THE UNIVERSITY OF CHICAGO

PROVISIONING COMPUTATIONAL RESOURCES USING VIRTUAL MACHINES AND LEASES

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY
BORJA SOTOMAYOR BASILIO

CHICAGO, ILLINOIS
AUGUST 2010

A mis padres,

Eduardo y Ana

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

Development of this dissertation was supported by the University of Chicago's Department of Computer Science, the Computation Institute of the University of Chicago and Argonne National Laboratory, the U.S. Department of Energy under Contract DE-AC02-06CH11357, the European Union through the RESERVOIR research grant (Grant Number 215605), and Spain's Ministerio de Ciencia e Innovacion, through the research grant TIN2009-07146. Early work on this dissertation was funded by NSF grant #509408 "Virtual Playgrounds".

* * *

What a journey this has been! A joyful, bizarre, amusing, exultant, strange, unexpected, cloudy, international, ineffable, harsh, passionate, rough, fabulous, vexatious, transformational, esoteric, instructive, embiggening, inscrutable, looney, serendipitious, abstruse, disheartening, heartbreaking, heartwarming, unpredictable, melancholy, sensational, cromulent, exciting, thrilling, stirring, scrumtrulescent, provocative, arduous, formidable, laborious, scientific, awesome, kickass, adventure-filled, and *fun* journey. And after so many emotions, at the end of it all there is mostly one: gratitude. I would not have made it here without the help, guidance, and support of many wonderful people. To all of them, I extend my most sincere and heartfelt thanks:

To Ian Foster, my PhD advisor. Looking back at the last six years, I am in awe of how profoundly and positively my life has been affected by knowing Ian and working with him. Besides steering my dissertation work to a successful conclusion, Ian has supported me tirelessly, standing up for me when I needed it most, and opened the door to more opportunities than I could possibly imagine when I first set foot on the University of Chicago.

To Anne Rogers, member of my PhD committee and Director of Graduate Studies at the Department of Computer Science. Sometimes, your life can change in one pivotal day.

One such day for me was January 18, 2008, when I was faced with a very difficult choice between two paths. Anne nudged me towards the choice that, at the time, went against every instinct in my body. Nonetheless, I opted to do as she suggested and, as it turns out, following her advice ended up sparking a chain of events that unequivocally brought me to where I am today and, more importantly, made me a fundamentally happier person. On top of this, Anne has also provided me with exquisitely detailed feedback on my work and many nuggets of sound advice laced with just the right amount of tough love.

To Ignacio Martín Llorente, member of my PhD committee and OpenNebula Project Lead, who has indefatigably supported and promoted my work for the last two years, giving it a home in the OpenNebula community, where it has grown beyond my expectations. Ignacio also gave me a chance to do an internship with the OpenNebula team in Madrid during the summers of 2008 and 2009, closer to home and surrounded by a truly kickass team of developers and researchers. Without a doubt, two of the best summers of my life.

To Rubén Santiago Montero, also an OpenNebula Project Lead, for the countless technical discussions that have so positively impacted my work.

To Tino Vázquez, Javier Fontan, and Jaime Melis, for making me feel part of a team during the summers of 2008 and 2009 and for being the most fun group of people to hang out with.

To Kate Keahey, who pioneered cloud computing before it was called cloud computing, and who first got me hooked on resource management with virtual machines.

To Mike Rainey, the first –and best– friend I made here in Chicago.

To Adam Shaw, my officemate for the last four years who, more often than not, manages to brighten my day with stimulating conversations and unadulterated hilarity.

To all the other CS PhD students –too many to list– who have shared this journey with me.

To Lisa Childers, for making me feel welcome in a strange city from day one, being a

confidante extraordinaire, and as awesome a friend as anyone could wish for.

To everyone on the Globus MUD, for all the whuggles, family muds, strokes, hello1's, rimshots, toasts, fireworks and general silliness over the years.

To Sharon Salveter, for allowing my teaching skills to grow, giving me a much needed escape valve whenever the research got too intense, and for always having an open door, a welcoming smile, and plenty of stuff to chit-chat about.

To all my University of Chicago students, for reminding me every day of how much I love to teach. Special shoutout to the ACM-ers and ICPC-ers: Ian Andrews, Jai Dhyani, Korei Klein, Michael Lucy, Cord Melton, Karl Norby, Matthew Steffen, Damon Wang, and Louis Wasserman.

To Rebeca Cortazar, my engineering advisor when I was a student at the University of Deusto, for suggesting years ago that I should maybe look into "that Grid Computing stuff that seems to be so hot right now".

To Javier García Zubía, for whom I RA'd at the University of Deusto and who first got me interested in academic research.

To the Department of Computer Science's Techstaff –Nick Russo, Tyler Bray, and Virgil Gheorghiu– for all their help in setting up and maintaining the department's Condor pool, which I used to run most of the experiments in this dissertation.

To Donna Brooms, Katie Casey, Margaret Jaffey, and Nita Yack, for all their administrative support and for helping to run a well-oiled department.

To the many friends who have enriched my life and, on many occasions, have been there for me when I needed them, particularly Borja Bacaicoa, Iñigo Calvo Sotomayor, Jim Campbell, Al Derus, Garikoitz Echebarría, Pablo "Txipi" Garaizar, Asier García, Jorge "Bardok" García, Ryan Hayes, Laura Plunkett, Eric Rogers, Isabel Romero, Josu Sauto, Samara Taber, and Anthony Todd.

A mis padres, Eduardo y Ana, por su inquebrantable apoyo e incombustible entusiasmo

a lo largo de toda esta travesía.

ABSTRACT

The need for computational resources has, over the years, become a fundamental requirement in both science and industry. In many cases, this need is transient: a user may only require computational resources for the duration of a well-defined task. For example, a scientist could require a large number of computers to run a simulation for just a few hours, but might not need those computers at any specific time (as long as they are made available in a reasonable amount of time). A college instructor may want to make a cluster of computers available to students during the course's lab sessions, at very specific times during the week, and with a specific software configuration. A telecommunications company could possess an existing infrastructure that hosts a number of websites, but may need to supplement that infrastructure with additional resources during periods of unforeseen increased web traffic, meaning those resources have to made available right away with very little advance notice.

These transient resource usage scenarios pose the problem of how to provision shared computational resources efficiently. This problem has been studied for decades, resulting in approaches that tend to be highly specialized to specific usage scenarios. For example, the problem of how to run multiple jobs on a shared cluster has been extensively studied, resulting in job management systems systems like Torque/Maui, Sun Grid Engine, LoadLeveler, and many others, that can queue and prioritize job requests efficiently (in these systems, efficiency is defined in terms of a variety of metrics, including waiting times and resource utilization). Such a system would meet the requirements of the scientist wanting to run simulations during a few hours but, on the other hand, the college instructor and the telecommunications company mentioned above would be ill-served by a job management system and the efficiency metrics typically used in job management. Conversely, other resource provisioning approaches are not particularly well suited for job-oriented computations.

Thus, there is no general solution that can provision resources meeting the requirements of different usage scenarios simultaneously, such as those mentioned above, reconciling the

different measures of efficiency in each scenario. More specifically, much of my work is motivated by the combination of best-effort resource requirements, where a user needs computational resources but is willing to wait for them (possibly setting a deadline), and advance reservation resource requirements, where the resources must be available at a specific time. In the former, efficiency is typically measured in terms of waiting times (or similar metrics such as turnaround times or slowdowns) or throughput, while the latter is usually concerned with providing the requested resources at exactly the agreed-upon times without interruption, and both are concerned with maximizing the use of hardware resources and possibly monetary profit. Although both best-effort and advance reservation provisioning have been studied separately, the combination of both is known to produce utilization problems and is discouraged in practice.

In this dissertation I develop a resource provisioning model and architecture that can support multiple resource provisioning scenarios efficiently and simultaneously, with an initial focus on the best-effort and advance-reservation cases mentioned above, and arguing in favour of a lease-based model, where leases are implemented as virtual machines (VMs). The main contributions of this dissertation are:

1. A resource provisioning model and architecture that uses leases as a fundamental abstraction and virtual machines as an implementation vehicle.

2. Lease scheduling algorithms that mitigate the utilization problems typically encountered when scheduling advance reservations.

3. A model for the various overheads involved in using virtual machines, and algorithms that (a) allow lease terms to be met even in the presence of this overhead, and (b) mitigate this overhead in some cases.

4. Price-based policies for lease admission, showing that an adaptive pricing strategy can, in some cases, generate more revenue than other baseline pricing strategies, but does

so by using fewer resources, thus giving resource providers more excess capacity that can potentially be sold to other users.

As a technological contribution, I also present Haizea (http://haizea.cs.uchicago.edu/), an open source reference implementation of the architecture and algorithms described in this dissertation.

CHAPTER 1

INTRODUCTION

From the moment that mainframes, back in the 1950s, had to be shared by more than one user, considerable effort has been spent in finding an answer to a simple question: *"Who goes first?"* How to allocate shared computational resources is, perhaps, one of the oldest practical problems in computing and many solutions have been developed over the decades, from operating systems that decide which user's process gets the CPU next, to sophisticated schedulers that decide if the tens of thousands of processors in a supercomputer will be dedicated next to simulating a supernova or to modelling global climate changes.

In this dissertation, I address a number of specific problems that arise when provisioning computational resources in a distributed system (involving multiple computers connected by a network, such as a cluster), and show how some of those problems can be overcome through the use of virtual machines and a leasing abstraction. This introduction provides an informal bird's–eye view of the scientific work presented in my dissertation, including what to expect in each chapter.

The problem in a nutshell

The problem I address in my dissertation is, broadly speaking, *how to provision computational resources* and, more specifically, how to do so in a distributed system. When these systems are shared, as they almost always are, by multiple users competing for these resources, *something* has to determine who gets the resources (if at all) and when. If I have a cluster of ten machines, and five users want to use all ten machines at the same time, how do I decide who gets the machines first? First come, first serve? What if one of the users has a pressing deadline? What if one of the users has historically used more resources than the other users? Wouldn't it be 'fair' to then give priority to the other users? etc.

Back in the 1950s, the solution was fairly rudimentary: users printed their programs on punched cards and submitted them to a human operator, who decided the order in which those programs would be run by the mainframe. Nowadays, this task is handled more automatically by software and, although many software solutions have emerged, some geared for managing just a few networked computers and others capable of managing supercomputers with more than a million cores, these solutions have tended to be specialized towards particular use cases.

For example, provisioning resources for batch jobs on a shared cluster has been extensively studied, resulting in job management systems systems like Torque/Maui, Sun Grid Engine, LoadLeveler, and many others, that can queue and prioritize job requests efficiently (in these systems, efficiency is defined in terms of a variety of metrics, including waiting times and resource utilization). These systems are ideal for scenarios where users are willing to wait for resources and, once they get them, use them for a relatively short time (in the order of minutes, hours, and sometimes days, but never indefinitely). However, these systems are not necessarily good at dealing with other provisioning scenarios, such as users that need resources at specific times or need guaranteed resources with very short notice. Similarly, resource provisioning approaches geared towards provisioning resources at specific times or immediately are not typically well suited for running batch jobs.

Thus, there is no general resource provisioning system that meets the requirements of different usage scenarios simultaneously, reconciling the different measures of efficiency in each scenario. That, in a nutshell, is the problem that I address in this dissertation.

The solution in a nutshell

If the problem is that there is no general resource provisioning system, a necessary first step is, thus, to define a general-purpose resource provisioning abstraction (i.e., one not coupled to a particular use case). In this dissertation, I propose using a *lease* abstraction. Informally, a

lease is a form of contract where one party –the lessor, or *resource provider*– agrees to provide a set of *resources* (an apartment, a car, computational resources, etc.) to another party – the lessee, or *resource consumer*– under a set of *terms* governing the amount of resources, access to the resources, the duration of the lease, etc. In the context of leasing computation resources, these terms should encompass the *hardware resources* required by the resource consumer, such as CPUs, memory, and network bandwidth; a *software environment* required on the leased resources; and an *availability period* during which the requested hardware and software resources must be available.

Of course, there are many different types of resource providers, such as a university's batch job scheduler, computational grids, Infrastructure-as-a-Service (IaaS) clouds, and companies that sell servers on their datacenters. When we obtain resources from any of these providers, we are implicitly entering into a lease agreement, although the set of possible terms we can specify to each provider is limited. For example, we can specify detailed software terms (by providing our own software stack in the form of a disk image) when requesting resources from an IaaS cloud, such as Amazon EC2, whereas job schedulers and grids typically provide little control over the software stack the jobs will run on. On the other hand, the availability terms in IaaS clouds only allow users to request resources for immediate use; unlike some job schedulers and grid interfaces, there is no possibility of reserving resources in advance or specifying other lease terms governing Quality-of-Service (QoS).

The lease abstraction that I propose in this dissertation aims to be expressive enough to support a variety of terms. Of course, defining a lease abstraction —and fantasizing about all sorts of lease terms— is easy. In fact, a leasing abstraction has been used in multiple fields of computer science, most notably networking, although there is no universally accepted definition of "lease". The hard part of solving this problem —the lack of a resource provisioning system supporting a general-purpose lease abstraction— is simultaneously and *efficiently* supporting leases with terms that have conflicting measures of efficiency.

For example, consider the problem of simultaneously supporting best-effort batch jobs, where the resource consumer needs some resources but is willing to wait for them, and advance reservations, where the resource consumer needs resources at a very specific time. In advance reservations, the measure of efficiency is very simple: the resources are either provisioned at the exact time they are requested or they are not. In best-effort jobs, the measure of efficiency is typically some variation on the turnaround time of the job: if a job runs for ten hours, and the job scheduler is able to provision resources for that job in less than an hour, most users may consider that "efficient". However, if we want to support both best-effort jobs and advance reservations, these measures of efficiency are in conflict. If I accept a reservation for all my resources from 2pm to 4pm, this introduces a 'roadblock' in the resource schedule, and best-effort jobs requested before or during that time may experience longer turnaround times. In fact, although many job schedulers do support advance reservations, system administrators use them judiciously, precisely because they are known to impact the performance of best-effort jobs negatively.

In this dissertation I hold that the best way of simultaneously and efficiently supporting multiple types of leases is by implementing them with *virtual machines* (VMs), which have a number of properties that make them attractive for this purpose. If we want to support hardware lease terms, virtualization technology can be used to partition a single physical machine into multiple virtual machines, allowing fine-grained division of hardware resources. Additionally, we can also support software lease terms since each VM can have its own software stack. Finally, the ability to suspend, resume, and migrate VMs transparently (without affecting computation inside the VM) is a promising mechanism for supporting multiple availability terms (best-effort, advance reservation, deadlines, etc.), and one that receives considerable attention in this dissertation.

So, in a nutshell, the solution to the problem is *leases and virtual machines*. In this dissertation, I flesh out this solution in five steps:

Step 1: I begin by providing a formal definition of a lease.

Step 2: Next, I explore whether VMs are, in fact, an adequate vehicle for implementing leases. I present and evaluate several scheduling algorithms that exploit the suspend/resume/migrate capability of VMs to determine if they can be used to support at least three types of leases efficiently: best-effort, advance reservation, and best-effort with deadlines.

