources (260 CPUs) but with different architectures:

**Xeon** 65 Xeon machines (Table 4.1 and used in the previous experiment).

**Atom** 130 Atom-based nodes (130 × 2 CPUs).

**Heterogeneous** 45 Xeon nodes and 40 Atom-based (45 × 4 + 40 × 2 CPUs).

Atom nodes require less space and reduce the capital costs, as two of them fit in the same space of one Xeon-based node and cost the half. Xeon machines are able to run 4 HPC tasks of 1 CPU in 1 hour while Atom machine can only run 2 in 5 hours. Therefore, the execution of 4 tasks in Xeon will take 1 hour and will consume 318 Watts while the other one would take 10 hours and will consume 398 Watts. Hence, it is more energy-efficient to run the HPC tasks in a Xeon machine and power it off.

Table 4.4 shows the performance of the different scheduling policies and Figure 4.10 shows the power consumption of our scheduling policy for the three datacenter setups.

<table>
<thead>
<tr>
<th></th>
<th>Nodes</th>
<th>Work/ON</th>
<th>Energy (kWh)</th>
<th>DL (%)</th>
<th>P (%)</th>
<th>B (€)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BF</td>
<td>Xeon</td>
<td>25.1 / 27.1</td>
<td>1495.7</td>
<td>59.8</td>
<td>84.2</td>
<td>948.3</td>
</tr>
<tr>
<td>BF</td>
<td>Atom</td>
<td>52.7 / 54.7</td>
<td>784.5</td>
<td>0.0</td>
<td>83.6</td>
<td>745.4</td>
</tr>
<tr>
<td>BF</td>
<td>Heterogeneous</td>
<td>38.1 / 40.1</td>
<td>1037.7</td>
<td>52.2</td>
<td>83.7</td>
<td>819.6</td>
</tr>
<tr>
<td>CDSP</td>
<td>Xeon</td>
<td>13.3 / 15.3</td>
<td>838.0</td>
<td>79.5</td>
<td>81.9</td>
<td>1412.3</td>
</tr>
<tr>
<td>CDSP</td>
<td>Atom</td>
<td>57.5 / 59.7</td>
<td>805.7</td>
<td>0.0</td>
<td>88.9</td>
<td>731.1</td>
</tr>
<tr>
<td>CDSP</td>
<td>Heterogeneous</td>
<td>13.3 / 15.3</td>
<td>647.2</td>
<td>72.2</td>
<td>82.2</td>
<td>1486.8</td>
</tr>
</tbody>
</table>

Table 4.4: Scheduling results of policies with heterogeneity

These results show a better economical performance for the heterogeneous approach. This is because it is able to reduce the power consumption thanks to the Atom power-efficiency and gets a reasonable SLA fulfillment thanks to the Xeon nodes.

On the one hand, our proposal run most of the HPC tasks in the Xeon nodes because they get a better performance. Moreover, these tasks have a deadline which cannot be achieved by Atom hosts.
Figure 4.10: Energy consumption using different provider configurations

On the other hand, in the case of applications with lower performance requirements such as web-applications or tasks with more relaxed SLAs, it is better to make use of Atom processors which have much more efficient power consumption. For example, the SLA fulfillment of web-applications \((P)\) is very similar for Atom and Xeon provider. However, the Atom datacenter violates all the deadline \((DL)\).

Finally, we have seen that resources features have an important effect in provider profitability. This experiment has shown that our proposal automatically balances workload among nodes according to power consumption and performance. Thus, we can conclude that our model is able to handle resources heterogeneity and gets a higher benefit for the provider.

### 4.6.5 Fault tolerance

This section demonstrates the ability of the presented scheduling policy to handle crashes. The experiment simulates a real environment where nodes crash with a given probability. In particular, nodes crash one or two times during the test week, getting a 99.98% and 99.99% uptime respectively.

<table>
<thead>
<tr>
<th></th>
<th>Work/ON</th>
<th>Energy (kWh)</th>
<th>DL (%)</th>
<th>P (%)</th>
<th>B (€)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BF+M</td>
<td>23.3 / 25.3</td>
<td>894.1</td>
<td>61.3</td>
<td>91.7</td>
<td>508.59</td>
</tr>
<tr>
<td>CDSP</td>
<td>16.1 / 18.6</td>
<td>1320.6</td>
<td>77.3</td>
<td>82.4</td>
<td>1324.32</td>
</tr>
<tr>
<td>FT CDSP</td>
<td>15.3 / 16.2</td>
<td>1232.7</td>
<td>77.3</td>
<td>81.3</td>
<td>1385.52</td>
</tr>
</tbody>
</table>

Table 4.5: Scheduling results of policies with a faulty environment

Table 4.5 compares the behavior of BF+M, the simple economic policy (CDSP), and an extended version of this policy which takes into account fault tolerance and performs checkpoint management (FT CDSP). This ex-
experiment demonstrates $BF+M$ is not able to deal with failures and loses many applications which makes it losing their the revenue. $CDSP$ is able to resubmit failed applications, but it gets low benefit as SLA fulfillment is compromised. Finally, $FT\ CDSP$ is able to recover long executions from a previous checkpoint and gets better SLA fulfillment. Thanks to this policy, provider can improve its benefits in a faulty environment.

4.7 Related work

**Energy efficiency.** It has been widely studied in the classical IT resource management area, and in a very different set of scenarios. For example, it has been already addressed in cluster-based systems by Petrucci et al. [95].

There are several works proposing energy management for servers which focus on applying energy optimization techniques in multiprocessor environments, such as the works of Bianchini et al. [96] and Chen et al. [97]. In particular, Chen’s work states that new power saving policies, such as Dynamic Voltage/Frequency Scaling (DVFS) or turning off idle servers [89], can increase hardware problems as well as the problem to meet SLAs in reduced environments. The latter is solved by adding a smarter scheduling policy to dynamically turn off idle machines to reduce the overall consumption. This technique is also applied by Kamitsos et al. [98], which sets unused hosts in a low consumption state to save energy.

The tradeoff between performance and power has also been tackled in local hosts where Chun et al. [99] propose a hybrid architecture which combines the selective usage of processors with different power consumptions and performances in a single host to apply local energy saving policies only when performance allows it. However, this idea has been also used for datacenters as presented in by Nathuji et al. [100] which state a good approach for saving energy is mixing low power systems and high performance ones in the same datacenter. Following this idea, our approach is able to take profit of this type of environment.

**Virtualization management.** Maximizing providers’ benefit has become a hot research topic thanks to the usage of the virtualization technology. During the last years, some works like Vogels [101] have studied the consolidation advantages using virtualization while other works, like the ones from Nathuji et al. [102] have widely explored its advantages from the power effi-
ciency point of view. Newer work from Petrucci et al. [103] presents the use of virtualization for consolidation. They propose a dynamic configuration approach for power optimization and outlines an algorithm to dynamically manage them. Their approach also focuses on a power efficiency strategy and consider the cost of turning on or off servers.

Discussed approaches can lead to a too slow decision process as it is focused in a very static environment and highly focused on HPC jobs. In this sense, our proposal is suitable for an on-line scheduler and manages both HPC jobs and Web-based services. Besides, Kumar et al. [104] propose a solution to facilitate coordination of power and virtualization management. Their approach provides a better VM placement and runtime management to achieve power savings (10%), as well as a significantly improved fulfillment of SLAs (i.e. 71% less violations).

To enhance VM placement, Liu et al. [17] aim to reduce virtualized provider power consumption by supporting VM migration and VM placement optimization while reducing the human intervention. Following this same idea, we propose the use of VMs for executing those afore-mentioned heterogeneous applications taking into account virtualization overheads. However, in addition of consolidating, virtualization makes the overall system more complicated and requires well-designed policies which take VM management problem into account. Until today, virtualization management overheads have been only considered individually when managing virtual resources [105, 106].

Economical approaches are also used to manage shared server resources. For example, Chase et al. [107] present a greedy resource allocation algorithm that distributes a web workload among multiple servers assigned to each service. This technique reduces the server energy usage by 29% or more for a typical Web workload. In addition, Chen et al. [108] propose and integrated management of application performance, dynamic workload migration and consolidation, power and cooling in providers and they present a prototype and demonstrate their integrated solution can reduce energy consumption of servers by 35% and cooling by 15%.

**Failure tolerance.** We state a methodology that tackles different challenges like fault tolerance. Fu [109] addresses this problem using virtualization and designing a reconfigurable distributed virtual machine (RDVM) infrastructure. Despite that work is focused on failure management, it also
uses a similar node selection approach which takes into account nodes reliability and tasks deadlines. Nevertheless, their approach is not focused on the aggregation of other costs such as virtualization overheads.

Summarizing, previous research only takes into account individual factors, while our contribution provides integrated and holistic solution to manage virtualized providers. Thus, we address all the important factors for managing a virtualized provider such as virtualization overheads, node reliability, outsourcing to third-party IaaS providers.

4.8 Conclusions

Towards making providers more profitable, energy consumption is a critical issue for large-scale providers, which hold hundreds or thousands of machines and cooling equipment. Virtualization is making providers more profitable every day as it provides consolidation and dynamism. Nevertheless, it requires new policies for managing the new virtualization capabilities.

We have created a mathematical apparatus to describe a virtualized provider and solves the management problem from an economic point of view. This model merges several factors, such as hardware and software requirements, SLAs, virtualization overhead, and power efficiency. Using this model, we use a simple optimization process to maximize the benefit of the provider. Finally, based on the optimization results we place VMs.