However, using VMs is itself problematic, since it introduces a number of overheads that can negatively impact performance. Most notably, running a VM with a custom software stack may involve transferring potentially large disk images to the physical machine where the VM will run. Similarly, although we can suspend the VMs in a lease to free up resources for another lease, this operation requires saving the VMs' entire memory and state to disk, which should be done before the start of the other lease.

Thus, in my initial analysis of the benefits and drawbacks of using VMs to implement leases, I make two simplifying assumptions about the VMs: (A) disk images for VMs are predeployed in the machines where they are needed and (B) there is at most one VM per physical machine. I also make a simplifying assumption about leases: (C) a lease request is never rejected if there are enough resources to satisfy that request.

Step 3: Although these two assumptions simplify an initial analysis, they must be removed. So, the next step is to remove assumption (A). I present and evaluate strategies for transferring and reusing disk images in such a way that lease terms are not broken (e.g., by guaranteeing that a required disk image is transferred before the start of an advance reservation) and to mitigate the overhead of having to deploy large disk images before the start of a lease.

Step 4: Next, I address assumption (B). In particular, supporting multiple VMs per physical

machines will affect how VMs are suspended and resumed, since there will now be multiple suspend/resume operations competing for I/O. I present a more general model for suspending and resuming VMs in a lease, and explore the effects of having multiple VMs per physical node.

Step 5: Finally, I address assumption (C). Having an "accept all" policy provides no incentives for resource consumers to *not* request more resources than they need, and also provides a resource provider with no mechanism other than lease refusal to signal that a resource is overloaded. I address this lack of admission policies by exploring price-based policies for lease admission, hypothesizing that prices can be used both to affect demand (by pricing leases according to the requested terms) and to signal (via higher prices) when a resource is overloaded.

In this dissertation, I also present an architecture for leasing with VMs and provide an open source reference implementation called Haizea (http://haizea.cs.uchicago.edu/). All the experiments in this paper have been performed using Haizea, which can simulate the scheduling of lease workloads. Haizea can also act as a drop-in scheduler for the open source virtual infrastructure manager OpenNebula (http://www.opennebula.org/) and some of the experiments were performed in this configuration, with Haizea using OpenNebula to manage real VMs on a pool of physical resources.

How this dissertation is structured

This dissertation is divided into several chapters that follow the general outline described above, but present my arguments more formally and in greater details:

Chapter 2: The Problem of Provisioning Computational Resources presents a more concise problem statement, with a discussion of prior work on this problem, and enumerating the precise goals that a solution should meet.

Chapter 3: Resource and Leasing Model provides a formal definition of leases. This corresponds to **Step 1** above.

Chapter 4: The Haizea Lease Manager. Since all the experiments presented in subsequent chapters rely on the Haizea Lease Manager, this chapter provides a description of Haizea and its architecture.

Chapter 5: Scheduling Leases with Virtual Machines corresponds to **Step 2** of the solution. The work presented in this chapter expands on the work presented in the following papers:

> B.Sotomayor, K.Keahey, I.Foster, *Combining Batch Execution and Leasing Using Virtual Machines*. The 17th International Symposium on High Performance Distributed Computing (HPDC 2008), June 23-27, 2008, Boston, Massachusetts, USA.

> B.Sotomayor, K.Keahey, I.Foster, T.Freeman, *Enabling Cost-Effective Resource Leases with Virtual Machines*. Hot Topics session in the 16th International Symposium on High Performance Distributed Computing (HPDC 2007), June 27-29, 2007, Monterey Bay, California, USA.

> B.Sotomayor. *A Resource Management Model for VM-based Virtual Workspaces*. Master's Paper, University of Chicago, Department of Computer Science. February 23rd, 2007.

Chapter 6: Scheduling Disk Image Transfers corresponds to **Step 3** of the solution, also expanding on the work presented in the above three papers.

Chapter 7: Modelling and Scheduling Virtual Machine Suspension and Resumption Times corresponds to **Step 4** of the solution and expands on the following paper:

> B.Sotomayor, R.Santiago Montero, I.Martín Llorente, I.Foster, *Resource Leasing and the Art of Suspending Virtual Machines*. The 11th IEEE International Conference on High Performance Computing and Communications (HPCC 2009), June 25-27, 2009, Seoul, Korea.

Chapter 8: Pricing Strategies for Leases corresponds to **Step 5** of the solution.

This dissertation concludes with Chapter 9, a discussion of the extent to which my solution meets the goals presented in Chapter 2 and what my future work will involve. A single appendix provides detailed instructions on how to reproduce the results presented in this text.

Supplementary files, including files necessary to reproduce the results, and errata for this dissertation can be found at the following URL:

http://people.cs.uchicago.edu/~borja/dissertation/

CHAPTER 2

THE PROBLEM OF PROVISIONING COMPUTATIONAL RESOURCES

The need for computational resources has, over the years, become a fundamental requirement in both science and industry. In many cases, this need is transient: a user may only require computational resources for the duration of a well-defined task. For example, a scientist could require a large number of computers to run a simulation for just a few hours, but might not need those computers at any specific time (as long as they are made available in a reasonable amount of time). A college instructor may want to make a cluster of computers available to students during the course's lab sessions, at very specific times during the week, and with a specific software configuration. A telecommunications company could posses an existing infrastructure that hosts a number of websites, but may need to supplement that infrastructure with additional resources during periods of unforeseen increased web traffic, meaning those resources have to made available right away with very little advance notice.

These transient resource usage scenarios pose the problem of how to *provision shared computational resources efficiently*. This problem has been studied for decades, resulting in approaches that tend to be highly specialized to specific usage scenarios. For example, the problem of how to run multiple jobs on a shared cluster has been extensively studied, resulting in job management systems systems like Torque/Maui [31], Sun Grid Engine[1], LoadLeveler[2], and many others, that can queue and prioritize job requests efficiently (in these systems, efficiency is defined in terms of a variety of metrics, including waiting times and resource utilization). Such a system would meet the requirements of the scientist wanting to run simulations during a few hours but, on the other hand, the college instructor and the telecommunications company mentioned above would be ill-served by a job management

1. http://gridengine.sunsource.net/

2. http://www.ibm.com/systems/clusters/software/loadleveler.html

system and the efficiency metrics typically used in job management. Conversely, other resource provisioning approaches are not particularly well suited for job-oriented computations (this point will be explored in greater detail throughout this chapter).

Thus, there is no general solution that can provision resources meeting the requirements of different usage scenarios simultaneously, such as those mentioned above, reconciling the different measures of efficiency in each scenario. More specifically, much of my work is motivated by the combination of *best-effort* resource requirements, where a user needs computational resources but is willing to wait for them (possibly setting a deadline), and *advance reservation* resource requirements, where the resources must be available at a specific time. In the former, efficiency is typically measured in terms of waiting times (or similar metrics such as turnaround times or slowdowns) or throughput, while the latter is usually concerned with providing the requested resources at exactly the agreed-upon times without interruption, and both are concerned with maximizing the use of hardware resources and possibly monetary profit. Although both best-effort and advance reservation provisioning have been studied separately, the combination of both is known to produce utilization problems (discussed in Section 2.3) and is discouraged in practice.

In this dissertation I seek to develop a resource provisioning model and architecture that can support multiple resource provisioning scenarios efficiently and simultaneously, with an initial focus on the best-effort and advance-reservation cases mentioned above, and arguing in favour of a *lease-based* model, where leases are implemented as *virtual machines* (VMs). This model must meet the following goals:

G1-ResProv **Provide an abstraction focused solely on resource provisioning**

Although the lease abstraction has been used in multiple fields of computer science, most notably networking, there is no universally accepted definition of "lease." However, leases generally always provide an abstraction for, first and foremost, provisioning a resource (bandwidth in networks, raw hardware resources in datacenters, etc.) oper-

ated by a lessor (or *resource provider* and provided to a lessee (or *resource consumer*), with relatively few restrictions on how the provisioned resources can be used. So, when proposing a lease-based model, the implied goal is that resource consumers will be able to use a general-purpose resource provisioning abstraction (i.e., not one that is coupled to a particular use case).

G2-HwSwAvail **Provision hardware, software, and availability**

Resource provisioning can encompass three dimensions: hardware resources, the software available on those resources, and the time during those resources must be guaranteed to be available. A complete resource provisioning model must allow resource consumers to specify requirements across these three dimensions, and the resource provider to efficiently satisfy those requirements.

G3-Reconcile **Reconcile requirements of different types of leases**

Best-effort and advance reservation provisioning have different measures of efficiency and, in some cases, these measures will be in conflict. For example, accepting advance reservations unconditionally may delay or even preempt best-effort leases but, on the other hand, a policy of not allowing best-effort leases to be delayed or preempted may reduce the number of advance reservations that can be accepted. Reconciling these measures of efficiency requires developing scheduling algorithms capable of combining both types of leases, and potentially others, and policies that can guide the scheduling decisions based on the goals and requirements of the resource provider. Furthermore, policies for *lease admission* are also required to determined whether a lease should be accepted or rejected up front, regardless of whether it can be scheduled (e.g., a provider may be unwilling to accept advance reservations that are requested less than 24 hours in advance). Taking into account the different overheads of virtual machines adds an additional layer of complexity to the problem of scheduling VM-based leases.

G4-ModelVirt **Model virtual resources accurately and schedule them efficiently**

The choice of virtual machines to implement leases requires modelling virtualized resources and operations on those resources. In particular, using virtual machines involves different types of overhead (most notably the overhead of transferring virtual machine images, and the overhead of suspending and resuming virtual machines) that must be modelled accurately so they can be taken into account when scheduling virtual machines.

The remainder of this chapter reviews existing approaches to this problem, including VM-based approaches (Section 2.1), lease-based approaches (Section 2.2), and job-based approaches (Section 2.3). After describing these approaches and, in particular, to what extent they meet the four goals described above, the chapter concludes with a discussion of their common shortcomings (Section 2.4) and presents the contributions this dissertation makes to the state of the art (Section 2.5).

2.1 VM-based approaches

In my work, I argue in favour of using virtual machines as a vehicle for implementing leases. Virtual machines are an appealing vehicle for resource management because they can be used to provision hardware, software, and availability (G2-HwSwAvail):

Hardware resources Virtual machines can be mapped to all or part of a physical node's hardware resources, allowing users to request fine-grained resource allocations. Additionally, virtual machines have the added property of allowing these allocations to be strictly enforceable.

Software environment A virtual machine can encapsulate a custom software environment, allowing users to support existing applications without modification. Resource providers can also allow users to have administrative privileges within their VMs, with

a reduced risk of malicious use thanks to the security and isolation properties of VMs. Additionally, popular VM systems, such as Xen and VMWare, primarily support the x86 architecture, facilitating virtualization of existing x86 computational resources. Although these systems require the VMs to use the x86 architecture too, a model based on VMs could easily support additional architectures once the VM vendors supported them.

Availability VMs can be suspended, potentially migrated, and resumed without modifying any of the applications running inside the VM. This functionality has the potential to allow leases with best-effort and advance reservation availability periods to be combined efficiently (G3-RECONCILE), by suspending VMs of best-effort leases before the start of advance reservation leases. Although there are other mechanisms to suspend/resume/migrate computation (such as checkpointing and preempting schedulers, described below), VMs provide a more versatile solution because they do not require applications, or even the OS running inside the VM, to be checkpointing-aware.

Although an attractive option, virtual machines also raise additional challenges (G4-MODELVIRT) related to the overhead of using VMs:

Preparation overhead When using VMs to implement leases, a VM disk image must be either prepared on-the-fly or transferred to the physical node where it is needed. Since a VM disk image can have a size in the order of gigabytes, this preparation overhead can delay significantly the starting time of leases. This delay may, in some cases, be unacceptable for advance reservations that must start at a specific time.

Runtime overhead Once a VM is running, actions like checkpointing and resuming can incur in significant overhead since a VM's entire memory space must be saved to disk, and then read from disk. Migration involves transferring this saved memory along

with the VM disk image. Similar to deployment overhead, this overhead can result in noticeable delays.

Several groups have explored the use of virtual machines as a resource provisioning mechanism, sometimes providing lease-like semantics. These approaches are broadly divided into those that propose using virtual machines to create "virtual clusters" and those that provide lease-like semantics on large datacenters.

2.1.1 Virtual clusters

A number of groups have developed solutions that use VMs to create "virtual clusters" on top of existing infrastructure. Nishimura et al.'s [41] system for rapid deployment of virtual clusters can deploy a 190-node cluster in 40 seconds. Their system accomplishes these low deployment times by representing software environments as binary packages that are installed on the fly on generic VM images. They optimize installation by caching packages on the nodes, thus reducing the number of transfers from a package repository. This approach limits the possible software environments to those that are expressible as installable binary packages (which is not always possible; e.g., highly specialized scientific environments where installable binary packages may not be readily available) but does provide a faster alternative to VM image deployment if the installation time is short enough. Yamasaki et al. [58] improved this system by developing a model for predicting the time to set up a new software environment on a node, allowing their scheduler to choose nodes that minimize the time to set up a new virtual cluster. This model takes node heterogeneity into account and uses the parameters of each node (CPU frequency and disk read/write speeds) and empirical coefficients to predict the time to transfer and install all required packages, and then reboot the node. However, their model does not include an availability dimension and assumes that all resources are required immediately.

The Nimbus toolkit[3] [32] has the ability to deploy "one-click" virtual clusters [19, 34] in sites with different software and network configurations, eliminating the need to adapt virtual machine images manually each time they are deployed in a new site (where, for example, the NFS server might have a different address, or services on the network might expect a digital host certificate signed by the local Certificate Authority). This automated configuration is accomplished by using a standalone context broker that *contextualizes* disk images to work in a specific site.

Fallenbeck et al. [12] extended the Sun Grid Engine scheduler to use the save/restore functionality of Xen VMs, allowing large parallel jobs to start earlier by suspending VMs running serial jobs, and resuming them after the large parallel job finished. Emeneker et al. [11] extended the Moab scheduler to support running jobs inside VMs, and explored different caching strategies for faster VM image deployment on a cluster. However, both studies use VMs only to support the execution of best-effort jobs and do not currently schedule image transfers separately; moreover, the Moab work does not integrate caching information into scheduler decisions.

Walters et al. [57] have proposed the use of a new VM-centric job scheduler, called UBIS, capable of scheduling both traditional batch jobs and high-priority interactive jobs by using the suspend/resume capability of virtual machines to preempt running batch jobs and accommodate incoming requests for interactive jobs. The UBIS scheduler not only facilitates support for interactive jobs, it also accomplishes impressive improvements (up to 500%) in resource utilization and response time for batch jobs. However, it does not support advance reservation of resources, instead focusing on supporting interactive jobs with near-immediate resource requirements, which simply allows the UBIS scheduler to perform preemption operations when an interactive job is requested. Supporting advance reservations in such a way that starting times can be guaranteed would require modelling this overhead

3. Previously known as the Virtual Workspaces Service [33]

and scheduling the preemption operations to finish before the start of a reservation.

Other groups have explored a variety of challenges involved in deploying and running a virtual cluster, including virtual networking and load balancing between multiple physical clusters (VIOLIN/VioCluster [49, 48]), automatic configuration and creation of VMs (In-VIGO [1] and VMPlants [36]), and communication between a virtual cluster scheduler and a local scheduler running inside a virtual cluster (Maestro-VC's two-level scheduling [35]). However, they do not explore workloads that combine best-effort and advance reservation requests, nor do they schedule deployment overhead of VMs separately.

In general, all these solutions use virtual machines to great effect, addressing goal G2-HwSwAvail and, to a certain extent, G4-ModelVirt. However, all of them focus on provisioning resources for batch jobs (not providing a general provisioning abstraction, G1-ResProv) and focus on a single availability scenario (mostly the execution of batch jobs on a virtual cluster, which makes G3-Reconcile moot), except for Walters et al., who consider workloads combining both batch jobs and interactive jobs with near-immediate availability requirements. As far as G4-ModelVirt, only Nishimura et al. and Yamasaki et al. model and schedule the deployment overhead of virtual machines, while other groups either assume that this overhead does not exist (e.g., by assuming that VM disk images are predeployed) or can be ignored.

2.1.2 Datacenter-based solutions

Whereas the above solutions focus on creating virtual clusters, mostly for the purposes of job-based batch processing, other solutions focus on providing lease-like semantics where resource providers manage a datacenter and allow resource consumers to lease parts of it using virtual machines; a virtual cluster would be just one possible application of what the resource consumers could do with their virtual machines. In the server hosting arena, datacenters with virtualized resources have been a popular option for several years as a way of leasing resources

where clients are given complete control over the leased resources, but without requiring a dedicated server per client. These are a popular option for hosting web/mail/DNS/etc. servers at a low price, but typically require leases with a minimum duration of a month. More recently, Amazon's EC2[4] introduced the notion of *cloud computing*, where virtual machines are provisioned immediately with customized software environments and charging for use by the hour. OpenNebula [55], Nimbus, and Eucalyptus [42] provide an open-source alternative to Amazon's EC2, using the same web services interface and providing similar functionality.

Since this datacenter-based approach typically involves managing a large amount of virtual and physical servers, in the order of hundreds or thousands, efficiently managing the *virtual infrastructure* in the datacenter becomes a major concern. Several solutions, such as VMWare VirtualCenter, Platform Orchestrator, Enomalism, or OpenNebula have emerged to manage virtual infrastructures, providing a centralized control platform for the automatic deployment and monitoring of virtual machines (VMs) in datacenters. These solutions excel at providing users with exactly the software environment they require, and most provide a large number of hardware options. However, they depend on an immediate provisioning model, where virtualized resources are allocated at the time they are requested, without the possibility of requesting resources at a specific future time and, at most, being placed in a simple first-come-first-serve queue when no resources are available.

Since the workloads in datacenters typically involve servers running for long periods of time (in the order of months) with variable resource requirements, several groups have looked into the problem of how to use fewer physical servers by *consolidating* multiple virtual servers into single physical machines. This challenging problem involves characterizing server workloads, predicting future resource demand [3], and then using this information to consolidate multiple servers on a single machine in such a way that the probability of breaching existing

4. http://aws.amazon.com/ec2/

service-level agreements (SLAs) is minimized. This consolidation can be done when processing the requests for new servers (static consolidation) or it can be done while the servers are running (dynamic consolidation [3, 39]), typically by leveraging the live migration capability of virtual machines to optimize the mapping of VMs to physical machines.

Although all these solutions use a general lease-like abstraction (G1-ResProv) that allows users to request both hardware and a specific software environment (G2-HwSwAvail), the lease terms are limited to just immediate availability; there is no possibility of requesting resources in advance or queuing requests, meaning these solutions have no need to address G3-Reconcile. Nimbus, Eucalyptus, OpenNebula, and Enomalism all use a basic resource model that does not explicitly schedule VM overhead (G4-ModelVirt). Since the other cited work is closed-source and not peer-reviewed, it is hard to assess to what extent it addresses G4-ModelVirt.

2.2 Lease-based approaches

A purely lease-based approach to resource provisioning has been proposed by Grit et al. [23] and other members of Jeff Chase's research group [30, 45]. However, their work focuses mainly on leases in *federated* systems managed by their ORCA and Shirako systems. In such a system, resource providers can donate part of their resources to a broker (or multiple brokers) which can, in turn, give resource consumers "tickets" redeemable for actual resources when presented to a resource provider. Federation of leases across multiple sites is outside the scope of my work, which focuses on how resources are managed inside the resource provider. In fact, in the ORCA architecture, the work resulting from this dissertation would be a resource provisioning "actor", the internal workings of which are supposed to be independent of ORCA, although the resources are assumed to be partitionable.

Most of their work uses virtual machines, managed by their Cluster-On-Demand system, as an example of a partitionable resource (although they emphasize that their work is ap-

plicable to any partitionable resource), and more recent work has focused on how to enable batch job execution within their architecture [22, 24]. However, their model assumes that any overhead involved in deploying and managing the VM will be deducted from the lease's availability, instead of scheduling it separately (G4-ModelVirt). Additionally, despite providing a well thought-out leasing abstraction (G1-ResProv and G2-HwSwAvail), this abstraction focuses on federated systems, whereas I focus on leasing at the local level (i.e., within a single administrative domain), which allows me to assume that a lease scheduler will have absolute control over all resources.

2.3 Job-based approaches

In science and academia and, to some extent, in industry, resource provisioning has been mostly tied to running *jobs* and many job schedulers have been developed over the years, such as Maui[5] [31], Moab[6], LSF[7], LoadLeveler[8], PBS Pro[9], and SGE.[10] However, job-based systems provision resources as a side-effect of having to run a job. So, although these systems can support both best-effort and advance reservation provisioning, resource consumers are required to interact with those resources using the job abstraction (i.e., goal G1-ResProv is not met). Additionally, these systems include limited support for custom software environments (G2-HwSwAvail), typically limiting resource consumers to whatever software environment happens to be available on the hardware resources being provisioned. The only exception is Moab, which provides limited support for starting up a virtual machine encapsulating the software environment required by the job. However, Moab only allows access to

5. http://www.clusterresources.com/pages/products/maui-cluster-scheduler.php/
6. http://www.clusterresources.com/pages/products/moab-cluster-suite/workload-manager.php
7. http://www.platform.com/
8. http://www.ibm.com/systems/clusters/software/loadleveler.html
9. http://www.pbspro.com/
10. http://gridengine.sunsource.net/

a limited number of VM-based environments, which still require considerable software contextualization on the part of the cluster administrator, and does not address the overhead of setting up those VMs (G4-ModelVirt)

Nonetheless, job scheduling has driven a considerable amount of research and led to important algorithms and results relevant to best-effort scheduling, including how to schedule advance reservations alongside best-effort workloads. Job schedulers typically depend on queues to prioritize access to resources, using backfilling [37, 40, 16] to optimize queue ordering. When using backfilling, the scheduler can make reservations in the future for requests that cannot be scheduled immediately, allowing subsequent requests to skip to the front of queue, as long as they finish before the future reservations. Some of the scheduling algorithms used in this dissertation depend on backfilling, but extend it by leveraging the suspend/resume capability of virtual machines.

The remainder of this section describes how advance reservations are supported in job-based systems, and how they can result in utilization problems. Although preempting schedulers partially palliate the utilization problems of advance reservations, they do not fully address some of the goals in my dissertation. Finally, I discuss multi-level scheduling solutions which use job-based systems purely as a resource provisioning tool (which would meet goal G1-ResProv), sidestepping the job abstraction and using other provisioning abstractions on the resources

2.3.1 Advance reservations in job-based systems

Although job schedulers can schedule advance reservations alongside best-effort workloads and, arguably, could be used to implement best-effort and advance reservation leases, these advance reservations fall short in several aspects. First of all, they are constrained by the job abstraction which, as described above, does not meet some of the goals I outlined for my dissertation. More specifically, when a user makes an advance reservation in a job-based

system, the user does not have direct access to those resources but, rather, is allowed to submit jobs to them. For example, PBS Pro creates a new queue that will be bound to the reserved resources, guaranteeing that jobs submitted to that queue will be executed on them (assuming they have permission to do so). Maui and Moab, on the other hand simply allow users to specify that a submitted job should use the reserved resources (if the submitting user has permission to do so). There are no mechanisms to directly login to the reserved resources, other than through an interactive job, which does not provide unfettered access to the resources (i.e., no possibility of root access), or more ad-hoc methods, like requesting login privileges from the cluster administrator for the duration of the reservation.

Additionally, it is well-known that advance reservations lead to utilization problems [18, 53, 54, 38], caused by the need to vacate resources before a reservation can begin. Unlike future reservations made by backfilling algorithms, where the start of the reservation is determined on a best-effort basis, advance reservations introduce roadblocks in the resource schedule. Thus, traditional job schedulers are unable to schedule workloads combining both best-effort jobs and advance reservations efficiently (G3-Reconcile).

However, advance reservations can be supported more efficiently by using a scheduler capable of preempting running jobs at the start of the reservation and resuming them at the end of the reservation. Preemption can also be used to run large parallel jobs (which tend to have long queue times) earlier, and is especially relevant in the context of urgent computing, where resources have to be provisioned on very short notice and the likelihood of having jobs already assigned to resources is higher. While preemption can be accomplished trivially by cancelling a running job, the least disruptive form of preemption is *checkpointing*, where the preempted job's entire state is saved to disk, allowing it to resume its work from the last checkpoint. Additionally, some schedulers also support job migration, allowing checkpointed jobs to restart on other available resources, instead of having to wait until the preempting job or reservation has completed.

However, although many modern schedulers support at least checkpointing-based preemption, the job's executable must itself be checkpointable. An application can be made checkpointable by explicitly adding that functionality to an application (application-level and library-level checkpointing) or transparently by using OS-level checkpointing, where the operating system (such as Cray, IRIX, and patched versions of Linux using BLCR [25]) checkpoints a process, without rewriting the program or relinking it with checkpointing libraries. However, this requires a checkpointing-capable OS to be available.

Thus, a job scheduler capable of checkpointing-based preemption and migration could be used to address G3-RECONCILE, by checkpointing jobs before the start of an advance reservation, minimizing their impact on the schedule. However, the application- and library-level checkpointing approaches burden the user with having to modify their applications to make them checkpointable, imposing a restriction on the software environment being leases (G2-HWSWAVAIL). OS-level checkpointing, on the other hand, is a more appealing option, but still imposes certain software restrictions on resource consumers. Systems like Cray and IRIX still require applications to be compiled for their respective architectures, which would only allow a small fraction of existing applications to be supported within leases, or would require existing applications to be ported to these architectures. This porting is an excessive restriction for supporting leasing, given the large number of clusters and applications that depend on the x86 architecture. Although the BLCR project does provide a checkpointing x86 Linux kernel, this kernel still has several limitations, such as not being able to checkpoint network traffic properly, and not being able to checkpoint MPI applications unless they are linked with BLCR-aware MPI libraries.

An alternative approach to supporting advance reservations was proposed by Nurmi et al. [43], which introduced "virtual advance reservations for queues" (VARQ). This approach overlays advance reservations over traditional job schedulers by predicting the time a job would spend waiting in a scheduler's queue, and submitting a job (representing the advance

reservation) at a time such that, based on the wait time prediction, the probability that it will be running at the start of the reservation is maximized. Since no actual reservations can be done, VARQ jobs can run on traditional job schedulers, which will not distinguish between the regular best-effort jobs and the VARQ jobs. Although these virtual reservations are an interesting approach that can be realistically implemented in practice (since it does not require modifications to existing schedulers), it still depends on the job abstraction (G1-ResProv).

Hovestadt et al. [27, 26] propose a planning-based (as opposed to queuing-based) approach to job scheduling, where job requests are immediately planned by making a reservation (now or in the future), instead of waiting in a queue. Thus, advance reservations are implicitly supported by a planning-based system. Additionally, each time a new request is received, the entire schedule is reevaluated to optimize resource usage. For example, a request for an advance reservation can be accepted without using preemption, since the jobs that were originally assigned to those resources can be assigned to different resources (assuming the jobs were not already running). Although this approach is promising, and is arguably better in qualitative terms to a queuing-based approach, the authors show no quantitative results comparing their approach to a queue-based system or to a checkpointing-capable system.

2.3.2 Hierarchical/multi-level scheduling

In a hierarchical, or multi-level, scheduling model, a scheduler allocates resources that will be managed by a different scheduler. This approach has been widely used in the context of OS process scheduling, where one scheduler is responsible for allocating and managing heavy processes, while a separate scheduler, inside the process or as part of the OS, manages the threads more efficiently than the heavy process scheduler. Since job-based systems tightly couple job execution to resource provisioning, multi-level scheduling solutions have

emerged to circumvent this coupling: a job-based system is used to provision the resources but, instead of "running a job", the provisioned resources are then managed by a different scheduler, which can use different provisioning abstractions. Thus, multi-level scheduling approaches could be used to meet goal G1-ResProv, allowing resource providers to lease resources without having to shift from a job-based system to a purely lease-based resource manager.

The Condor scheduler's "glidein" mechanism [21] was the first to apply this model on compute clusters, using existing job schedulers to provision resources, starting (or "gliding in") Condor daemons on the provisioned resources, and then using an existing Condor pool to manage those resources. The MyCluster project [56] similarly allows Condor or SGE clusters to be overlaid on top of TeraGrid resources to provide users with "personal clusters." The Falkon task scheduler [29] can also be deployed through a GRAM interface on compute resources, and is specifically optimized to manage the execution of lightweight tasks, typically in a workflow managed by the Swift [60] system. Virtual workspaces [33, 19] also follow a multi-level scheduling approach by allowing users to create a workspace (represented as either a virtual machine or a dynamically-created UNIX account) on a remote site through a Virtual Workspace Service (VWS), and then allowing the user to access that workspace directly, and not through the VWS. The Workspace Pilot furthermore allows a job scheduler to allocate resources for the VWS, which are then managed by the VWS to create a virtual workspace on those resources[20].

By using a multi-level scheduling approach, Condor, MyCluster, Falkon, and the VWS can use their respective provisioning abstractions, sidestepping the site's job scheduling entirely. The first level of these multi-level scheduling solution approximates leasing, since it focuses only on resource provisioning. However, it is still limited to requesting the availability periods supported by job schedulers (best-effort jobs or advance reservations), and does not allow users to access them directly. Condor still requires the user to access the resources

through Condor job submissions; similarly, MyCluster requires Condor or SGE jobs to be used, and Falkon requires computation to be expressed as tasks. Even if we overcame this issue (e.g., in theory, a job scheduler could be used to provision resources that could then be used to run an SSH server that allows the user to log into the provisioned resources), there is no support for deploying custom software environments.