We use a power-aware simulator which allows us to compare our proposal with other plausible solutions. By using a heterogeneous workload, several metrics like power consumption and SLA fulfillment have been compared and analyzed. Experiments have demonstrated the presented model is able to increase the provider benefit by a 30% and can deal with problems like resource heterogeneity. Ultimately, it has also shown that for this kind of environment, the most energy-efficient approach is running jobs very fast and then powering the system off.

Finally, we can conclude that thanks to our policy we achieve multifaceted provider management, which is able to increase provider profitability.
Chapter 5

Managing federated virtualized providers

5.1 Introduction

One of the biggest challenges when managing a provider is its loads varies over time: there are peaks during daytime, and valleys during nights and weekends (Figure 5.1).

On the one hand, if the provider wants to provide every single request, it must support the peaks. In this situation, the provider is over provisioned and its resources would be underutilized during long periods. Operational costs can be reduced during these underutilized periods by turning idle resources off [89, 110]. However, this technique does not reduce capital costs related to the purchase and hosting of IT equipment, which needs to be amortized.

Figure 5.1: Provider workload over time
On the other hand, owning just enough resources to provide the average number of users or under provisioning reduces underutilization and saves infrastructure costs. Nevertheless, if a provider has not enough local resources to fulfill its customers’ requirements, it have to start canceling or denying the acceptance of new customers. This has further implications than just losing the revenue from some services, because it also implies a loss of reputation and therefore, a loss of future customers [20, 111].

The scheduling policy presented in Chapter 4 hindered the market potential of a provider by considering a limited amount of resources. This limitation can be overcome via federation [21].

In a federated scenario, multiple providers running services that have complementary resource requirements over time can mutually collaborate to share their respective resources to fulfill each one’s demand. For example, a provider can run a VM locally, decide to start running VMs in other providers, or turn idle nodes off to save power. Thus, it is difficult to perform resource provisioning when federation is used.

To support the management of a provider in this new scenario, we present an analytical model that characterizes a federated environment and drives provider’s decisions [32]. In this model, we analyze the impact of federation as a mechanism for maximizing provider’s benefit. Thus, we study the effect of these decisions on the provider’s profit and we evaluate the most appropriate provider’s configuration depending on the environment conditions.

To demonstrate the advantages of federation, we propose a scheduler able to support these capabilities and we evaluate its profitability using information of actual providers. This Provider Federation wraps multiple providers and it is supported by the Virtual Machine Scheduler as it is highlighted in Figure 5.2.

---

**Figure 5.2: Logic architecture of a virtualized provider: provider federation**
5.2 Analyzing profitability in actual providers

Actual providers that offer IaaS already deal with the load variability and the low profitability it implies. In this section, we study an actual IaaS provider, and estimate its basic profitability parameters. We have chosen RackSpace Cloud [112] because they provide detailed information of their specifications and we can make realistic assumptions about their infrastructure.

They use quad-core machines with 16 to 32 GB memory and 10 to 620 GB of disk to host the VMs. Using their standard VMs with 1 GB memory and 40 GB of disk costs $0.06, we can calculate they can host up to 16 VMs per host. Hence, they have a potential benefit of $2.4 per hour per machine.

Regarding the infrastructure, they lease their datacenters from DuPont Fabros [113]. This company provides information of some of their datacenters. In this case we will evaluate their largest datacenter, which is also one of largest datacenters in the US: ACC4. This is located in the Ashburn Corporate Center campus (ACC) in Northern Virginia. It comprises approximately 348,000 gross square feet with 171,000 raised square feet and 36.4MW of critical load.

To calculate the costs, we get the most relevant information for Ashburn location: the energy costs $0.0673 per KWh [114], the land costs $95 per square foot [115], and the maximum temperature is 26°C [116].

Using this information and the datacenter costs and models from [2] and [117], we can calculate the datacenter details. We assume that a datacenter with a maximum outside temperature of 26°C will have a maximum PUE of 1.6. Hence, the datacenter would have around 23MW to power IT equipment. In addition, a quad core machine as the ones RackSpace is using has a maximum power consumption of less than 300W. From this information, we can infer this datacenter can host around 80 thousand machines.

Using a datacenter cost of $12 per W [2], building the datacenter costs 436.8 million dollars. The cost of the whole land will be around 3.3 million dollars. Amortizing them over 12 years, building is 3 million per month and the land is 230K dollars per month.

At this point, we calculate the cost for each one of the hosts based in a Dell PowerEdge 2970 with the specifications provided by RackSpace, which costs around $4000 dollars (amortizing over 4 years is $85 per month). Assuming the datacenter hosts 80K servers and distributing the other costs among them, each server has an associated capital cost of $103 every month. On
the other hand, operating and maintaining this datacenter has a typical cost of $0.05 per Watt per month, which is a total of $23 per machine.

Using the highest and lowest ranges for each cost, the error in these estimations is around 10%. Nevertheless, it is accurate enough for our purpose of giving a broad view of the profitability of the provider.

If we assume the machines are fully used all the time, each one consumes 216 KWh each month and 345KWh taking into account the PUE, which implies an energy cost of $23 every month. Hence, a fully used machine would have a total cost of $173 and would generate an income of $1728.

Nevertheless, literature suggests the utilization of servers is around 30% [2]. Servers at this utilization consume around 200W which implies $16 per month. This implies a total cost of $165 per month, and an income of $518 per server. In fact, with a 5% use, a host would imply a cost of $123 and an income of $90 per month and thus, the provider would lose money.

**High prices vs. federation: two roads toward profitability.** Providers with large infrastructures are proportionally cheaper than small ones [2]. In addition, they have big margins in the VM pricing and thus, they can afford low utilization.

If actual providers would use a lower pricing with lower benefit margins, they could increase their competitiveness by offering the same service at lower rates. For instance, RackSpace Cloud offers a pricing of $0.06 per GB of memory and uses a linear conversion to get the other pricings, e.g. 8GB of memory costs $0.048 but the cost of maintaining an average load of 30% is $165. Applying the formulation for a single provider, with an average load of 30% they would be profitable with a price of $0.048 per GB.

In a tighter scenario with lower VM prices, they would be more competitive but their profitability would be lower. To overcome this low profitability, they could take advantage of federation capabilities and their profitability would increase thanks to a more efficient usage of the resources.

Applying federation, the workloads of multiple federated providers could consolidate their loads and avoid the purchase and maintenance of resources.

### 5.3 Federation in Cloud providers

Federation in virtualized providers is possible thanks to virtualization and Cloud computing (i.e. IaaS). The first one allows running applications with-
out taking care of the underlying hardware or the setup of the external provider [4]. The second one makes the access to other provider resources easier as it offers computing resources in a pay-as-you-go manner. Taking this into account, we can talk about federation of Cloud providers.

In this Cloud computing scenario, we can exploit both Private and Public Clouds [20]. By Private Clouds we mean, a private infrastructure dedicated to one organization with a limited capacity. We refer as Public Cloud the utility computing available to the general public in a pay-as-you-go manner.

Once services can be provided by external resources, a provider can outsource to other providers when its workload cannot be attended with its local resources (peak hours). In this way, the provider would obtain higher profit because it can attend more customers and does not lose reputation.

Similarly, a provider that has underused resources can rent part of them to other providers, which we refer as insourcing resources. In this way, the provider increases its benefit, exploits better its resources, and compensates the maintenance cost. Again, the expected benefit from renting its resources should be higher than the cost of maintaining them running. Otherwise, it would be preferable to turn them off to save power (and thus reduce costs).

From previous discussion, one could realize that the profitability of using Cloud federation for a service provider highly depends on a number of parameters, such as the provider’s incoming workload, the cost of outsourcing additional resources, the revenue for renting unused resources, or the cost of maintaining the provider’s resources operative. All these parameters must be considered to decide the most adequate resource management action for the provider depending on current (and foreseen) environment conditions. Depending on their value, the provider could decide at every moment whether to outsource, insource, or turn resources off.

To exploit federation capabilities, we require resource management mechanisms that can dynamically manage both internal and external resources in the most cost-effective way while satisfying the QoS agreed with the users.

5.3.1 Federated scheduler

To bring federation in IaaS providers, we envision a global scheduling layer on each provider. This is able to interact with other federated providers and exchange load according to the provider requirements. In addition, as other common schedulers, it decides the placement of the VMs in local resources.
To achieve this, we develop an implementation of the *Virtual Machine Scheduler* layer which supports *Provider Federation* and is referred as *Federated Scheduler* (FEDS).

*FEDS* is divided in two different parts: the management of the local resources and the management of external resources using Cloud federation. The scheduling policies for managing local resources are already presented in Chapter 4. Notice that this includes migrating, pausing, or even canceling tasks if the cost of these actions is compensated with higher utility for the provider. When it manages local resources, *FEDS* can also decide to shut down provider’s unused nodes to save power.

Regarding external resources, *FEDS* is able to allocate additional resources from a federated Cloud provider or insource load from other federated providers when internal resources are unused. Outsourcing to other federated Cloud providers is performed by adding the other providers as internal resources where VM can be executed.

For instance, Figure 5.3 shows an example situation with three Federated Cloud Providers (FCP) which are running VMs. In this example, *FCP 2* is running 2 VMs and it is already full. There is a third request to run a new one, *VM 3*, but *FCP 2* is not able to run it as it does not have enough resources. Nevertheless, *FCP 1* has free resources and offers them to *FCP 2*. At this point, *FCP 2* decides to outsource *VM 3* execution to *FCP 1*.