2.3.3 Quality-of-Service and pricing in resource provisioning

Support for advance reservations is sometimes framed in the more general context of providing specific Quality-of-Service (QoS) guarantees, including scheduling jobs with deadlines. Approaches to guaranteeing QoS include the use of preemption [14, 13, 10], laxity in reservations [28, 14, 13, 46], and incentives to request constraints that maximize performance [4]. The benefits of ARs to provide QoS in distributed systems has also been explored by Siddiqui et al. [50] and Castillo et al. [5, 6, 7].

A related problem is how resources are priced in systems that provide QoS. Singh et al. [52] propose an adaptive pricing scheme for ARs where the price of the reservation is based not just on its size and duration, but also on how much it delays other running (non-AR) jobs. Additionally, consumers are given a choice of several possible start times for their reservations (with different prices) and can choose the one that minimizes a cost function that depends on how urgently the user needs the resources. In our work, consumers are given a single price determined by an adaptive pricing strategy that seeks to maximize provider revenue.

Infrastructure-as-a-Service "clouds" allow consumers to purchase metered capacity in the form of VMs with some QoS guarantees, albeit not for each individual request. For example, Amazon EC2 guarantees "an Annual Uptime Percentage of at least 99.95% during the Service Year"[11], but consumers cannot request stricter guarantees such as ARs or

11. http://aws.amazon.com/ec2-sla/

deadlines. Large public clouds such as Amazon EC2 arguably do not need to offer ARs or deadline-sensitive provisioning because they have enough resources to provide the illusion of infinite capacity where resources can be provisioned immediately. Also of note are the pricing levels offered by Amazon. *On-demand instances* are provisioned immediately and charged at a flat rate (between $0.085 and $2.88 depending on the VM's capacity and OS), and are recommended for "applications with short term, spiky, or unpredictable workloads that cannot be interrupted"[12]; *Reserved Instances*[13], allow users to pay a lower hourly rate per instance during a one or three year term if they make an up-front payment for that instance and are recommended for "applications with steady state or predictable usage [or requiring] reserved capacity, including disaster recovery"; *Spot instances* are charged at a variable rate, and remain available as long as that rate is below a maximum rate specified by the consumer, and are recommended for "applications that have flexible start and end times [or] that are only feasible at very low compute prices", thus providing a lower QoS alternative to on-demand instances. In our work, we propose a model suitable for applications with rigid QoS requirements. In this model, leases are priced at a variable rate but, unlike Amazon's spot pricing, but the terms of a lease cannot be breached.

Since part of G3-RECONCILE involves developing policies to determine whether a provider should enter into a lease agreement, regardless of whether the lease can be scheduled or not, the problem of how to price resources involving a QoS guarantee, such as advance reservations, is relevant to my work because it provides a way of performing lease admission.

2.4 Discussion of Existing Approaches

Although many of the approaches described in this chapter excel at addressing one or more of the four goals described earlier, no single solution manages to meet all four goals (Figure 2.1

12. http://aws.amazon.com/ec2/purchasing-options/

13. http://aws.amazon.com/ec2/reserved-instances/

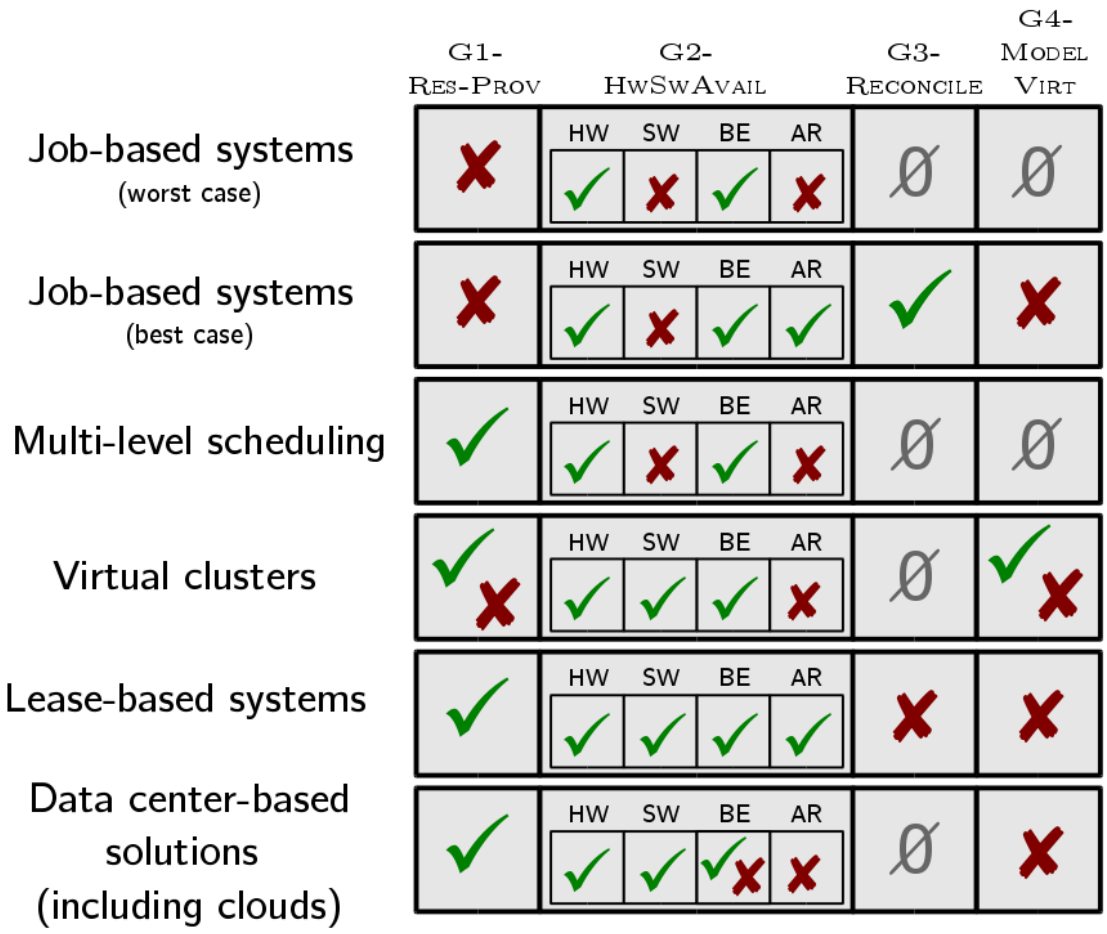


Figure 2.1: Summary of related work. Each of the columns corresponds to the four goals described at the beginning of this chapter. G2-HWSWAVAIL is further divided into whether the solution can provision hardware (HW), software (SW), best-effort availability (BE), and advance reservation availability (AR). A ✓ indicates the solution satisfies the goal, a ✗ indicates it does not, and ∅ indicates the goal is not applicable (e.g., G4-MODELVIRT is not applicable to solutions that do not use VMs).

provides a summary of the extent to which each type of solution meets these goals). Most of these solutions share the same two shortcomings: focusing on just one provisioning use case, and not fully modelling virtual resources.

As described earlier, particularly in the case of best-effort provisioning and advance reservations, different types of provisioning use cases tend to have conflicting measures of efficiency. Thus, it is not surprising that the solutions that have emerged over time have tended to focus on supporting one use case and supporting it well. In some cases, the provisioning abstraction itself precludes supporting multiple use cases; for example, datacenter-based solutions, such as clouds, do not allow resource consumers to specify availability constraints beyond immediate availability. In other cases, although the architecture might support multiple use cases, such as batch job schedulers that support both best-effort provisioning and advance reservations, these are rarely combined in practice because of the utilization problems they pose.

This raises the question of whether there would be any value in developing a system that is capable of supporting a general-purpose lease abstraction. Although specialization has resulted in impressive solutions (both experimental and in production), it has done so to the detriment of use cases with stricter resource requirements such as coscheduling of multiple resources [18, 61, 9], urgent computing applications [44], applications expressible as a workflow of independent tasks that can be executed more efficiently by multilevel scheduling methods [51, 59, 29], and service provisioning clouds like the one being built by the RESERVOIR project[14] [47], requiring reservation of cloud resources at specific times to meet service-level agreements or peak capacity requirements.

A solution that meets all the goals described earlier would allow these use cases to be supported more widely, while allowing resource providers to continue to support existing users. However, this poses the following research question:

14. http://www.reservoir-fp7.eu/

What are the benefits, disadvantages, and tradeoffs of a general-purpose resource leasing architecture?

In this dissertation, I explore this question specifically in the case where virtual machines are used to provision resources in such an architecture. Although some of the work discussed earlier has already explored the use of virtual machines for resource provisioning, most of these approaches use a basic resource model that neglects the overhead of deploying and running VMs and, even when deployment overhead is modelled and taken into account for scheduling, as in Nishimura et al. and Yamasaki et al., they focus only on immediate availability, and do not support workloads combining different types of leases. Thus, we also need to address the following question:

How can we control and mitigate the various overheads involved in using virtual machines as a resource provisioning mechanism?

2.5 Contributions to the State of the Art

In pursuit of an answer to the above research question, and of a solution to the resource provisioning problem that meets the four goals described at the beginning of this chapter, the main contribution of this dissertation are:

1. A resource provisioning model and architecture that uses leases as a fundamental abstraction and virtual machines as an implementation vehicle (Chapters 3 and 4).

2. Lease scheduling algorithms that mitigate the utilization problems typically encountered when scheduling advance reservations (Chapter 5)

3. A model for the various overheads involved in using virtual machines, and algorithms that (a) allow lease terms to be met even in the presence of this overhead, and (b) mitigate this overhead in some cases (Chapters 6 and 7).

4. Price-based policies for lease admission, showing that an adaptive pricing strategy can, in some cases, generate more revenue than other baseline pricing strategies, but does so by using fewer resources, thus giving resource providers more excess capacity that can potentially be sold to other users (Chapter 8)

As a technological contribution, I also present Haizea (http://haizea.cs.uchicago.edu/), an open source reference implementation of the architecture and algorithms described in this dissertation.

CHAPTER 3

RESOURCE AND LEASING MODEL

The fundamental resource provisioning abstraction in this work is the lease. Informally, a lease is a form of contract where one party –the lessor, or *resource provider*– agrees to provide a set of *resources* (an apartment, a car, computational resources, etc.) to another party –the lessee, or *resource consumer*– under a set of *terms* governing the amount of resources, access to the resources, the duration of the lease, etc. Although the lease abstraction has already been used in the context of computer systems, most notably in computer networks, there is no universally accepted definition of "lease."

Thus, this chapter provides a formal definition of what a lease is within the context of this work. Since the leasable resources in this work are computational resources, Section 3.1 starts by defining what resources are subject to be leased and provides a formal resource model, followed by a definition of leases and lease terms in Section 3.2.

3.1 Resource model

This work focuses on leasing computational resources from a within a single administrative domain, or *site*. The set of leasable computers, or *nodes*, in a site is denoted P. Each node $x \in P$ has *leasable resources* and *attributes*. Leasable resources within a node can be allocated to one or more leases, up to a maximum capacity, and may include processors, memory, disk space, network bandwidth, etc. The set of the types of leasable resources in a site is denoted Γ_R (e.g., $\Gamma_R = \{\mathsf{memory}, \mathsf{disk}, \ldots\}$). A specific quantity of a resource is denoted r, with the following fields:

1. $type[r] \in \Gamma_R$, the type of resource.

2. Some resources can appear multiple times within a same node, e.g., a node has only "one memory" but can have multiple processors. Instead of modelling each processor

as a separate resource, these types of resources are modelled as a single resource with multiple *instances*. Thus, $n[r] \in \mathbb{N}$ is the number of instances of the resource.

3. The quantity of the resource in each instance, $q_1[r]$, $q_2[r]$, ..., $q_{n[r]}[r]$. Every quantity is a positive integer. When $n[r] = 1$, the quantity is denoted $q[r]$.

Attributes are used to denote properties of a node that may be used for match-making (e.g., the node's architecture). The set of types of attributes in a site is denoted Γ_A (e.g., $\Gamma_A = \{\mathsf{arch}, \mathsf{vmm}, \ldots\}$. The domain of an attribute type τ_a is denoted $Dom(\tau_a)$ (e.g., $Dom(\mathsf{arch})$ could be $\{\mathsf{x86}, \mathsf{x86\_64}, \ldots\}$). An attribute a has the following fields:

1. $type[a] \in \Gamma_A$, the type of the attribute.

2. $value[a] \in Dom(type[a])$, the value of the attribute.

The set of all possible resource quantities is denoted R, and the set of all possible attributes is denoted A.

Every node $x \in P$ is defined with the following fields:

1. $res[x] \subseteq R$, the set of leasable resources in node x. There cannot be more than one resource of the same type in the same node ($\forall q \in res[x] \forall r \in res[x].q \neq r \Rightarrow type[q] \neq type[r]$)

2. $attr[x] \subseteq A$, the set of attributes in node x. There cannot be more than one attribute of the same type in the same node ($\forall q \in attr[x] \forall r \in attr[x].q \neq r \Rightarrow type[q] \neq type[r]$)

Some discussions in the following chapters will assume a site with homogeneous resources, such that every node has p processors, m MB of memory, d of disk space, bi MB/s of incoming bandwidth, and bo MB/s of outgoing bandwidth. The site is defined as follows:

$$\Gamma_R = \{\mathsf{proc}, \mathsf{mem}, \mathsf{disk}, \mathsf{net\text{-}in}, \mathsf{net\text{-}out}\}$$

$$(\forall x \in P)(\forall r \in res[x]) \quad (type[r] = \mathsf{proc} \rightarrow n[r] = p)$$
$$(\forall x \in P)(\forall r \in res[x])(\forall i \in \{1 \ldots p\}) \quad (type[r] = \mathsf{proc} \rightarrow q_i[r] = 1.0)$$

$$(\forall x \in P)(\forall r \in res[x]) \quad (type[r] \in \{\mathsf{mem}, \mathsf{disk}, \mathsf{net\text{-}in}, \mathsf{net\text{-}out}\} \rightarrow n[r] = 1)$$
$$(\forall x \in P)(\forall r \in res[x]) \quad (type[r] = \mathsf{mem} \rightarrow q[r] = m)$$
$$(\forall x \in P)(\forall r \in res[x]) \quad (type[r] = \mathsf{disk} \rightarrow q[r] = d)$$
$$(\forall x \in P)(\forall r \in res[x]) \quad (type[r] = \mathsf{net\text{-}in} \rightarrow q[r] = bi)$$
$$(\forall x \in P)(\forall r \in res[x]) \quad (type[r] = \mathsf{net\text{-}out} \rightarrow q[r] = bo)$$

3.2 Leases

A lease is *a negotiated and renegotiable agreement between a resource provider and a resource consumer, where the former agrees to make a set of resources available to the latter, based on a set of lease terms presented by the resource consumer.* More specifically:

Agreement: An arrangement between parties regarding the delivery of a service by one of the parties. In the context of this work, only two parties are involved —a resource provider and a resource consumer,— and the service to be provided is access to computational resources[1].

Terms: The specification of the resources to be provided by the resource provider to the resource consumer. The specific lease terms used in this work are described below.