To decide the placement of the VMs, *FEDS* uses the characterization presented in next section. This characterization calculate the foreseen profit of every allocation and applies the one with higher profit.
5.4 Characterizing a federated Cloud

To obtain the maximum benefit from Cloud federation, and given the described complexity of federation decisions, it is important that the provider has a clear understanding of the potential of each federation decision. Using this, the provider can take the most convenient decision depending on the environment conditions.

To take the decisions (e.g. outsourcing a VM to another provider, running it locally), \textit{FEDS} uses this model to estimate the profitability of every action. This model evaluates the provider utilization (number of total and used nodes), the pricing of the VMs, the capital costs (CAPEX), and the operational costs (OPEX) [20].

With all this information, this model uses the characterization of a federated environment to estimate the profitability of each situation. Using this model, \textit{FEDS} picks the most profitable situation and performs the next scheduling actions:

- Run VMs using local resources.
- Turn on stopped nodes.
- Turn off idle nodes.
- Run VMs in a federated provider (outsource).
- Offer idle resources to other federated providers (insource).

Next sections present the characterization of a federated provider according to its capabilities: (1) only with local resources, (2) using external providers, (3) offering idle resources to external providers, or (4) using all the federation capabilities.

5.4.1 Allocation within the provider

In Chapter 4, we have presented a scheduling policy to consolidate VMs in fewer machines to be able to turn off machines and save energy. From this point of view, the provider consolidates the maximum number of VMs in a single node to maximize its usage. This allows applying different techniques for reducing the power consumption of the provider, such as Dynamic Voltage/Frequency Scaling (DVFS) and node shut down [118, 119].

In this chapter, we model this behavior from a higher level perspective leaving the internals of the provider scheduling to the policy presented in
Chapter 4. We focus on deciding the number of running physical machines considering the number of VMs in the provider.

Following the same philosophy used in the previous chapter, the expected profit for the provider derives the allocation decisions of the scheduler. We define the profit obtained from executing tasks in a provider \( p \) during in a certain period \( \Delta t \) as \( \text{Profit}_p(\Delta t) = \text{Revenue}_p(\Delta t) - \text{Cost}_p(\Delta t) \). As in this scenario we are only considering the nodes of this single provider, its total profit \( \text{Profit}(\Delta t) \) is equal to \( \text{Profit}_p(\Delta t) \).

\( \text{Revenue}_p(\Delta t) \) is obtained by multiplying the number of VMs running in the provider during that period of time \( \text{VM}_p(\Delta t) \) with its corresponding price (e.g. \( \text{Price}_\text{VM}_\text{Hour} \) of a small instance in Amazon EC2 is \( €0.085 \) per hour). Notice \( \text{VM}_p(\Delta t) \) depends on the provider’s incoming workload.

\[
\text{Revenue}_p(\Delta t) = \text{VM}_p(\Delta t) \cdot \text{Price}_\text{VM}_\text{Hour} \cdot \Delta t \quad (5.1)
\]

\( \text{Cost}_p(\Delta t) \) is defined as the cost of maintaining all the nodes in the provider up \( (\text{Nodes}_p \cdot \text{Cost}_\text{Node}_\text{Hour}_{\text{var}}) \) during a certain period \( \Delta t \). In addition, since shutting down idle nodes would reduce the costs for the provider, we add a factor to the formula \( (C_p(\Delta t)) \), which indicates the capacity of the system (understood as the ratio of nodes that are up), to reflect this. If all the nodes in the system are up, capacity is 1. If the provider shuts down half of the nodes, capacity is 0.5. Finally, we add also some fixed costs per node \( (\text{Cost}_\text{Node}_\text{Hour}_{\text{fix}}) \), which include the costs of acquiring the nodes and the physical space they occupy, taking into account their amortization.

\[
\text{Cost}_p(\Delta t) = C_p(\Delta t) \cdot \text{Nodes}_p \cdot \text{Cost}_\text{Node}_\text{Hour}_{\text{var}} \cdot \Delta t \\
+ \text{Nodes}_p \cdot \text{Cost}_\text{Node}_\text{Hour}_{\text{fix}} \cdot \Delta t 
\quad (5.2)
\]

To normalize the provider’s incoming workload (i.e. the number of VMs to be executed), we define the provider’s utilization \( (U_p(\Delta t)) \). It is calculated using as reference the maximum number of VMs that the provider can host, which is given by multiplying the number of nodes \( (\text{Nodes}_p) \) and the number of average number of VMs per node \( (\text{VM}_\text{Node}) \), in the following way:

\[
U_p(\Delta t) = \frac{\text{VM}_p(\Delta t)}{\text{Nodes}_p \cdot \text{VM}_\text{Node}} 
\quad (5.3)
\]
As discussed before, a single provider is profitable when $Revenue_p(\Delta t) > Cost_p(\Delta t)$. Using previous equations and operating on this formula, we obtain Equation 5.4, which establishes the relationship between the utilization (i.e. the amount of VMs to execute) and the capacity (i.e. the ratio of nodes that are operative) for provider’s profitability. Obviously, this and the subsequent equations require $C_p(\Delta t)$ to be greater or equal than $U_p(\Delta t)$. This equation will allow FEDS to determine the number of nodes to shut down ($C_p(\Delta t)$) given the current workload ($U_p(\Delta t)$) to get the best profit.

$$C_p(\Delta t) < \frac{U_p(\Delta t) \cdot VM\_Node \cdot Price\_VM\_Hour - Cost\_Node\_Hour_{fix}}{Cost\_Node\_Hour_{var}}$$  \hspace{1cm} (5.4)

### 5.4.2 Outsourcing to federated Clouds

As described by Armbrust et al. [20], outsourcing resources to federated providers is preferable over provision a private datacenter for peaks. In addition, it allows provider to insource idle resources to other providers if these are not being used. These authors also introduce an equation that evaluates whether outsourcing resources to an external provider is profitable or not. It essentially compares the profit (resulting from $Revenue - Cost$) when outsourcing to external resources against executing in its own resources.

Our analysis starts from this formula to decide grabbing additional resources when there is a resource demand that cannot be fulfilled using local resources. In particular, the additional revenue obtained when outsourcing resources $Revenue_o(\Delta t)$, which is shown in Equation 5.5, is calculated in the same way as Equation 5.1 and depends on the number of VMs that are outsourced ($VM_o(\Delta t)$). Notice that, in this scenario, the total revenue for the provider is $Revenue(\Delta t) = Revenue_p(\Delta t) + Revenue_o(\Delta t)$.

$$Revenue_o(\Delta t) = VM_o(\Delta t) \cdot Price\_VM\_Hour \cdot \Delta t$$  \hspace{1cm} (5.5)

The total cost for the provider in this scenario is $Cost(\Delta t) = Cost_o(\Delta t) + Cost_p(\Delta t)$. The cost of outsourcing ($Cost_o(\Delta t)$) could be calculated also from Equation 5.5. However, we assume that the provider can buy these VMs cheaper than the revenue it obtains for selling them. According to this,
we apply a factor $\alpha$ to the cost of the VM, obtaining Equation 5.6.

$$\text{Cost}_o(\Delta t) = \text{VM}_o(\Delta t) \cdot \alpha \cdot \text{Price\_VM\_Hour} \cdot \Delta t \quad (5.6)$$

In this scenario, the provider is profitable when $\text{Revenue}_p(\Delta t) + \text{Revenue}_o(\Delta t)$ is bigger than $\text{Cost}_p(\Delta t) + \text{Cost}_o(\Delta t)$. Notice that $U(\Delta t) = U_p(\Delta t) + U_o(\Delta t)$ includes both the VMs executed in the provider and those outsourced to other providers. According to this, we define $U_{ratio}(\Delta t) = \frac{U_p(\Delta t)}{U(\Delta t)}$, which represents the ratio of incoming workload that is executed locally in the provider. Using previous equations, we can derive Equation 5.7, which allows to determine the number of nodes to shut down ($C_p(\Delta t)$) and the distribution of local and outsourced VMs ($U_{ratio}(\Delta t)$) to get the best profit.

$$C_p(\Delta t) < \frac{U(\Delta t) \cdot (1 - \alpha + U_{ratio}(\Delta t) \cdot \alpha) \cdot \text{VM\_Node} \cdot \text{Price\_VM\_Hour} - \text{Cost\_Node\_Hour\_fix}}{\text{Cost\_Node\_Hour\_var}} \quad (5.7)$$

### 5.4.3 Insourcing from federated Clouds

As commented before, in a federated environment the provider can offer its unused resources to other providers (i.e. insourcing). In this case, the total cost for the provider does not vary ($\text{Cost}(\Delta t) = \text{Cost}_p(\Delta t)$). This means that there are not additional costs if the provider rents its free resources. However, the total revenue is expected to increase ($\text{Revenue}(\Delta t) = \text{Revenue}_p(\Delta t) + \text{Revenue}_i(\Delta t)$). For calculating $\text{Revenue}_i(\Delta t)$, we use again the $\alpha$ factor (offering idle resources cheaper) and we include another factor ($\beta$) that represents the ratio of free resources that can be offered. This serves to model the market demand of resources, since not all the resources can be always sold to external providers. This factor also allows the provider to reserve some free resources to able to react to variations in its workload. According to this, the number of potential VMs that could be sold is:

$$\text{VM}_{free} = (C_p(\Delta t) - U_p(\Delta t)) \cdot \text{Nodes} \cdot \text{VM\_Node} \quad (5.8)$$