Negotiable agreement: An agreement where the resource consumer must first propose a set of terms to the resource provider. In turn, the resource provider can accept, reject,

1. This definition is based on the definition provided in the WS-Agreement specification [2]: *"An agreement defines a dynamically–established and dynamically–managed relationship between parties. The object of this relationship is the delivery of a service by one of the parties within the context of the agreement. The management of this delivery is achieved by agreeing on the respective roles, rights and obligations of the parties. The agreement may specify not only functional properties for identification or creation of the service, but also non-functional properties of the service such as performance or availability. [...]. Agreement terms define the content of an agreement. It is expected that most terms will be domain-specific defining qualities such as for example service description, termination clauses, transferability options and others."*

or send back to the former an alternate set of terms it would be willing to accept. If the resource provider accepts the terms presented by the resource consumer, or if the resource consumer accepts the alternate terms presented by the resource provider, the agreement is established.

Renegotiable agreement: An agreement such that, once the terms have been agreed to by the resource consumer and resource provider, the former can request an alteration of the terms, without having to form a new agreement. The resource provider can accept or reject the new terms.

The lease terms encompass the *hardware resources* required by the resource consumer, such as CPUs, memory, and network bandwidth; a *software environment* required on the leased resources; and an *availability period* during which the requested hardware and software resources must be available. The terms of a lease l are specified through the following fields:

1. $nodes[l]$, a set of nodes requested by the user. Similarly to the nodes in P, each node $x \in nodes[l]$ has the following fields:

   (a) $res[x] \subseteq R$, the set of resources required for node x. There cannot be more than one resource of the same type in the same node ($\bigcap_{r \in res[x]} type[r] = \emptyset$)

   (b) $attr[x] \subseteq A$, the set of attributes required in node x. There cannot be more than one attribute of the same type in the same node ($\bigcap_{a \in attr[x]} type[a] = \emptyset$)

2. $reqstart[l]$, the requested starting time. This field can either be an exact time t, indicating that the lease must start no earlier than t, or can be left unspecified, indicating that the resource provider can set any starting time (as long as all the other lease terms are met).

3. $reqduration[l] \in \mathbb{N}$, the requested duration of the lease in seconds.

4. *deadline*[l], the time by which the lease must have been satisfied to its full duration. Can be left unspecified.

5. *price*[l], the price the resource provider expects the resource consumer to pay for this lease.

6. *user*[l], the resource consumer, or *user*, who requested this lease.

7. *preemptible*[l] $\in \{\mathsf{true}, \mathsf{false}\}$, indicating whether the resource consumer allows the lease to be preempted. When a lease is preempted, the resources allocated to the lease are freed up and reallocated in the future in the same state they were at the time they were preempted.

8. $software[l]$, the software environment. In this work, a software environment is specified in the form of a file with the entire contents of a disk, or *disk image*. A disk image *img* has a single field *size*[*img*], the size in bytes of the file. In cases where a resource consumer needs to reserve computational capacity in advance, but the required software environment is not yet known, this field can be left unspecified when the lease is requested.

   There are several other ways of representing a software environment, such as listing the names of software packages that would have to be installed in the software environment, and, in general, they could be used in this model as long as there is a way of estimating the time and resources required to deploy a given software environment.

3.2.1 Characteristic lease types

Throughout this dissertation, some discussions and analysis will revolve around certain characteristic types of lease:

Best-effort lease without deadline or, simply, a *best-effort lease*. A lease where both $requested[start[l]]$ and $deadline[l]$ are left unspecified.

Best-effort lease with deadline or *deadline lease*. A lease where $deadline[l]$ is specified. If $requested[start[l]]$ is specified, then:

$$deadline[l] - requested[start[l]] > requested[duration[l]]$$

i.e., there may potentially be several starting times for the lease such that the deadline is still met.

Advance reservation lease or *AR lease*. A lease where $requested[start[l]]$ and $deadline[l]$ are both specified, and

$$deadline[l] - requested[start[l]] = requested[duration[l]]$$

i.e., the only starting time that can satisfy the deadline constraint is $requested[start[l]]$, with the lease ending at exactly $requested[start[l]] + requested[duration[l]]$.

Immediate lease . A special case of an AR lease where $requested[start[l]]$ equals the time when the lease was submitted.

3.2.2 Internal lease attributes

The above fields are enough to express the hardware, software, and availability terms of a lease. In other words, a resource consumer requesting a lease would have to present values for all the above fields to a resource provider. However, a lease l has additional fields that are not part of the lease terms, but are necessary to discuss lease scheduling algorithms in the following chapters:

1. *state*[l], the state of the lease. These states are described in Section 3.2.3

2. *submission*[l], the time at which the lease was submitted by the resource consumer.

3. *actual_start*[l], the actual time at which the lease started.

4. *elapsed_duration*[l], the duration in seconds that the lease has accrued. In other words, this field keeps track of how much time the requested resources have been available to the resource consumer.

5. *end*[l], the time at which the lease ended.

6. Based on the lease terms presented by the resource consumer, the resource provider must allocate resources in P to satisfy those lease terms. A lease may include multiple allocations. For example, besides allocating the actual nodes requested in the lease, a lease could require transferring a disk image (containing a software environment) to a node, and resources must also be allocated for this operation. Thus, a lease has a set of allocations *allocations*[l], with each allocation *alloc* having the following fields:

   (a) *start*[*alloc*], the starting time of the allocation.

   (b) *end*[*alloc*], the ending time of the allocation.

   (c) *res*[*alloc*], a set of tuples of the form $(x, r_1, r_2, \ldots)$ where $x \in P$ and $r_i \in R$, representing the resources allocated in each node of P.

   Subsequent chapters describe the algorithms used to determine these allocations based on the lease terms.

3.2.3 Lease states

Before a lease is submitted to a resource provider, it is in a *New* state. Once it is submitted, the lease can either be accepted (*Pending*) or rejected (*Rejected*). When a lease is accepted,

the resource provider agrees to abide by the lease terms; however, in the *Pending* state, the resource provider has not yet allocated specific resources for that lease. Once a resource provider has done this, the lease is *Scheduled*. Once the resources become available and the resource consumer can access them, the lease is in an *Active* state, although it may have to pass through a *Preparing* state first if any actions have to be completed before the resources are ready (e.g., transferring a disk image to a node). If the lease is preemptible, it may be preempted while in the *Active* state. When this happens, all the lease's state information is saved (*Suspending*), the resources allocated to the lease are freed up and resources are allocated for it in the future, possibly migrating all the lease's state information to new nodes (*Migrating*). Before it can be active again, the lease's state information must be loaded again (*Resuming*). Once a lease is finished, it reaches the final state *Done*. The full state machine of leases is shown in Figure 3.1, and the description of the states and transitions is shown in Table 3.1.

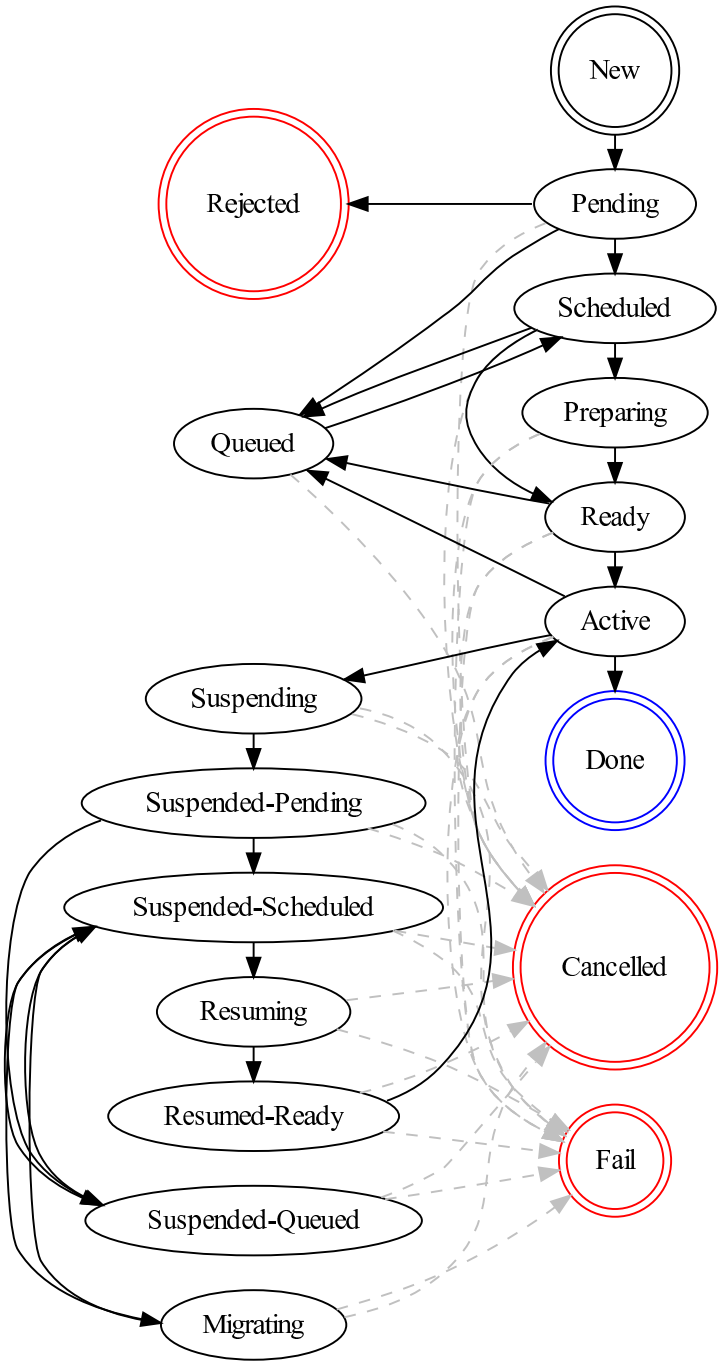


Figure 3.1: Lease state machine

Table 3.1: Lease states

		Transition to...	
State	**Description**	**State**	**Reason**
New	The initial state of all leases	Pending	Lease is submitted.
Pending	The resource consumer has received a lease request, but has not yet decided whether to accept it or reject it.	Scheduled	The resource provider accepts the terms, and can allocate resources for the lease right away.
		Queued	The resource provider accepts the terms, but cannot allocate resources for the lease right away. resources.
		Rejected	The resource provider rejects the terms.
Rejected	The terms of the lease have been rejected by the resource provider.		

Continued on next page...

Table 3.1 – continued from previous page

		Transition to...	
State	**Description**	**State**	**Reason**
Scheduled	Resources have been allocated for the lease in the future, but any preliminary actions required by the lease (e.g., transferring a disk image containing the software environment required by the lease) have not started yet.	Preparing	A preliminary action has started.
		Queued	The scheduled resources have been preempted, and the lease must be placed back on the queue.
		Ready	No preliminary actions are required for this lease.
Queued	The lease is waiting for resources to become available.	Scheduled	The resource provider has found resources for the lease.
Cancelled	The lease has been cancelled by the resource consumer		
Preparing	Preliminary actions, such as transferring a disk image, are being carried out for the lease.	Ready	Preliminary actions have finished.

Continued on next page...

Table 3.1 – continued from previous page

		Transition to...	
State	**Description**	**State**	**Reason**
Ready	Resources have been allocated for the lease in the future, and any preliminary actions have already been completed.	Active	The starting time of the allocated resources has been reached.
		Queued	The scheduled resources have been preempted, and the lease must be placed back on the queue.
Active	The resource consumer has access to the resources requested in the lease.	Suspending	The lease has been preempted and the resource provider has the ability to save the lease's complete state.

Continued on next page...

Table 3.1 – continued from previous page

		Transition to...	
State	**Description**	**State**	**Reason**
		Queued	The lease has been preempted and the resource provider lacks the ability to save the lease's complete state (which means the lease must be placed back on the queue).
		Done	The duration of the lease has expired.
Suspending	A lease's state is being saved.	Suspended-Pending	The lease's state has been saved.
Suspended-Pending	A lease has saved state, but the resource provider has not allocated resources where the lease can be resumed.	Suspended-Queued	The resource provider cannot find resources for the lease.
		Suspended-Scheduled	The resource provider can find resources for the lease.

Continued on next page...

Table 3.1 – continued from previous page

		Transition to...	
State	**Description**	**State**	**Reason**
Suspended-Queued	A lease has saved state, but is waiting for resources to become available.	Suspended-Scheduled	Resources become available.
Suspended-Scheduled	A lease has saved state, and the resource provider has allocated resources where the lease can be resumed.	Suspended-Queued	The allocated resources have been preempted.
		Migrating	The saved state must be migrated before the lease can be resumed.
		Resuming	The saved state can begin to be loaded.
Migrating	A lease's saved state is being migrated.	Suspended-Scheduled	The migration finishes.
Resuming	A lease's saved state is being loaded.	Resumed-Ready	The saved state has been loaded.

Continued on next page...

Table 3.1 – continued from previous page

		Transition to...	
State	**Description**	**State**	**Reason**
Resumed-Ready	A previously suspended lease has had its state loaded again, and resources for the lease have been allocated in the future.	Active	The starting time of the allocated resources has been reached.
Done	All the terms of the lease have been fulfilled.		
Fail	A failure has happened.		

CHAPTER 4

THE HAIZEA LEASE MANAGER

Haizea (http://haizea.cs.uchicago.edu/) is an open-source *lease manager* implemented in Python that has been developed as part of this dissertation. It can take leases (as described in the previous chapter), schedule them on physical resources, and send enactment commands (start VM, stop VM, etc.) to those physical nodes based on the lease schedule. Originally, Haizea was designed as an experimentation tool, providing a framework where multiple scheduling algorithms could be implemented and compared using a simulator. Nonetheless, Haizea eventually also became a drop-in scheduler for the open source virtual infrastructure manager OpenNebula (http://www.opennebula.org/).

In particular, Haizea can currently run in three modes:

Unattended simulation mode In this mode, Haizea takes a list of lease requests (specified in a file) and a configuration file specifying simulation and scheduling options (such as the characteristics of the hardware to simulate), and processes them in simulated time. Enactment commands are not sent to any physical hardware, and are assumed to complete successfully. The goal of this mode is to obtain the final schedule for a set of leases, without having to wait for all those leases to complete in real time (allowing explorations of the effect that a certain scheduling configuration could have over a period of weeks or months). The final result of an unattended simulation is a datafile with scheduling data and metrics that can be used to generate reports and graphs.