Using this parameter, $\text{Revenue}_i(\Delta t)$ can be calculated using Equation 5.9.

$$\text{Revenue}_i(\Delta t) = \beta \cdot \text{VM}_{free}(\Delta t) \cdot \alpha \cdot \text{Price\_VM\_Hour} \cdot \Delta t \quad (5.9)$$
Having the option to offer unused resources to other providers, or to shut down them to reduce power consumption, the provider could doubt on which is the more profitable decision. The answer comes from resolving the following inequation: $\text{Revenue}_p(\Delta t) + \text{Revenue}_i(\Delta t) > \text{Cost}_p(\Delta t)$. Again, using previous equations and operating on this formula, we obtain Equation 5.10, which allows to determine the number of nodes to shut down ($C_p(\Delta t)$) and the ratio of free resource to insource ($\beta$) given the current workload ($U_p(\Delta t)$) to get the best profit.

$$C_p(\Delta t) < \frac{U_p(\Delta t) \cdot (1 - \alpha \cdot \beta) \cdot \text{VM Node} \cdot \text{Price VM Hour} - \text{Cost Node Hour fix}}{\text{Cost Node Hour var} - \alpha \cdot \beta \cdot \text{VM Node} \cdot \text{Price VM Hour}}$$

\section*{5.4.4 Insourcing and outsourcing in federated Clouds}

The final step is putting all together. In this case, profitability occurs when $\text{Revenue}_p(\Delta t) + \text{Revenue}_o(\Delta t) + \text{Revenue}_i(\Delta t) > \text{Cost}_p(\Delta t) + \text{Cost}_o(\Delta t)$. After operating on this formula, we obtain Equation 5.11. Using this equation, \textit{FEDS} can decide whether outsourcing resources, renting free resources to others providers, or shutting down nodes is profitable for the provider.

$$C_p(\Delta t) < \frac{U(\Delta t) \cdot (1 - \alpha + U\text{ratio}(\Delta t) \cdot \alpha \cdot (1 - \beta)) \cdot \text{Price VM Hour} \cdot \text{VM Node} - \text{Cost Node Hour fix}}{\text{Cost Node Hour var} - \alpha \cdot \beta \cdot \text{VM Node} \cdot \text{Price VM Hour}}$$

\section*{5.5 Implementing a federated provider}

We have already presented which are the basic components of a provider to support federation, and the characterization of the provider’s profitability in this environment. In this section, we present some other considerations that must be taken into account when federating different providers.

\subsection*{5.5.1 Capacity planning}

When a provider wants to offer its service, it must decide the amount of resources it will need to provide it. This depends on the federation capabilities it will support (i.e. outsourcing, insourcing, none...) and the expected
number of users. To estimate this expected number of users, the provider can use former workloads and apply business models to predict their growth.

Using the characterization and the expected workload, the provider decides the amount of resources (i.e. number of nodes) required. In addition, with this information the provider fixes the VM pricing scheme.

5.5.2 Federated resource management

Assuming that the provider supports both insourcing and outsourcing, and VM pricing previously fixed for public use, it will use Equation 5.11 to decide how to manage its resources. As a reminder, the final equation is generic and works for the other scenarios.

**FEDS** calculates the profitability of the provider using this equation and the current utilization. According to the utilization of the provider, it can be full and it would require external resources to run more VMs or it can have not used resources. In the case the provider has no free resources; it can use the equation to:

- Cancel the execution of new or already running VMs.
- Outsource VMs to other federated providers with a given pricing ($\alpha$).

Otherwise, if the provider has idle resources, it uses the equation to:

- Shut down part of the idle resources ($\beta$ parameter).
- Offer part of the idle resources to federated providers and set a pricing for federated providers ($\alpha$).

Every time the utilization of the provider or the policies change, the scheduler must take the required actions.

5.5.3 Extending local scheduling policy

In this section, we show how we can extend the scheduler presented in Chapter 4 to support outsourcing. Firstly, we add support to use of rented resources from other providers. To model this, a renting cost is introduced and it is expressed as $C_{\text{rent}}(h_p, vm)$, where $h_p$ is the external provider. This cost typically depends on the amount of time the VM will be executed and its features. For example, Amazon EC2 in the US specifies a cost of 0.08€ per hour for small instances, which corresponds to a VM of 1.7 GB of memory,
1 EC2 Compute Unit (1 virtual core with 1 EC2 Compute Unit), 160 GB of storage, and 32-bit platform [9].

\[ C_{rent}(h_p, vm) = Pr_{hour}(h_p, Type(vm)) \cdot (T_d(vm) + T_{extra}(vm)) \]

In addition, there are no SLA penalties as an static resource allocation that fulfills the SLA is performed in the external provider. Moreover, there is no power consumption costs to be considered \( C_{pwr}(h_p, vm) = 0 \) because executing a VM into an external provider does not incur any energy cost for the local provider.

Finally, in terms of virtualization penalties, there are two special cases. Firstly, possible migrations of a given \( vm \) from and to those special hosts are avoided \( (T_m(h, vm) = T_m(h_p, vm) = \infty) \). Secondly, concurrency penalty \( (C_{conc}(h_p, vm) = 0) \) is not considered in an external provider, so \( C_{virt}(h_p, vm) \) is equal to the cost of creating a VM in the external host, \( T_c(h_p, vm) \).

5.5.4 Interconnecting providers

To communicate between multiple providers, Amazon EC2 is becoming the standard de facto. Nevertheless, there exist multiple alternatives (e.g. OCCI [120]) each of them with different features.

To solve this problem, we propose the usage of a wrapper (i.e. \( VtME \) component) that implements all the required APIs and offers a single interface to \( FEDS \). The current implementation already supports EC2 and OCCI APIs. It offers a transparent way to operate with other providers as they were large resources with special features.

Regarding the semantic differences of each provider, the component in charge of this is the \( VtME \). For example, the standard VMs in Amazon EC2 are categorized in: small, large, and extra large; while RackSpace distributes them according to the memory size. Therefore, \( VtME \) must deal with the heterogeneity when defining the VM size.

5.5.5 Resource availability

A provider is able to know the status of its own resources and evaluate if it has enough room to host a new VM. In a federated environment, a provider should be also able to know the availability of the other providers to host a given VM. For this reason, when a provider is federated it must provide the
Chapter 5. Managing federated virtualized providers

5.5.6 Service Level Agreements

Providers typically offer a given SLA to their customers [28], but not every provider supports the same SLAs. Hence, provider must perform a previous triage to discard those providers with incompatible SLAs.

Every federated provider must monitor continuously the resource usage of the tasks it is executing and the fulfillment of their SLAs. If any SLA violation is detected, an adaptation process for requesting more resources to the provider is started, first locally in each node, and then globally in the provider as presented in Chapter 3. If the violation cannot be solved in the provider, this situation must be forwarded to the client. In case it is a VM outsourced by another provider, the client will be a federated provider. This mechanism would allow the other party evaluating the situation and taking the required actions.

5.6 Experimental environment

We evaluate the proposed characterization by studying the potential benefit in a service provider when doing outsourcing and insourcing and shutting down nodes by means of the equations described in this chapter.

We follow the pricing idea for Grids presented by Opitz et al. [121]. We use reference values for revenues, costs, and virtualization parameters just to demonstrate how our equations can drive resource allocation decisions, though the particular values of these parameters will highly depend on the real provider’s characteristics. Anyway, the presented equations remain valid. As base node, we use a mid-range server with a direct consumption of 638 W in mean [122, 123]. These mid-range servers support in mean a maximum amount of 6 VMs per node, assuming small EC2 instances, which have a cost of 0.085 €/hour (EC2 pricing in Europe).

Nevertheless, power consumption is not just the server direct consumption. It must also take into account all the related energy costs such as
cooling and other infrastructure consumptions. This is evaluated using the Power Usage Effectiveness (PUE), which is defined as the ratio of datacenter power to IT power draw [124]. According to historical trends, site infrastructure consumes 50% of all datacenter energy, which corresponds to a PUE of 2.0 (this means that the datacenter must draw 2 Watts for every 1 Watt of power consumed by the IT equipment). Therefore, we assume an average consumption of 1276 W per node. The pricing used for the electricity is the Spanish one, which corresponds to 0.09 €/KWh [92].

Finally, to calculate the cost of the nodes, we also take into account the amortization of the servers (in 3 years) and the space (in 10 years) required to deploy them using a price of 4000 €/node and 2000 €/m², respectively.

5.7 Evaluation

In this section, we analyze the profitability of a provider using the different federation scenarios previously presented. Then, we evaluate the extension to support outsourcing in the scheduling policy presented in Chapter 4. Finally, we evaluate the potential benefit of a Cloud provider that takes advantage of federation, including two different types of providers, namely small and large providers.

5.7.1 Profitability analysis in a federated Cloud

Taking into account previous information, we first calculate how a service provider should be dimensioned (i.e. its capacity) according to its utilization applying Equation 5.4. The results are displayed in Figure 5.4, showing the capacity as a function of the price of the VMs and the provider’s utilization. The two dimension figure shows the profitability for a price 0.09 €/per VM (darker is more profitable, outside of the drawn area results are not possible or profitable). In particular, high capacities are only profitable when the utilization is greater than 40% and the price per VM is higher than 0.05 €. Being below these values, the provider is not profitable since the fix costs are too high.