Interactive simulation mode The datafile produced in unattended simulation mode is not directly human-readable and instead meant for consumption by other programs (e.g., to generate graphs and reports). Thus, the unattended simulation mode can be cumbersome for getting immediate feedback on a scheduling action. In interactive simulation mode, enactment actions are still simulated, but Haizea runs in real time

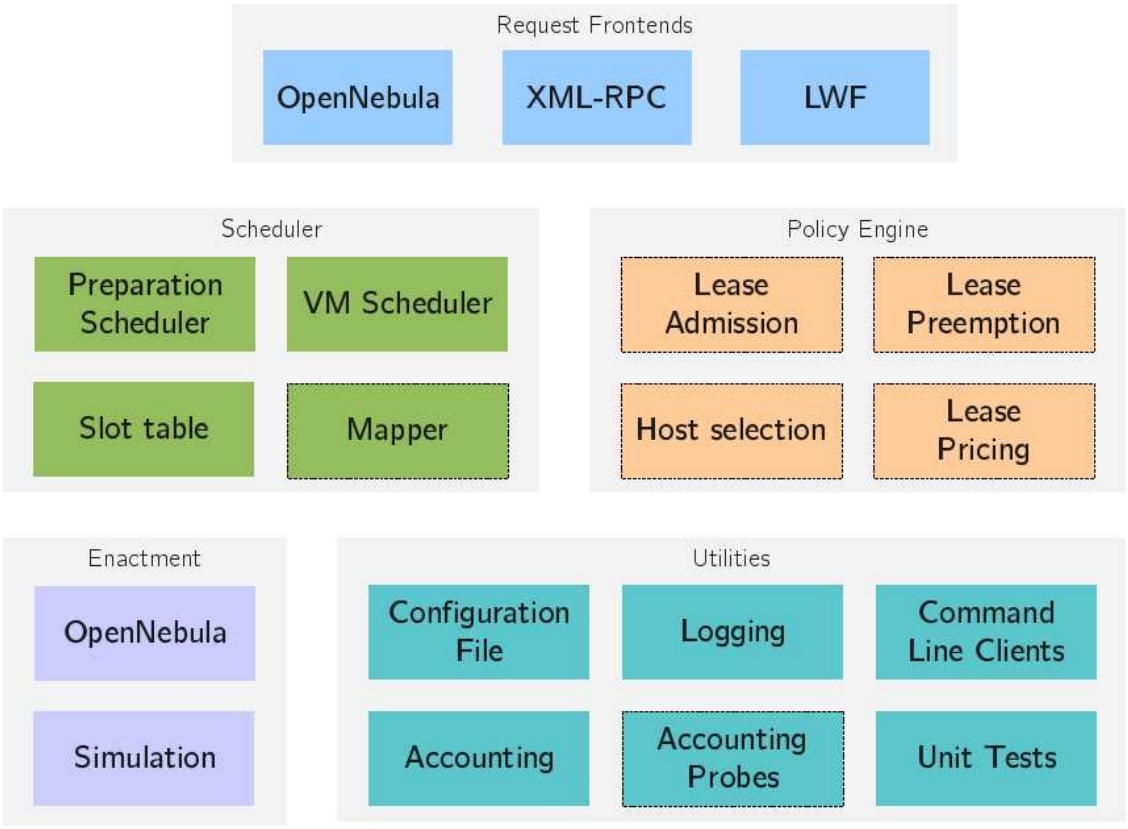


Figure 4.1: Haizea Architecture

and, instead of processing a prepared list of lease requests, Haizea provides a command-line interface to request leases interactively and query the status of Haizea's schedule (e.g., to find out the state of a lease).

OpenNebula mode OpenNebula is a virtual infrastructure manager that handles the deployment and configuration of virtual machines on a pool of physical resources. Although OpenNebula handles all the low-level details of managing distributed VMs (Xen, KVM, and VMWare VMs are currently supported), its default scheduler is designed with immediate scheduling in mind. Haizea can be used as a drop-in replacement for OpenNebula's scheduling daemon, providing support for advance reservations, preemption, deadlines, etc.

All the algorithms, models, and policies presented in the following chapters have been implemented on Haizea. All the experiments have also been carried out with Haizea (most in simulation, some in combination with OpenNebula). Thus, this chapter provides an introduction to the Haizea architecture, summarized in Figure 4.1. Haizea's architecture is divided into five major components, each of which is described in further detail in the remainder of this chapter:

Request Frontends To process and schedule a lease, the resource consumer must provide Haizea with a lease *request*. Haizea supports, and includes, multiple frontends for requesting a lease.

Scheduler This component contains all the scheduling code.

Policy Engine Certain scheduling decisions depend on resource provider-specific policies (e.g., a resource provider might have a policy of never accepting AR leases). Haizea includes a policy engine that allows a variety of leasing and scheduling policies to be implemented as pluggable modules, allowing new policies to be deployed without modifying Haizea's source code.

Enactment This component handles the enactment of the scheduling decisions made by the Scheduler component (e.g., sending a command to start a VM on a certain physical node).

Utilities This component contains miscellaneous utilities used by all the other components.

4.1 Request Frontends

Leases are requested through a request frontend. Haizea has been designed to support multiple frontends, and currently includes three frontends: the OpenNebula frontend, the XML-RPC frontend, and the LWF frontend.

4.1.1 OpenNebula

When running in OpenNebula mode, lease requests are not sent directly to Haizea. Instead, users request individual VMs using OpenNebula's command-line interface or its XML-RPC API, both of which expect certain basic information about the VM: the number of CPUs it needs, the kernel it requires, etc. This request is stored in OpenNebula's database, but is also accessible through OpenNebula's XML-RPC API. Haizea's request frontend uses this API to poll the requests received by OpenNebula, converting them to leases, scheduling them, and managing the leases' VMs using the OpenNebula enactment module (described in Section 4.4). Thus, in OpenNebula mode, users continue to request their VMs through OpenNebula, but Haizea handles all the scheduling decisions in the backend.

However, as mentioned earlier, the OpenNebula scheduler is designed with immediate scheduling in mind, resulting in an API that does not support specification of certain scheduling constraints natively, such as an advance reservation. To work around this, users can specify an additional `HAIZEA` parameter when requesting a VM from OpenNebula; this parameter, which can be used to specify additional scheduling constraints, is ignored by OpenNebula but passed along to Haizea. Figure 4.2 shows a sample OpenNebula request with an `HAIZEA` parameter.

4.1.2 XML-RPC

The XML-RPC frontend allows users to request leases and query their state using an XML-RPC API. Although this API can be accessed programmatically, Haizea also includes a command-line interface to it. The XML-RPC API is used in interactive simulation mode, and can be used in OpenNebula mode to query the state of leases but not to request leases. Figure 4.3 show a few sample commands and their output.

```
NAME = vm-example

CPU    = 1
MEMORY = 1024

OS = [
  kernel   = "/vmlinuz",
  initrd   = "/initrd.img",
  root     = "sda" ]

DISK = [
  source   = "/local/xen/domains/etch/disk.img",
  target   = "sda",
  readonly = "no" ]

NIC = [ mac="00:ff:72:17:20:27"]

HAIZEA = [
  start        = "+01:00:00",
  duration     = "00:30:00",
  preemptible  = "no"
]
```

Figure 4.2: Sample OpenNebula file with `HAIZEA` parameter

```
$ haizea-request-lease -f ar.xml
Lease submitted correctly.
Lease ID: 1
$ haizea-list-leases
 ID   Type          State      Starting time           Duration      Nodes
 1    AR            Scheduled  2009-08-04 11:25:57.00  00:10:00.00   1
```

Figure 4.3: Sample Haizea commands

```
<lease id="1" preemptible="true">
  <nodes>
    <node-set numnodes="4">
      <res amount="100" type="CPU"/>
      <res amount="1024" type="Memory"/>
    </node-set>
  </nodes>
  <start>
    <exact time="00:30:00.00"/>
  </start>
  <duration time="01:00:00.00"/>
  <deadline time="03:00:00.00"/>
  <software>
    <disk-image id="foobar1.img" size="1024"/>
  </software>
</lease>
```

Figure 4.4: Sample Haizea lease XML

4.1.3 LWF

The LWF (Lease Workload Format) is an Haizea-specific XML format that can be used to describe a workload of leases to be processed in unattended simulation mode. The main element in this format is the `<lease>` element, which provides an XML representation of leases as specified in the previous chapter (see Figure 4.4 for a sample XML representation of a lease). An LWF file contains a sequence of `<lease>` elements with additional information, such as the time at which the lease must be submitted when simulating the workload and information about the site to be simulated (Figure 4.5 shows a sample LWF file).

4.2 Scheduler

The scheduler is the main component of Haizea, and is responsible for scheduling leases. It is divided into two schedulers, one to schedule VMs on physical hosts, and one to schedule the preparation of a lease (e.g., transferring a disk image to a physical node).

```
<lease-workload name="sample">
  <description>
  Sample LWF file
  </description>

  <site>
    <resource-types names="CPU Memory"/>
    <nodes>
      <node-set numnodes="4">
        <res type="CPU" amount="100"/>
        <res type="Memory" amount="1024"/>
      </node-set>
    </nodes>
  </site>
  <lease-requests>

    <lease-request arrival="00:00:00.00">
      <lease ...>
         ...
      </lease>
    </lease-request>

    <lease-request arrival="00:20:00.00">
      <lease ...>
         ...
      </lease>
    </lease-request>

  </lease-requests>
</lease-workload>
```

Figure 4.5: Sample Haizea LWF

4.2.1 VM Scheduler

The VM scheduler is in charge of determining how a lease will map on to VMs and on what physical hosts those VMs will be deployed. The scheduling algorithms used in this module are described in the next chapter.

4.2.2 Preparation Scheduler

A lease may need to be prepared before it can start. In this dissertation, I focus on the scenario where disk images for a lease's VMs must be transferred to certain physical nodes. The preparation scheduler will schedule these transfers separately; the actual algorithms are described in Chapter 6.

4.2.3 Mapper

The mapper is used by the VM Scheduler to map VMs to physical hosts at a specific time (the VM scheduler is responsible for determining this time). The mapper is a pluggable module, meaning that developers can write custom mappers without modifying Haizea's source code. Haizea's default mapper uses a greedy algorithm described in the next chapter.

4.2.4 Slot Table

The slot table is one of the main data structures in Haizea. It contains the resource allocations of all the leases (i.e., $allocations[l]$, as defined in the previous chapter), and can thus be used to determine the current and future capacity of the physical nodes. The slot table is queried heavily by the VM scheduler and the mapper to determine if and when an allocation can be made for a lease. Since querying resource allocations is the most frequent operation in Haizea, the slot table provides $O(\log n)$ access to allocations starting or ending at a given time or falling within an interval of time (where n is the number of resource allocations).

This access time is currently achieved by storing allocations in two ordered lists (one ordered by starting time and another by ending time). Although using ordered lists results in $O(n)$ time for removals and insertions of allocations into the slot table, slot table queries far outweigh insertion and removal operations. Furthermore, lists (which are native data types in Python and implemented internally as C arrays) are also preferable as they allow faster iteration through an interval of allocations (once the first allocation at a given starting time is located) than $O(\log n)$ structures such as trees.

4.3 Policy Engine

Haizea includes a policy engine that allows certain scheduling decisions to be delegated to pluggable policies, allowing developers to write their own scheduling policies by writing a single Python class (implementing an interface for pluggable policies) which can then be plugged into Haizea without having to modify Haizea's source code. Four policies are currently pluggable: lease preemptability ("Can lease L_1 preempt lease L_2?"), host selection ("Given a VM, what physical node should it be assigned to?"), lease admission ("Given a lease, should it be accepted or rejected?"), and lease pricing ("Given a lease, what price should be assigned to it?").

4.3.1 Lease Admission

A lease admission policy determines whether a given lease request should be accepted or not by Haizea. Admission is distinct from whether a lease can be scheduled or not (although schedulability could be a part of the policy); the policy takes a lease in a Pending state (see Section 3.2.3) and decides whether the lease can be considered for scheduling or not. For example, a user could submit an AR lease that must start in 5 hours, but the policy could dictate that all ARs must be requested at least 24 hours in advance (and the lease would be rejected, regardless of whether there were resources available for it in 5 hours). Similarly, an

AR lease could be requested 48 hours in advance, be accepted by the lease admission policy, but then be rejected by the scheduler if there are no resources available.

4.3.2 Lease Preemption

Given a lease that needs to preempt resources (the "preemptor"), another lease (the "preemptee") that may be preempted by it, and a time, this policy determines if the preemptor can preempt the premptee. More specifically, this policy will assign a preemptability score to the preemptee, meaning some leases may be more preemptable than others. For example, a policy could be specified where AR leases can preempt any best-effort leases and preferring the leases that have been running the least. Thus, if an AR lease can only run by preempting either lease l_1, which has been running for 24 hours, or lease l_2, which has been running for only 5 minutes, the policy will determine that l_2 has to be preempted.

4.3.3 Lease Pricing

The lease pricing policy determines a lease's price ($price[l]$, as defined in the previous chapter). For example, a resource provider may determine the price of a lease based on the requested duration and number of nodes. Pricing policies are discussed in Chapter 8.

4.3.4 Host Selection

When the mapper maps VMs to physical hosts, this policy determines what hosts are more desirable. For example, an energy-saving policy might value hosts that already have VMs running (to leave as many empty machines as possible, which could then be turned off), whereas another policy might prefer empty hosts to make sure that VMs are spread out across nodes. Similar to the lease preemption policy, this ranking is done by assigning a score to each host.

```
[general]
loglevel: INFO
mode: opennebula
lease-preparation: unmanaged

[opennebula]
host: localhost

[scheduling]
policy-preemption: ar-preempts-everything
backfilling: aggressive
suspension: all
suspend-rate: 32
resume-rate: 32
suspendresume-exclusion: global
migration: no

[accounting]
datafile: /var/tmp/haizea/results.dat
probes: ar best-effort immediate cpu-utilization
```

Figure 4.6: Sample Haizea configuration file

4.4 Enactment

The enactment component is responsible for translating Haizea's scheduling decisions into concrete actions. For example, if an AR lease is scheduled to start at time t, the scheduler will use the enactment component to start the VMs for that lease. Haizea currently includes two enactment modules: an OpenNebula one and a simulation one. With the OpenNebula enactment module, Haizea will use OpenNebula's XML-RPC API to instruct OpenNebula to start, stop, suspend, and resume VMs for a lease. The simulation enactment module, on the other hand, does not manage real VMs. Instead, given an enactment request by the scheduler, it will always respond instantly that the request succeeded.

4.5 Utilities

The utilities component contains an assortment of modules that are used across Haizea. The *Configuration File* module takes care of loading Haizea's configuration file, where most of the scheduling options are specified (See Figure 4.6 for an example). The *Accounting* module

collects information while Haizea is running and saves it to a data file for off-line processing. The related *Accounting Probe* module allows developers to write their own "probes" to collect specific information when certain events happen (e.g., a developer can write a probe that is run every time a lease is completed to collect information about its final state). The *Logging* module provides various levels of logging, facilitating troubleshooting of problems without debugging the code. The *Command-line Clients* module contains all of Haizea's command-line clients. Finally, a *Unit Tests* module is used to test many of Haizea's modules.

4.6 Getting Haizea and Additional Documentation

Haizea is available for download at http://haizea.cs.uchicago.edu/. This site includes a manual (http://haizea.cs.uchicago.edu/manual/) providing instructions on how to install Haizea, and how to use it in the three modes (unattended simulation, interactive simulation, OpenNebula) described at the beginning of this chapter. This manual also provides a more detailed specification of Haizea's configuration file options, command-line interface and the LWF format.