Figure 5.5 shows the maximum capacity a provider should have to be profitable when using outsourcing according to Equation 5.7. This figure relates the capacity with the provider’s global utilization (i.e. the incoming workload) and the ratio of VMs that are locally executed, assuming that
Figure 5.4: Relation between utilization (%) and capacity (%) in a single provider (Equation 5.4). Darker is more profitable.

\( \alpha = 0.75 \). The two dimension figure shows the profitability for an utilization of 80\% (darker is more profitable, outside of the drawn area results are not possible or profitable). As shown in the figure, the higher the workload and the number of VMs locally executed are, the higher capacities are allowed.

Figure 5.5: Relation between utilization (%) and capacity (%) using outsourcing (Equation 5.7). Darker is more profitable.

Figure 5.6 shows the maximum capacity a provider should have to be profitable if it is able to insource resources according to Equation 5.10. It relates the capacity with the provider’s utilization and the factor \( \beta \) (the ratio
of free VMs that it sells), assuming an $\alpha$ factor of 0.75. If the provider sells less than the 30% of its free resources, high capacities are only profitable when the utilization is greater than 50%. On the other side, higher values of $\beta$ and small utilizations allows also high capacities since the surplus resources can be sold to other providers. Figure 5.7 shows the profitability when offering the 20 and the 80% of the idle resources (darker is more profitable, outside of the drawn area results are not possible or profitable).

![Figure 5.6: Relation between utilization (%) and capacity (%) when introducing insourcing (Equation 5.10)](image)

(a) Offering 20% of idle resources  (b) Offering 80% of idle resources

![Figure 5.7: Relation between utilization (%) and capacity (%) when introducing insourcing (Equation 5.10). Darker is more profitable.](image)

Finally, using Equation 5.11, which models the provider taking into account both outsourcing and insourcing, we get Figure 5.8, which relates the capacity with the utilization and the ratio of local VMs assuming $\alpha = 0.75$
and $\beta = 0.5$. It shows that having low provider’s utilization (because the workload is low or it is being mostly outsourced) allows higher capacities since free VMs can be insourced.

![Figure 5.8: Relation between utilization (%) and capacity (%) when introducing insourcing and outsourcing (Equation 5.11)](image)

5.7.2 Extending local scheduling

Evaluation of Chapter 4 has shown that a regular provider has big fixed costs of more than 350 € for maintaining a datacenter with 65 nodes during a week, while not all of them are used all the time. To tackle this high cost, the scheduling policy takes advantage of outsourcing to withstand periods of high load. In this experiment we use the same setup used in Section 4.6.3 and we add the ability to outsource to two external providers. In addition, we have used the presented model to get the optimal number of nodes according to our workload and this is 20 nodes.

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Out/€</th>
<th>Work/ON</th>
<th>Power (kW)</th>
<th>$DL$ (%)</th>
<th>$P$ (%)</th>
<th>$B$ (€)</th>
</tr>
</thead>
<tbody>
<tr>
<td>65</td>
<td>0</td>
<td>0.0</td>
<td>13.3 / 15.3</td>
<td>838.0</td>
<td>79.5</td>
<td>81.9</td>
</tr>
<tr>
<td>65</td>
<td>50</td>
<td>26.5</td>
<td>13.1 / 15.0</td>
<td>817.3</td>
<td>82.9</td>
<td>83.9</td>
</tr>
<tr>
<td>20</td>
<td>0</td>
<td>0.0</td>
<td>14.2 / 15.5</td>
<td>850.9</td>
<td>30.1</td>
<td>83.7</td>
</tr>
<tr>
<td>20</td>
<td>992</td>
<td>89.4</td>
<td>13.2 / 14.9</td>
<td>800.3</td>
<td>59.8</td>
<td>86.3</td>
</tr>
<tr>
<td>0</td>
<td>2038</td>
<td>1345.7</td>
<td>0.0 / 0.0</td>
<td>0.0</td>
<td>49.1</td>
<td>99.7</td>
</tr>
</tbody>
</table>

Table 5.1: Scheduling results of policies introducing outsourcing

Table 5.1 shows that the difference between using outsourcing or not when having enough resources for the whole workload (65 nodes) is not very big as it only outsources 50 VMs, which implies an outsourcing cost of
26.5 €. Using outsourcing, it reduces the power consumption less than a 1% and does not improve the fulfillment too much.

On the other hand, when the provider has only 20 nodes to execute the whole workload, it highly reduces fixed costs and power consumption but it gets bigger SLA penalties. Nevertheless, it gets bigger benefit when using outsourcing as it is able to reduce the amount of local resources and rent external resources to external providers during peak loads. Having only 20 nodes implies a cost of 89.4 € for renting 992 VMs. Reducing the amount of local resources and renting makes the provider getting bigger benefit.

It is worth recalling that cost of outsourcing depends on the duration of the VMs which have been rented and this is why 50 VMs can cost 26.5 € and 992 VMs only 89.4 €. In addition, performance in an external provider is better since it is assumed the provider gives the required amount of resources while it only adds the creation cost of 300 seconds for EC2, which implies a big overhead for short deadline tasks.

Finally, it might seem that reducing the amount of local resources will make the provider having better performance. Nevertheless, the option of having a provider with no local resources gets a poor benefit since the outsourcing costs are not compensated with the power saving or SLA penalties.

5.7.3 Potential benefit in a federated Cloud

In this section, we apply the formulas presented in Section 5.3 to calculate the potential benefits and costs of a service provider during a week with a real workload. This workload, which is shown in Figure 5.9, is extracted from a provider’s log during the week from Monday 27th of April until Monday 4th of May 2009.

The provider has a cluster with 100 nodes. However, as shown in Figure 5.9, sometimes the customers’ demand is higher. Using a traditional resource management approach, the provider has to reject all the services that exceed its maximum capacity. Therefore, it loses many clients during rush hours. These lost clients can represent a great amount of money that is being wasted. In addition, the reputation of this provider is going down since customers stop trusting on it. For this reason, outsourcing resources to external providers can increase the provider’s capacity when it is not enough to satisfy the demand. On the other side, the provider’s capacity is underused during some periods. This reduces its total profit, since under-
used nodes are also consuming power. To avoid this, the provider can shut down those machines it guesses will not be required during a long period, for instance, during night. This decreases the power consumption during that period. Alternatively, the provider can also offer these unused resources to other providers, so they can execute their services on them. This option will be profitable for the provider when the obtained revenue is enough to compensate the cost of maintaining all these nodes up.

Of course, turning off nodes, outsourcing, and insourcing can be jointly applied to maximize the provider’s profit. In this section, we evaluate the impact on the profit when using these techniques. The results, which assume $\alpha = 0.75$, $\beta = 0.5$, are displayed in Table 5.2 and demonstrate the benefit of outsourcing resources, which is inversely proportional with the $\alpha$ factor. In addition, outsourcing allows the provider maintaining its reputation by being always available to give service to its customers. The second part of the table presents the profit in case the provider decided to increase its maximum capacity (up to 200 nodes) to support the whole workload without using outsourcing. It shows that the revenue has increased regarding the previous table, but also the fixed costs, such as hardware and maintenance. For this reason, global profit is lower in this case. In fact, only the Insource-Nodes always up configuration is profitable in this case.

The values in these tables are graphically represented in Figures 5.10(a) and 5.10(b). The benefit from shutting down nodes can be clearly appreciated in the ‘Typical’ and the ‘Outsourcing’ configurations in Figure 5.10(a).
127 Related work

<table>
<thead>
<tr>
<th></th>
<th>Nodes always up</th>
<th>Shutting down nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Revenue</td>
<td>Cost</td>
</tr>
<tr>
<td>100 nodes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Typical</td>
<td>5997.60</td>
<td>5185.84</td>
</tr>
<tr>
<td>Outsource</td>
<td>7270.56</td>
<td>6140.56</td>
</tr>
<tr>
<td>Insource</td>
<td>8843.40</td>
<td>5185.84</td>
</tr>
<tr>
<td>Out &amp; Insource</td>
<td>10116.36</td>
<td>6140.56</td>
</tr>
<tr>
<td>200 nodes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Typical</td>
<td>7270.56</td>
<td>10371.68</td>
</tr>
<tr>
<td>Insource</td>
<td>11888.1</td>
<td>10371.68</td>
</tr>
</tbody>
</table>

Table 5.2: Service provider’s profit in Euros

The same argument applies in the ‘Typical’ configuration in Figure 5.10(b). In the configurations with ‘Insourcing’ in Figure 5.10(a), it is more profitable not to shut down nodes, because the provider has more resources to offer to other providers. This also applies to the ‘Insourcing’ configuration in Figure 5.10(b), though in this case the profit is noticeably smaller due to the increased fixed costs.

![Figure 5.10](image.png)

(a) 100 nodes  
(b) 200 nodes

Figure 5.10: Comparison of provider’s profit with different capacities

5.8 Related work

Cloud computing has rapidly spread in recent times. Armbrust et al. [20] present some key concepts of this paradigm such as the illusion of infinite computing resources available on demand and the ability to pay for use of computing resources on a short-term basis as needed. This allows companies
to have a small set of resources that can be increased according to their needs, saving costs. Jha et al. [125] establish how Clouds can be viewed as a logical continuation from Grids by providing a higher-level of abstraction. Buyya et al. [126] define Cloud computing and provides the architecture for creating Clouds with market-oriented resource allocation by leveraging technologies such as VMs. It also proposes ideas for interconnecting Clouds for dynamically creating global Cloud exchanges and markets and presents some representative Cloud platforms.