CHAPTER 5

SCHEDULING LEASES WITH VIRTUAL MACHINES

So far, I have presented a leasing model (Chapter 3) and a leasing architecture (Chapter 4) without explaining how those leases would be scheduled, or exploring the benefits and drawbacks of using virtual machines to implement those leases. In this chapter, we begin that exploration, albeit making some assumptions that will be removed in later chapters.

As I have discussed previously (Section 2.1), virtual machines have a number of properties that make them an attractive vehicle for implementing leases. However, although they can be used to provision hardware, software, and availability (G2-HwSwAvail), we need to explore whether they can do so efficiently (G3-Reconcile) and overcoming the various overheads involved in using virtual machines (G4-ModelVirt). In this chapter, I show that using the suspend/resume/migrate capability of VMs can provide better performance (measured in terms of various metrics) than a scheduler that does not support preemption, and only slightly worse performance than a scheduler that does support preemption. I begin in Section 5.1 by describing how to schedule leases without deadlines, and without preempting leases. Next, Section 5.2 describes how preemption is incorporated into the scheduling, and discusses its possible benefits, and Section 5.3 describes the algorithm used to schedule with deadlines. Finally, Section 5.4 presents an experimental evaluation of the scheduling algorithms described in this chapter.

Throughout this chapter, I make three simplifying assumptions: (A) disk images for VMs are predeployed in the machines where they are needed and (B) there is at most one VM per physical machine, (C) a lease request is never rejected if there are enough resources to satisfy that request. I address these assumption in the following chapters.

5.1 Scheduling without deadlines or preemption

Given a lease l, each node n in $nodes[l]$ (each requiring $res[n]$ resources, such as CPU, memory, etc.) will be implemented as a virtual machine. Each VM, in turn, must be mapped to a physical node in P during a time interval. This time interval must meet the lease terms (e.g., a VM cannot be mapped to a physical node before $start[l]$) and there must be at least $res[n]$ available in the physical node during that interval.

Determining a time interval that meets the lease terms may require testing several different mappings. I present first the algorithm used in Haizea to attempt a mapping at a specific time interval, and then present how different types of leases use this mapping algorithm to find a time interval that meets the lease terms.

5.1.1 Mapping VMs to physical nodes at a specific time

To determine if it is possible to map all the VMs in a lease at a specific time interval, starting at a time t and lasting d seconds, Haizea first sorts the physical nodes in the following order:

1. First, the physical nodes with fewest leases scheduled on them at time t
2. Next, given two physical nodes with the same number of leases scheduled on them, pick the one with the largest available capacity at time t.
3. Finally, given two physical nodes with the same available capacity at time t, pick the one where that capacity remains unchanged the longest (i.e., this favours nodes where the same capacity is available during the entire interval from t to $t + d$)

The above is the default host selection policy (and the one used in all the experiments), although this sorting is configurable using Haizea's pluggable host selection policy (see Section 4.3.4). Once the nodes are sorted, Haizea uses a simple greedy algorithm to try to fit all the VMs, checking the physical nodes in the above order. Algorithm 1 shows the exact

Algorithm 1 Greedy mapping without preemption

Input: A lease l, a time t, a duration d

Output: A mapping, if one can be found, $nodes[l] \rightarrow P$ (from lease nodes to physical nodes), starting at time t and ending at $t + d$.

```
map ← empty dictionary
P' ← sort(P) {Sorted according to host selection policy }
cur_node ← First node in nodes[l]
for all p ∈ P' (while there are still nodes in nodes[l] left to map) do
  p_done ← false
  while not p_done do
    if cur_node fits in p from t to t + d then
      map_cur_node ← p
      cur_node ← Next node in nodes[l]
    else
      p_done ← false
    end if
  end while
end for
if all nodes in nodes[l] have been mapped then
  return map
else
  return ∅
end if
```

algorithm; note that this algorithm already accounts for the possibility of fitting multiple VMs in a physical host (in this case, the algorithm tries to fit as many VMs in a physical host before moving on to the next) even though the experiments in this section assume that there is only one VM per physical host.

5.1.2 Best-effort leases

Best-effort leases are scheduled on a FCFS (First Come First Serve) basis, with each requested best-effort lease placed at the end of a queue. When Haizea's scheduling function runs, the queue is inspected, starting at the head, in search of leases that can be scheduled at the current time. When a lease is found that cannot be scheduled, Haizea stops evaluating the queue.

However, FCFS by itself is known to be an inefficient strategy, and Haizea augments it with *backfilling* [37, 40, 15, 16], which allows certain requests to "jump the queue", as long as they do not delay the N leases at the head of the queue (called *aggressive backfilling* when $N = 1$, and *conservative backfilling* when N is the length of the queue). Preventing the head N request from being delayed is achieved by allowing at most N best-effort lease requests to be scheduled in the future (instead of only allowing them to be scheduled at the time the queue is being processed), and processing the entire queue each time the scheduling function runs.

Algorithm 2 describes how each lease request is scheduled when processing the queue. The boolean $allow\_future$ parameter is set to true if there are fewer than N best-effort leases scheduled in the future, and false otherwise. If a lease cannot be scheduled, it simply remains in a Queued state and placed back into the queue in its previous position.

5.1.3 AR and immediate leases

AR leases, on the other hand, require a simpler algorithm (see Algorithm 3), since there is only one mapping to test: between $start[l]$ and $duration[l]$. When an AR lease is requested, a mapping is attempted and, if the mapping is feasible, the lease is accepted; otherwise, it is rejected.

Immediate leases are handled as a special case of an AR lease, where $start[l]$ is set to the time the lease is requested.

5.2 Preemption with VMs

When combining best–effort and advance reservation leases, we can find that the time before an advance reservation is underutilized, as we cannot schedule any best-effort requests that would end after the scheduled start time of the advance reservation (see Figure 5.1, top diagram). Although backfilling partially palliates this by allowing lower-priority requests to

Algorithm 2 Best-effort

Input: A lease l, a boolean $allow\_future$
Output: A lease l

```
m ← map(l, now, duration[l]){See Algorithm 1}
if m ≠ ∅ then
    vmrr ← new reservation
    start[vmrr] ← now
    end[vmrr] ← now + duration[l]
    res[vmrr] ← ...
    Add vmrr to reservations[l] and to slot table.
    state[l] ← Scheduled
else if m = ∅ and not allow_future then
    state[l] ← Queued
else
    changepoints ← Time in the future in which there is a change in the slot table
    for all cp ∈ changepoints do
        m ← map(l, cp, duration[l])
        if m ≠ ∅ then
            break
        end if
    end for
    if m = ∅ then
        state[l] ← Queued
    else
        vmrr ← new reservation
        start[vmrr] ← cp
        end[vmrr] ← cp + duration[l]
        res[vmrr] ← ...
        Add vmrr to reservations[l] and to slot table.
        state[l] ← Scheduled
    end if
end if
return l
```

Algorithm 3 Advance reservation

Input: A lease l
Output: A lease l

```
m ← map(l, start[l], duration[l])
if m = ∅ then
    state[l] ← Rejected
else
    vmrr ← new reservation
    start[vmrr] ← start[l]
    end[vmrr] ← start[l] + duration[l]
    res[vmrr] ← ...
    Add vmrr to reservations[l] and to slot table.
    state[l] ← Scheduled
end if
return l
```

be scheduled before the advance reservation (see Figure 5.1, middle diagram), it can still result in some underutilization. Another alternative is to use the suspend/resume capability of VMs to suspend best–effort leases before an advance reservation, and resume them as soon as the advance reservation ends (see Figure 5.1, bottom diagram).

To make use of suspend/resume, the mapping algorithm (Algorithm 1) must also be able to look for a mapping that will last for less than the requested duration, e.g., when a lease can only be partially scheduled until the start of another lease, as shown in Figure 5.1. Additionally, the mapping algorithm must also look for mappings that involve preempting other leases. Algorithm 4 shows the updated algorithm, which now has an additional parameter $fulldur$, indicating whether the mapping must be for the full requested duration, and which tries to find mappings by preempting other leases, which are chosen according to the lease preemption policy (see Section 4.3.2)

The algorithms for best-effort leases (Algorithm 2) and advance reservation leases (Algorithm 3) now need to schedule the suspension and resumption of virtual machines if a preemption is taking place. During VM suspension and resumption, the entire state of the VM (mostly its memory) must be saved to disk, and this operation must be scheduled in

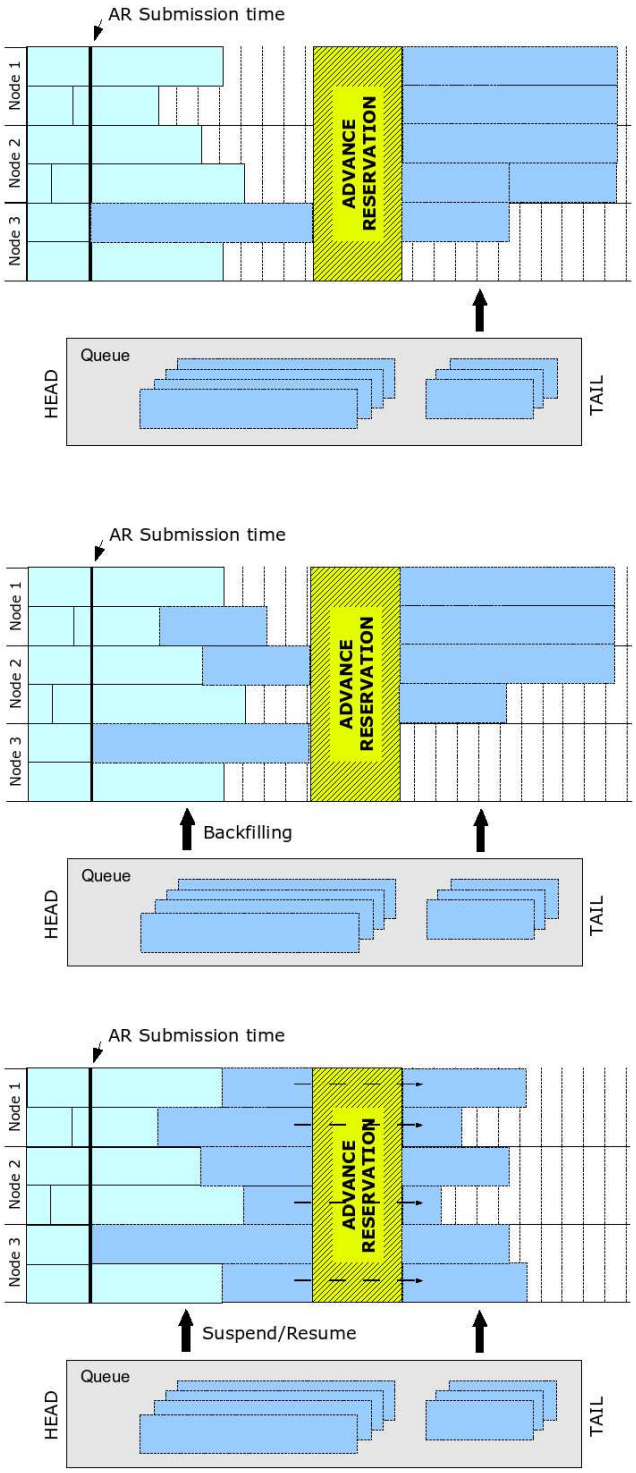


Figure 5.1: Underutilization before an advance reservation. Top: Draining nodes before an AR. Middle: Backfilling the time before an AR. Bottom: Suspending before an AR, and resuming after the AR.

Algorithm 4 Greedy mapping with preemption

Input: A lease l, a time t, a duration d, a boolean $fulldur$

Output: (1) A mapping, if one can be found, $nodes[l] \rightarrow P$ (from lease nodes to physical nodes), starting at time t and ending at $t+d'$, where d' must be exactly d if $fulldur = \mathsf{true}$, or a duration $d' \leqslant d$ if $full = \mathsf{false}$. (2) The duration d'. (3) A set *preempt* of preempted leases.

```
map ← empty dictionary
P' ← sort(P) {Sorted according to host selection policy (See Section 4.3.4)}
possible_preempt ← Preemptable leases scheduled between t and t+d {Sorted according to lease
preemption policy (See Section 4.3.2)}
preempt ← ∅
d' ← d
done ← false
found_mapping ← false
while not done do
  cur_node ← First node in nodes[l]
  for all p ∈ P' (while there are still nodes in nodes[l] left to map) do
    p_done ← false
    while not p_done do
      if cur_node fits in p starting at t then
        if fulldur and doesn't fit until t + d then
          p_done ← true
        else
          map_cur_node ← p
          if doesn't fit until t + d' then
            d' ← maximum duration cur_node will fit in p
          end if
          cur_node ← Next node in nodes[l]
        end if
      else
        p_done ← true
      end if
    end while
  end for
  if all nodes in nodes[l] have been mapped then
    done ← true
    found_mapping ← true
  else if |possible_preempt| > 0 then
    Extract first element of possible_preempt and add to preempt.
  else
    done ← true {But not valid mapping found}
  end if
end while
if found_mapping = true then
  return map, d', preempt
else
  return ∅
end if
```

such a way that it will not delay other leases (e.g., if we are suspending a best-effort lease to free resources for an AR lease, the suspension must finish before the start of the AR lease). At this point, I assume that this VM state is saved to the physical node's local filesystem at a rate of h_s megabytes of VM memory per second (h_r similarly for VM resumption). Since I also assume that each physical node runs at most one VM, this means a lease can be suspended by suspending all its $nodes[l]$ VMs in parallel, since each physical node will only have one VM to suspend, and each physical node's local filesystem is independent from the others. Thus, t_s and t_r, the time to suspend and resume an entire lease can be modelled by $\frac{mem[l]}{h_s}$ and $\frac{mem[l]}{h_r}$. Chapter 7 explores more complex models for estimating the time to suspend and resume a lease.

5.3 Scheduling with deadlines

Haizea schedules leases with deadlines by leveraging the previous algorithms. In particular, depending on the tightness of the deadline, the deadline scheduling algorithm (Algorithm 5) will try to schedule the lease either using the best-effort algorithm or the advance reservation algorithm. The tightness of a deadline is defined in terms of its slack, $\frac{deadline[l]-start[l]}{duration[l]}$ (the less slack, the tighter the deadline). If the slack is below a certain threshold, the algorithm first attempts to schedule the lease as an advance reservation starting at precisely $start[l]$ and allowing the lease to preempt other leases. If this fails (e.g., because no leases can be preempted without making them miss their deadlines), or if the lease is above the threshold, the algorithm tries to schedule the lease at the earliest time when resources are available without preemption and, if this results in a starting time that prevents the lease from meeting its deadline, it tries to alter the order of leases that have already been scheduled in a Least-Slack-First order.