Distributing load of a provider among different datacenters has been studied by Le et al. [127]. Particularly, the authors propose the usage of different datacenters according to its geographical distribution and the power consumption at each moment. However, this is applied to resources of the same provider which can be controlled by the same entity.

The idea of federating systems was already present in the Grid. For instance, Boghosian et al. [128] and Sobolewski et al. [129] use federation to get more resources in a distributed Grid environment. The application of federation in the Cloud was initially proposed within the Reservoir project. In particular, Rochwerger et al. [21] describes the difficulty to merge different providers with different APIs and features. Nevertheless, they do not present any model to decide when to move tasks to a federated provider based on economic criteria. A first approach to this idea was introduced by Campbell et al. [130] where they state some factors such as provider occupation and maintaining costs to dimension a Cloud provider and when to outsource to a federated provider.

One of the strong points of Cloud federation is the possibility of incorporating public Clouds within the federation. There are a number of Cloud offerings that provide VMs on demand, being Amazon EC2 [9] probably the most popular. Nevertheless, it is a private implementation and it does not allow working with low-level aspects. Other public Cloud solutions with similar capabilities are GoGrid [131], 3Tera [132], and ElasticHosts [133].

To set up private Clouds with the same capabilities than public Clouds, different Cloud solutions that implement the EC2 API, such as Eucalyptus [134] or Nimbus [135], have appeared. Similarly, Aneka [136] is a .NET-based service-oriented resource management platform, which is based on the creation of containers that host the services and it is in charge of initializing services and act as a single point for interaction with the rest of the Aneka
Moreover, it provides SLA support such that the user can specify QoS requirements such as deadline and budget. Other open-source alternatives, such as AbiCloud [137], and OpenNebula [44], also add outsourcing capabilities by adding external resources.

These works consider basically economical factors over power elements, looking for the improvement of the infrastructure owner’s economical profit and revenue. None of these works consider the option of outsourcing and insourcing resources.

5.9 Conclusions

In this chapter, we have studied how to implement federation in Cloud providers to increase their profitability by saving capital and operational costs. Studying actual IaaS providers, we can see that they have a huge profit margin as they sell their resources with a much more expensive price than the actual costs for maintaining them. In this way, they can easily deal with underutilization because of valley hours. If providers would start using federation capabilities, they could start reducing their prices and getting more clients while tackling the load variability.

To support federation in virtualized providers, we have described and implemented a scheduler, $FEDS$, which is able to operate with external providers. In addition to the managing of local resources, this scheduler is also able to decide when to use other providers resources or when to offer them the own idle resources. This is performed using a characterization of Cloud federation aimed at enhancing providers’ profit. Our characterization includes equations that assist decisions in a federated Cloud, namely when to outsource resources to other providers, when to insource free resources to other providers, and when to shut down unused nodes to save power.

Our experimentation has evaluated these equations with actual data to determine the impact of some parameters in the providers’ profit. Evaluated parameters include the provider’s incoming workload, the cost of outsourcing additional resources, the ratio of outsourced resources, the ratio of unused resources to be sold, and the cost of operating the provider’s resources.

Our results show the provider requires a minimum utilization and a minimum price per VM to be profitable when all the nodes are running. Moreover, local resources are preferred over outsourced resources, though
the latter can enhance the provider’s profit when the workload cannot be provided locally. Furthermore, when the utilization is low, the best option is insourcing the unused resources (though this is not always possible). We can summarize that all the described actuations can have a positive impact on the provider’s profit depending on the environment conditions. Finally, we have studied the profitability of an actual IaaS provider and studied in which situation it could take advantage of federation capabilities.
Chapter 6

Conclusions

We have seen how the introduction of virtualization in providers has changed the way they managed their own infrastructure and the kind of services they provide. For example, they started offering rental of virtual machines to the customers resulting in the IaaS paradigm. Furthermore, virtualization has also opened new chances for resource management. This metamorphosis in the providers has made them unable to be managed as traditional ones anymore as they pose new challenges and new capabilities to exploit.

Decentralized virtualization management. Despite the fact virtualization offers new capabilities, the way they were currently designed was too centralized. For this reason, we have introduced a fabric to support and exploit these capabilities. Firstly, we have proposed a system able to distribute the creation process. This on-demand VM creation system reduces the required storage and delegates the creation load to the nodes.

We have developed a data management system for virtualized service providers that provide data to VMs and enables live-migration. In our approach, each node in the provider shares his storage and others can access when migration occurs. This approach is fully distributed what increases performance and reliability, and reduces bottlenecks.

The last virtualization capability enhanced has been the checkpointing. We have proposed a smart checkpoint infrastructure that saves time required to checkpoint and store a VM while enhancing a distributed and fault tolerant infrastructure.

Increase provider utilization. The resources management of VMs in a host has also been enhanced to exploit the ability for fine-grain dynamic
Chapter 6. Conclusions

resource distribution provided by virtualization. The system implements a self-adaptive behavior driven by the SLA agreed for each VM. Nevertheless, current SLAs hinder the dynamic allocation of resources. To overcome this limitation, we have proposed a resource-level SLA metric for specifying fine-grain QoS guarantees regarding the computing capacity of the provider.

Following our proposal, a better resource utilization in the provider is achieved while each application receives enough resources to meet the agreed QoS. The evaluation of this resource manager demonstrates the functionality of our system and how it is able to adapt to SLA violations.

Multifaceted provider management. The scheduling of VMs among hosts of a provider has been the most important part and the key to enhance provider’s profitability. Understanding and taking advantage of the proposed virtualization fabric, providers can exploit new mechanisms.

We have proposed a new scheduling policy which mainly focuses on the allocation of VMs in datacenter nodes according to multiple facets while keeping in ming the overhead of managing virtualization. It is able to merge several factors, such as hardware and software requirements, SLAs, or power efficiency. We have created a mathematical apparatus to describe this VM scheduling from an economical point of view. Based on this model, we are able to increase the provider benefit by a 30% regarding basic policies.

Tackle load variability by means of federation. One of the big problems of providers this time is the high variability of load over time. On the other hand, peak hours are crowded and they need lots of resources to serve all the demand. However, during valley hours much of these resources are under-utilized. We avoid this variability problem by federating multiple providers with different workloads. This is possible thanks to the easiness to renting resources from other IaaS providers.

We exploit the federation capabilities and characterize this scenario to create policies that take the most convenient decision depending on the environment conditions. Our characterization includes equations that assist decisions in a federated Cloud, namely when to outsource resources to other providers, when to insource free resources to other providers, and when to shutdown unused nodes to save power.

Our experimentation has evaluated these equations with realistic data to determine the impact of some parameters in the providers’ profit. We
can summarize that all the described actuations can have a positive impact on the provider’s profit depending on the environment conditions. We have also studied the profitability of an actual IaaS provider and studied in which situation it could take advantage of federation capabilities.

The main and final conclusion after tackling the problem of managing the resources of a virtualized provider is that a traditional one can highly and easily improve its performance by exploiting virtualization. Nevertheless, for achieving this improvement, providers must remodel the way they manage their resources. Thus, they can deal efficiently with them, exploit all their potential, and solve previously unsolvable problems.

6.1 Future work

In this thesis we have covered the main aspects of the management of a virtualized provider. Porting these innovations into actual providers would bring huge benefits to them. From a research perspective, it has also opened a large number of challenges to be tackled.

The main difficulty for managing VM resources is the provider does not know which applications are being executed. To avoid this limitation, there exist approaches which are able to detect which types of applications are running. Thanks to this knowledge, provider would be able to make a more efficient management of its resources.

Another issue to be improved is the management of the SLAs. Firstly, we have not discussed how these SLAs are negotiated and this procedure can provide valuable information. To start tackling this problem, we have proposed a metric that permits a more efficient exploitation of the resources. Nevertheless, higher level SLAs metrics like throughput and response time are required to specify the requirements of web services. The main problem of these high-level metrics is the mapping between them and the required physical resources. For doing these mapping, it is required new approaches to manage SLAs which tackle this from a business perspective.

Performing a good scheduling of VMs highly depends on the knowledge of the system. However, the problem is the provider does not know what is going to happen next. To overcome this limitation, prediction of workloads can be used. Thanks to this technique, the provider is ready to support the future load and make a scheduling keeping this in mind. In addition to
the usage of future data, presented policy can be extended to support new ways to manage aspects like energy or reliability. For instance, regarding the energetic aspect, current providers start considering the usage of renewable energies. This brings new challenges like how to schedule the workload when dealing with the variability of green sources of energy.

Current federation capabilities have been tested in an analytical way or a limited environment. It would be necessary to build a simulated environment that mimics the behavior of multiple providers with different workloads. Using this simulator, we can develop new policies that would focus in low-level aspects.

Finally, analyzing the profitability of a provider, one can observe that the costs highly depend on the location of the datacenter. For example, areas with cheap land may have cheap energy but expensive water. To tackle this problem, the placement of a datacenter can be done reducing the costs of building and operating a datacenter, and thus, increase provider’s benefit.
Appendix A

Architecture for virtualized providers

Nowadays, service providers offer complex services ready to be used, as it was a commodity like water or electricity, to their customers. A key technology for this approach is virtualization which makes provider’s management easier and provides on-demand virtual environments, which are isolated and consolidated to achieve a better utilization of the provider’s resources. Moreover, it allows agile and fine-grain dynamic resource provisioning by offering mechanisms to carefully control how and when the resources are used.

However, dealing with some virtualization capabilities, such as the creation of virtual environments, implies a big effort to fully exploit them. To avoid this problem, there are multiple middlewares that make the management of virtualized environments easier. Some examples are OpenNebula [44] and Eucalyptus [134] which are open source and allow instantiating VM on demand and transparently manage the provider resources. Among the multiple interfaces they offer, they implement one which is compatible with Amazon EC2. Thanks to this interface, private resources can be used as they were EC2 resources.

Nevertheless, these implementations are focused in production environments and they add complexity which makes research harder. For example, they have to deal with keys and certificates to provide security and their architecture is not suitable for distributed solutions.

We are contributing the research community with the EMOTIVE (Elastic Management of Tasks in Virtualized Environments) middleware [138].
This is a virtualized environment manager which aims to provide VMs that fulfills with the user requirements in terms of software and system capabilities. It also supports fine-grained resource management and provides enhanced mechanisms, such as migration or checkpointing, for developing scheduling policies. Thanks to its capabilities and its flexible architecture, it is easy to extend which make it suitable for our research purposes.

It provides resource management at several layers, namely locally to each node, among nodes of the same provider, and among multiple providers. Next sections present the core of the proposed middleware in a detailed way and give the required background for the rest of the dissertation.

A.1 Related work

The first steps to use virtualization in providers were taken in SoftUDC [35]. This is a virtual machine monitor that let applications share physical resources while maintaining full functional isolation.

Abadala et al. proposed the In-VIGO project [139] which defined a three layer architecture to virtualize grid resources. This architecture enables the creation of dynamic pools of virtual resources that can be aggregated on demand for grid-computing. This was one of the first proposals of an architecture for provisioning virtualized resources. Nevertheless, this is a generic architecture which just defines the layers without giving details of its implementation. Our proposal follows a similar three layer architecture.

At the same time, paradigms like Grid computing started moving to a virtualized approach. This happened with the Globus Virtual Workspace [11] which refocus the Globus Toolkit by using virtual environments. This approach schedules virtual environments among multiple nodes and uses virtualization features such as pausing and migration.

At this point, Amazon started offering EC2 [9], where users can rent VMs for a period of time. Nevertheless, this is a closed implementation based on Xen to support virtualization but the underlying architecture is unknown. To overcome the problems of a closed infrastructure, Nurmi et al. proposed an open implementation of an IaaS provider, which emulated the EC2 API, called Eucalyptus [134]. However, this implementation suffered of one of the problems of Grid computing: it was very difficult to setup because of its security. This was the case of The Globus Toolkit [140],
which security required to perform a manual and laborious setup. For this reason, Eucalyptus was very promising but it has not been widely used.

Another IaaS implementation is OpenNebula [44]. This software is easier to setup and allows the addition of modifications, which make it suitable for research [141, 142]. In addition, it is intended for production scenarios which require a safe and fixed architecture with a single point of entrance. On the contrary, we propose a widely distributed architecture where every node is almost autonomous. Furthermore, it is easy to modify and extend without taking into account production issues like security.

A.2 Architecture

Typical implementations of middlewares for managing virtualized providers have a very common architecture, where there are multiple worker nodes and a global scheduler in charge of coordinating these workers. This architecture can be easily divided in multiple layers. Our architecture for virtualized providers is based on this layered model, as it is shown in Figure A.1.

![Virtualized provider architecture](image)

Figure A.1: Virtualized provider architecture

Our architecture is composed by three different layers: Virtualization Fabrics, Virtual Machine Manager, and Virtual Machine Scheduler. The Virtualization Fabrics layer comprises the physical resources where the VM will run. The Virtual Machine Manager layer comprises all the local resource management decisions (i.e., in a single node). This layer is in charge of man-
aging the physical resources of a node and distributing these resources among all the VMs running on that node. Finally, the Virtual Machine Scheduler layer comprises all the global resource management decisions, both among different providers and different nodes in a single provider. This layer is in charge of deciding where a VM will be executed.

To give a better understanding of the proposed architecture, we will start explaining how virtualization works. Following sections describe in detail the three layers of our architecture. This is done in an incremental way going from the basement of the architecture to the top layer.

A.3 Virtualization technology

Our proposals rely on the usage of virtualized environments to enhance infrastructure management for providers. In this section we present the most relevant aspects of virtualization technology and we discuss the technologies that are going to be mainly used for this dissertation.

Virtualization is a technology that is causing a major transformation of the IT infrastructures in the coming years [12]. Among all the virtualization technologies, those which give the desirable level of performance for working as a platform for a provider are full-virtualization and paravirtualization.

On the one hand, full-virtualization, also known as native virtualization, uses a VM that mediates between guest operating system and the native hardware. The Virtual Machine simulates enough hardware to allow an unmodified operating system run on top. Certain machine instructions must be trapped and handled within the hypervisor because the underlying hardware is not owned by an operating system but instead, it is shared by it through the hypervisor. It has the disadvantage of being slower than the native hardware but does not require any modification of the guest OS. There are multiple alternatives for this technique like VirtualBox [143], Microsoft 2008 Hyper-V [144], and VMWare [145].

On the other hand, paravirtualization is similar to full-virtualization as it also uses a hypervisor but integrates some virtualization parts into the operating system. It is born with the need to increase full virtualization performance and explores ways to provide high performance virtualization of x86 by implementing a VM that differs from the raw hardware. Hence, guest operating systems are ported to run on the resulting VM.
To implement this method, hypervisor offers an API to be used by the
guest OS. This call is called “hypercall”. This issue increases the perfor-
mance with regard to full virtualization. Figure A.2 shows how full virtuali-
ization offers the same interface to the VM of the underlying hardware while
paravirtualization offers a modified interface.

However, thanks to these modifications it offers a performance close to
the one in an unvirtualized system. Some of the most famous examples of
paravirtualization are Xen [66] and KVM [146].

A.3.1 Implementation issues

VMs execute software in the same manner as the machine for which the
software was developed. The VM is implemented as a combination of a real
machine and virtualizing software and implementations issues depend on
the virtualization technique, nevertheless, the main part of them follow the
same philosophy more or less.

Typically virtualization is done by a layer that manages guest petitions
(processor demand, memory or input/output) and translates them into the
underlying hardware (or to the underlying operating system in some cases)
making them executable.

A typical implementation decision in emulation and full virtualized envi-
ronment is separating an executed code between privileged and non-privileged
for performance reasons. This decision is based on the principle that code is
executed in different ring levels and VMs are typically in the non-privileged
layer and it demands a special control for the privileged instructions.

Processor. Emulating instructions executed by the underlying processor is
the key feature of different virtualization implementations. The main task of
the emulator is to convert instructions which is done with interpretation or
binary translation. Then, it executes this code in the underlying machine.
Nevertheless, current architectures like IA-32 is not efficiently virtualizable because it does not distinguish between privileged and non-privileged instructions implying that every instruction must be identified. Some improvements in newest processors to avoid this problem are tackled using the Hardware support which will be detailed in Section A.3.1.

Meanwhile, a typical operating system uses a scheduling algorithm that determines which processes will be executed in which processor and how long, in a virtualized environment, virtualization layer must take the decisions of which VM will be executed.

**Memory.** Operating system assigns memory pages among processes with a page table that assigns real memory among processes running on the system. Virtual machine monitors uses this host operating capabilities to map memory to each process.

To implement memory sharing between VMs there are several ways, but every method maps guest application memory into the host application address space, including the whole VM memory. This mapping can be done in a more software way or relying this decision on the hardware depending on the virtualization method.

Paging requests are converted into disk read/writes by the guest OS (as they would be on a real machine) and they are translated and executed by the virtualization layer. With this technique, standard memory management and replacement policies are still the same than in a non-virtualized machine.

**Input/Output.** Operating system provides an interface to access I/O devices. These accesses can be seen as a service that is invoked as a system call which transfers control to the operating system. It uses an interface to a set of software routines that converts generic hardware requests into specific commands to hardware devices and this is done through device driver calls.

Implementing Input and Output typically only store the I/O operation and pass it to the overlying system and then return it to the application converting petitions to system specific formats.

**Hardware support.** At the beginning, x86 architecture did not support virtualization and it made difficult to implement a virtualized environment on this architecture.

Virtualization software needs to employ sophisticated mechanisms to trap and virtualize some instructions. For example, some instructions do not trap and can return different results according to the level of privilege
mode. In addition, these mechanisms introduce some overhead.

A sign of how important virtualization has become is the addition of hardware capabilities that support it [147]. Main chip vendors, Intel and AMD, have introduced extensions to resolve these difficulties. They have independently developed virtualization extensions to the x86 architecture that allow a hypervisor to run an unmodified guest operating system without introducing emulation performance penalties.

This improvements are based on the inclusion of a special mode, VMX, that supports privileged and non-privileged operations and then any instruction can be easily executed without taking into account if it is privileged or not. Thanks to this hardware support, most of the paravirtualization products can act as a full-virtualizer.

A.3.2 Libvirt

Our proposals are valid for any virtualization technology, like KVM, VMWare or Xen. For example, we have tested dynamic creation with KVM and Xen and our live-migration approach is also supported by other hypervisors like VMWare. However, each technology has a different API and this makes it complex to work with all of them. To overcome this problem, libvirt [148] was developed. This is a library intended for a generic use of multiple virtualization technologies such as Xen, KVM, or VMWare.

This library allows accessing data from the virtualized platform in a generic way and it also allows operating all the hypervisors with a common API. For example, if we setup a machine with KVM and install libvirt on top, we can just use it as it was a machine with Xen or VMWare. This is what we do from the Resource Fabrics where we just use libvirt in a machine without taking care of the underlying technology.

Our proposals are independent of any virtualized technology. Nevertheless, we have used Xen to conduct the major part of experiments. For this reason we will give a detailed vision of this hypervisor and its internals.

A.3.3 Xen

This project [149] was created in 2003 by the computation laboratory of the University of Cambridge and it was known as the Xen Hypervisor project, leadered by Ian Pratt [66]. During the next years, the present Xen company was created: XenSource.
Many distributors such as Intel, AMD, Dell, Hewlett-Packard, IBM, Novell, Red Hat or Sun Microsystems use this software. In addition, it has a GPL license and can be freely downloaded.

The key of Xen success is paravirtualization that allows obtaining a high performance level. It gives to the guest operating system an idealized hardware layer. Intel has introduced some extensions in Xen to support the newest VT-X Vanderpool architecture. This technology allows running operating systems without any modification to support paravirtualization.

It provides mechanisms to manage resources, including CPU, memory and I/O. At that moment, this is the fastest and safest virtualization infrastructure. Nevertheless, paravirtualization requires introducing some changes in the virtualized operating system but resulting in near native performance.

In this environment a virtual server is just an operating system instance (called domain) and its load is being executed on top of the hypervisor. These instances access devices through the hypervisor, which shares resources with other virtualized OS and applications.

The Xen hypervisor is the lowest and most privileged layer. Above the hypervisor guest operating systems or VMs are allocated. It has its own notation for VMs, it refers them as domains and it is always a guest operating system that is booted when the hypervisor boots and have special management privileges, this domain is called “Domain-0”.

Overhead introduced by Xen hypervisor is less than 5% in CPU intensive applications [66]. Moreover, thanks to paravirtualization, I/O operations are executed out of the hypervisor and shared between domains following resource sharing policies [150]. However, these virtualized domains are fully isolated from each other. When the system supports Intel-VT or AMD Pacifica, operating systems without any modification like Windows can be run. Thanks to this new architecture and paravirtualization, an OS without modifications achieves virtualized Linux performance levels.

This hypervisor also offers some tools like live migration, CPU scheduling and memory management, which combined with open source software advantages make Xen a great alternative to other software. It also provides different tools and interfaces that enable accessing to the hypervisor values and modifying and monitoring them.

One of the most relevant interfaces for us of the hypervisor is XenStore, which provides a way for communicating VMs and easily extends its func-
tionalities. For instance, default memory monitoring does not take into account the real consumption of the guest system and just reports the initially allocated, for this reason and following the philosophy of VMWare tools, we have developed an extension that is located in the VM and informs about its resource usages using XenStore.

Taking advantage of these monitoring and managing tools, XenMonitor has been developed. It allows a full control of resources of each VM and monitors a large set of resources like, disk, network, CPU usage, memory...

A.4 Virtualization fabrics

The basements of the system are the resources which allow running VMs on top. To fulfill all the requirements of the system, multiple facilities are required (i.e., storage and networking). In addition, these facilities must make the system able to support migration and checkpointing, as they will be used by upper layers. This layer capabilities are detailed in Chapter 2.

Current implementation supports virtualization using libvirt [148]. To cover lacks in libvirt such as fine-grain management of resources like CPU or memory, which management highly depends on the virtualization technology, we have extended the standard API using the XenMonitor.

To provide users' with the data they require to execute their tasks, the provider requires a way to store data persistently. The system supports multiple ways to store this data and allows creating VM with input data or storing output results on this storage. In addition, it can also store the whole VM image to resume its execution in the future. Data storage is detailed in Section 2.3.

The Virtualization Fabrics layer also supports a way to access the VMs using a system that provides addressing and network naming. It uses a DHCP server that dynamically assigns an IP address to each VM and updates the local DNS server to provide human-friendly names. Thanks to this naming, the middleware can access the VM using a SSH wrapper, that allows the submission of tasks described in a JSDL file. This file describes the tasks and allows defining input data (using the data management support).

Data management is another challenge in virtualized providers. It must support an efficient access to VMs disks while enabling migration. This is achieved by making each node able to access its own local disk and the disk
of the other nodes. This allows each node creating VMs and executing tasks efficiently using the local disk. Furthermore, tasks can also be migrated with minimum overhead, since it is not necessary to transfer the VM image, while maintaining their accessibility during the whole process. Moreover, migrated tasks can access remotely their required data without noticeable performance penalty. Further details can be found in Section 2.4.

Finally, the Virtualization Fabrics supports fault-tolerance by adding VM checkpointing. It is also able to store these checkpoints taking into account two main factors. On the one hand, the checkpoint mechanism must be fault-tolerant and thus, any single point of failure must be eliminated. On the other hand, checkpoints can be concurrently recovered from multiple nodes to resume VM execution faster after a node crash. Both requirements are achieved by a checkpoint storage which distributes and replicates the checkpoints in all the nodes of the provider. The implementation and evaluation of this approach is presented in Section 2.5.

A.5 Virtual machine manager

The main responsibility of the Virtual Machine Manager layer is managing the VMs hosted in a node and distributes the resources among them. This task is carried out by the Virtualized Resource Management and Monitoring (VRMM), which wraps the physical machine and allows creating new customized VMs, executing tasks on these VMs, monitoring their execution, extracting output data, and destroying them. A single Virtualization Manager (VtM) per physical node is in charge of VM management and resource distribution, while monitoring support is provided by a Resource Monitor (RM). The proposed solutions for this layer and their evaluations can be found in Chapter 2 and Chapter 3.

A.5.1 Virtualization manager

The VtM is in charge of managing the life-cycle of the VMs and their resources during its execution. It creates a customized VM, according to the user requirements, manages its resources to ensure the QoS, and finally destroys the VM. The most complex task is the VM creation which implies copying the data and the images required by the VM and booting it up. This process can imply a high cost but we propose a new caching system
which allows reducing dramatically the time required to have a VM ready to be used. This mechanism is presented and evaluated in Section 2.2.

Another task of VtM is providing fault-tolerance to the upper layers. It periodically checkpoints the running VMs to be able to resume their execution if the node where they run crashes. To provide an efficient checkpoint mechanism, we use a new technique to store just the last changes in the VM. In addition, these checkpoints are stored in the storage provided by the Virtualization Fabrics. The proposed checkpointing mechanism is described and evaluated in Section 2.5.

Once the VM is running, VtM can submit tasks into the VMs using a SSH wrapper able to manage the task execution. While tasks run, VtM calculates the amount of resources a task really needs and allocates the required resources to each VM. Further details of all these resource management mechanisms are described in Section 3.2.

A.5.2 Resource monitor

Resource Monitor provides information of the resources to upper layers or external components. This information contains some dynamic metrics such as the CPU and memory usage or static ones like the architecture and the number of CPUs of each VM. The RM has a distributed architecture where there is a master that manages the monitoring of each resource. At the same time, each resource is monitored by a RM slave which uses the capabilities of Virtualization Fabric to obtain these metrics from libvirt. The master gets the information from the resources and provides a Ganglia-like XML file with this information.

However, RM not only provides information about the VMs but it is also in charge of resource discovery and topology maintaining. It periodically checks if new nodes has been added and informs the Virtual Machine Scheduler layer about this. In addition, it also provides failure management by periodically checking if resources are still available and notifying the upper layers if any of them breaks down.

A.5.3 External virtualization manager

Proposed architecture is also able to deal with external resources from federated providers. This functionality is provided by the Virtualization Manager External (VtME), which allows renting VMs in external providers such
as Amazon EC2 and deploying tasks within them as it was done locally. However, VtME provides less functionality than VtM because the control over external resources is lower than over local ones, which makes capabilities such as efficient migration or checkpointing not feasible. VtME also provides capabilities to offer unused resources to other providers. This component and its functionalities are used by federation models in Chapter 5.

### A.6 Virtual machine scheduler

The top layer in the EMOTIVE middleware is the Virtual Machine Scheduler. It is in charge of merging all the nodes and resources, and abstracting them as a single large resource. Using this approach, VMs run on top without taking into account if these VMs are running in a node or another.

This layer decides where and when to run the VMs according to multiple parameters and to do so, it uses the capabilities from bottom layers. For example, when it needs to provide a new one, it will use the fast creation mechanism. Another example is the efficient migration which can be used to make a more efficient usage of the resources. Moreover, the Virtual Machine Scheduler uses the resource discovery and monitoring capabilities from RM to maintain the topology of the provider’s resources.

Using these capabilities, the provider can schedule resources focusing in different requirements, such as fault-tolerance, economic benefit or energy consumption. Multiple implementations of this layer which use different policies are detailed in Chapter 4.

### A.7 Summary

We have presented a stack of software that permits implementing a Cloud solution which simultaneously ensures an efficient usage of the local and the remote resources. We propose an approach that exploits the advantages of virtualization for accomplishing resource distribution as well as it deals with application life cycle management.

The presented middleware offers a working platform which takes advantage of virtualization capabilities, and provides transparent management of the resources. This will be used as a framework to develop the contributions of this thesis and it is available on the project site [151].
Bibliography


BIBLIOGRAPHY


[33] GNU Debian. Linux debootstrap.