The lease preemption policy is also modified to allow a lease to be preempted only if it can be rescheduled and still meet its deadline. Furthermore, when preemption is possible,

Algorithm 5 Scheduling a lease l with a deadline

$slack[l] \leftarrow \frac{deadline[l]-start[l]}{duration[l]}$
if $slack[l] \leqslant$ SLACK_THRESHOLD **then**
  Attempt to schedule at exactly $start[l]$ with preemption
  if unable to schedule **then**
    Attempt to schedule any time after $start[l]$ without preemption
  end if
else
  Attempt to schedule any time after $start[l]$ without preemption
end if
if unable to schedule **then**
  $ls \leftarrow$ Leases scheduled after $start[l]$
  Add l to ls
  Sort leases in ls from least to most slack. For each lease $l' \in ls$, the slack will be $\frac{deadline[l']-submit[l]}{duration[l']}$, since any rescheduled lease will have to start at least at $submit[l]$.
  Reschedule leases in ls without preemption
  if unable to reschedule all the leases **then**
    Restore previous schedule. Reject l
  else
    Accept l
  end if
else
  Accept l
end if

the scheduler will choose the leases with the highest slack, increasing the likelihood that they can be rescheduled before their deadlines.

5.4 Experimental Results (scheduling without deadlines)

The scheduling algorithms and policies described in this chapter were implemented in Haizea, which was used to simulate a month of lease requests and observe the effect of using these algorithms. This section describes the results when scheduling without deadlines.

5.4.1 Workloads

The workloads used in these experiments are constructed by adapting the SDSC Blue Horizon cluster job submission trace from the Parallel Workloads Archive [62]. In general terms, I take a set of job submission requests from that trace and treat them as a set of best-effort lease requests and then insert an additional set of advance reservation requests. Keeping the best-effort requests fixed, I vary the advance reservation requests to obtain a set of 72 different workloads.

More specifically, I take a 30 day extract of requests in the SDSC Blue Horizon trace starting at time 5:02:14:30[1] (this extract will be referred to as BLUE1 in subsequent chapters. For each of these 5,545 requests, I extract from the trace its submission time, requested duration, and requested number of nodes (in this 30-day extract, 66.86% of the requests have a requested duration of one hour or less, and 64.10% of the requests require four nodes or less,) and set the per-node resource allocation to $p = 1$ and $m = 1024$ (a single processor and 1024 MB of memory). Since traces provide both the requested duration of the job and its actual run time (which tends to be shorter than the amount requested by the user), Haizea also makes a note of the actual run time so the simulator will end a lease once that

1. This is essentially the first 30 days of requests in the trace, chosen arbitrarily; the first five days of the trace contain only twelve requests, and thus not useful for simulation purposes.

time has passed (the scheduler still bases its decisions only on the requested duration).

To generate the workloads, I then interleave with this set of best-effort requests a set of advance reservation requests, generated according to three parameters:

- ρ, the aggregate duration of all advance reservation leases in a workload, computed as a percentage of the total CPU hours in the simulation's run, which is the number of nodes multiplied by the time when the last best-effort request is submitted. I use the values ρ =5%, 10%, 15%, 20%, 25%, and 30%. (We do not explore larger values because the trace's utilization is 69.60%.)

- δ, the duration of each advance reservation lease, for which I use average values of 1, 2, 3, or 4 hours. (The duration is selected randomly from a range spanning $\delta \pm 30$m.)

- ν, the number of nodes requested by each lease, for which I use three ranges, from which the value is selected using a uniform distribution: *small* (between 1 and 24), *medium* (between 25 and 48), or *large* (between 49 and 72).

Given values for ρ, δ, and ν, I then determine the arrival times of the advance reservation requests as follows. First, I determine the number of requests that will be generated, and divide that number into 30 days to obtain an average interlease interval i. Then, I choose the individual intervals between requests at random in the range $(i - 1 \text{ hour}, i + 1 \text{ hour})$. Thus, the smaller the average lease duration, the more frequent is the arrival of requests (since there will be more advance reservation lease requests). Similarly, the smaller the average number of nodes, the higher the frequency. I further constrain advance reservation lease requests to involve an advance notice of exactly 24 hours. As with the best-effort requests, the advance reservation lease requests have a per-node resource allocation of $p = 1$ and $m = 1024$ MB.

In these experiments, I explore every combination of the parameters ρ, δ, and ν, for a total of 72 workloads. I refer to workloads using the notation [ρ%/δH/ν] (e.g., [10%/2H/medium]).

5.4.2 Simulated Physical Resources

The simulated pool of physical resources that leases will be scheduled on is modelled after the SDSC Blue Horizon cluster. It comprises 144 single-CPU nodes, each with 40 GB of disk and 1 GB of memory, connected with a switched Ethernet network (1000 Mb/s, I conservatively assume a 100 MB/s bandwidth). For the purposes of estimating the time required to suspend and resume a VM, I assume h_s and h_r to be 50 MB/s, based on results presented by Fallenbeck et al. [12]. I conservatively assume that the sum of the boot-up and shutdown time of a VM does not exceed 20 seconds. To account for the slowdown produced by running inside a VM, I assume that any computation running inside a VM requires 5% more time to run. Furthermore, I assume that the time required to send commands from the resource manager to the nodes is negligible and that the hardware will not behave erratically, and do not inject hardware failures into the simulated cluster.

5.4.3 Experiments

Each of the workloads described above was simulated using Haizea with the following configurations:

NOVM–NOSR — No VM, no suspend/resume: Leases do not run on VMs, so there is no VM image to transfer and no 5% runtime slowdown. In addition, leases cannot be suspended or resumed; thus, a preempted lease is cancelled and requeued.

NOVM–SR — No VM, with suspend/resume and migration: Like NOVM–NOSR, but leases can be suspended and resumed. This configuration represents a job scheduler capable of checkpointing and migrating any job. I assume that, to be able to checkpoint any application, system-level checkpointing is used, and that suspending a lease requires saving the entire memory to disk, as would happen in the VM cases.

VM–PREDEPLOY — With VM, single predeployed image: The leases run on VMs and all leases use the same VM image, which is assumed to be predeployed on the nodes.

During the experiments, I observe, for each lease, the times t_a (the arrival time, or time when the lease request is submitted), t_s (the start time of the lease) and t_e (the time the lease ends). At the end of an experiment, I compute the following metrics:

all-best-effort: I define all-best-effort as the time from the start of the simulation to the moment the last best-effort request is completed. I normalize this value by presenting the relative difference between this time and the time required to run all the best-effort requests *without* advance reservation leases in configuration NOVM–NOSR (this time is 2,668,432 seconds, or roughly 30.8 days). Thus, a value x indicates that an experiment took $2668432 \cdot x$ to run (with $x = 1.0$ meaning that the experiment took the same time as the baseline case).

Wait time of best-effort requests: I define wait time as $t_s - t_a$, the time a best-effort request must wait before it starts running.

Bounded slowdown of best-effort requests [17]: If t_u is the time the lease would take to run on a dedicated physical system (i.e., not in a VM), the lease's slowdown is $\frac{t_e - t_a}{t_u}$. If t_u is less than 10 seconds, the bounded slowdown is computed the same way, but assuming t_u to be 10 seconds [17].

When computing the last two metrics, I discard the first 5% of measurements, to avoid ramp-up effects. I retain the ramp-down period because it is where the requests that languish in the queue (until there are no more advance reservation leases) will finish.

Figure 5.2 shows the all-best-effort results for all experiments. This metric provides a good measure of utilization, taking into account both VM runtime slowdown and the overhead of VM deployment. Using suspend/resume and migration (with and without VMs)

results in a shorter run time than NOVM–NOSR in every case. In fact, using suspend/resume and migration produces a slowdown of, at most, 6.00% relative to not injecting any advance reservation leases at all, whereas not using suspend/resume can produce a slowdown of up to 29.36%. Additionally, the duration of the advance reservation leases is more likely to affect this metric in the NOVM–NOSR configuration, with shorter-duration (and thus shorter-interval) leases producing the worst results.

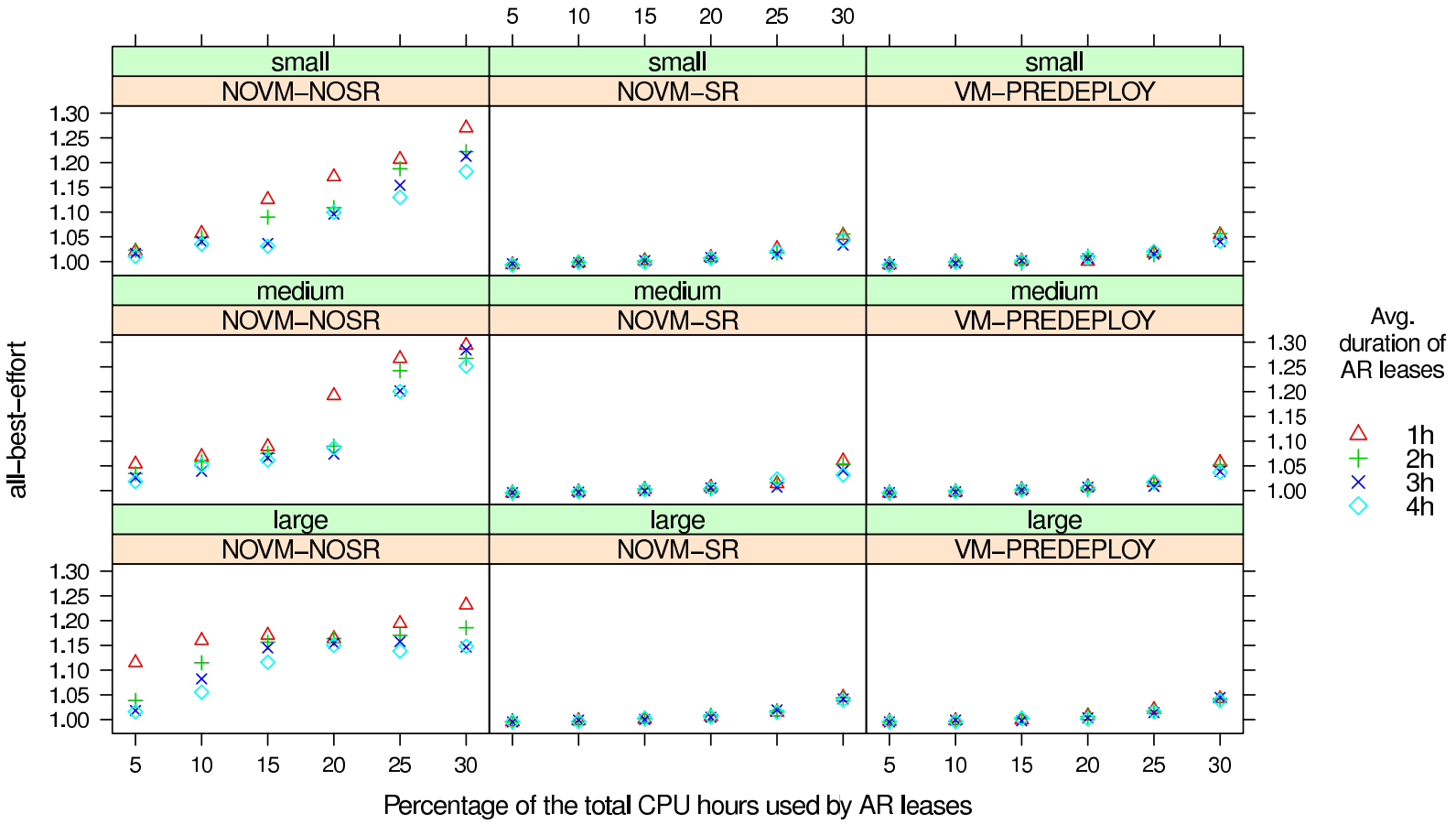


Figure 5.2: Each data point in this graph represents the all-best-effort metric in each experiment. The x axis represents ρ and the y axis represents the value of all-best-effort (normalized as described in the text). The graphs are grouped by ν (the number of nodes requested by each lease) and the experiment configuration. The symbol of each point denotes the value of δ (see legend).

While all-best-effort gives a good measure of effective utilization (indicating how much faster an entire workload will be run from beginning to end), it does not say much about individual leases, which requires looking at the other two metrics. For example, an inspection of the individual waiting times reveals that the longer total run time when not using suspend/resume (in NOVM–NOSR) is due to best-effort leases that remain in the queue and are not run until the ramp down period, when no more best-effort leases are being submitted. These leases remain in the queue because the advance reservation leases prevent them from being scheduled, especially when the interval between advance reservation leases is short. I constrain most of my discussion to the cases [10%/3H/medium], [20%/2H/medium], and [30%/1H/medium], which are representative of the trends that I observe across all cases.

Table 5.1 shows the total run time, average wait time, and average slowdown for these three cases. In every case, the NOVM–NOSR configuration results in longer running times, longer average wait times, and larger average slowdowns. I attribute this result to the fact that, without suspend/resume, the scheduler must rely heavily on backfilling to use the time and space efficiently before a blocking lease (such as an advance reservation lease). This behavior will favor short leases, which "skip to the front of the queue" when used as backfill. Since the majority of best-effort leases in the BLUE1 workload are shorter than one hour, they are ideal candidates for backfilling. However, suspend/resume need not look ahead for shorter best-effort leases when backfilling: it can simply take the next best-effort lease, even if long, knowing that it can suspend this lease if it has not run to completion before a blocking lease is scheduled.

I can support this observation by looking at how wait times and the slowdown vary with requested duration and number of nodes. Figures 5.3, 5.4, and 5.5 show the regression curves (using the Lowess smoother [8]) for these metrics and variables for [20%/2H/medium]. Figure 5.3 shows how wait time and slowdown vary with the requested lease duration and, in particular, how short leases in the NOVM–NOSR configuration have shorter wait times and

Table 5.1: Experiment running times, average waiting times, and average slowdowns for [10%/4H/medium], [20%/3H/medium], and [30%/2H/medium]

Time to complete best-effort leases, relative to time without advance reservation leases

$\rho \rightarrow$	10%	20%	30%
NOVM–NOSR	5.15%	7.37%	26.71%
NOVM–SR	0.46%	1.26%	6.09%
VM–PREDEPLOY	0.46%	1.40%	5.99%

Average waiting time for best-effort leases, in thousands of seconds

$\rho \rightarrow$	10%	20%	30%
NOVM–NOSR	14.05	24.73	86.50
NOVM–SR	6.24	10.76	26.00
VM–PREDEPLOY	6.18	10.88	27.43

Average bounded slowdown

$\rho \rightarrow$	10%	20%	30%
NOVM–NOSR	56.55	90.65	318.56
NOVM–SR	28.05	51.40	134.46
VM–PREDEPLOY	24.66	47.27	121.33

smaller slowdowns than all other configurations (which use suspend/resume). However, the tendency is for both the wait time and slowdown to increase as the requested duration increases since, as noted above, backfilling favors short requests. When using suspend/resume, on the other hand, the trend is for the wait times to not vary with the requested duration and for the slowdowns to exhibit a slight decreasing trend. Thus, all requests are treated more fairly, although the overall average does increase (the shorter requests, which make up the majority, have longer wait times because they no longer "skip the queue," thanks to backfilling). This is a desirable effect in many scenarios as it provides a more "democratic" treatment for different application types.

Figure 5.4 shows how the wait time and slowdown vary with the number of requested nodes. When looking just at the effect of node counts, adding suspend/resume has little

effect. In fact, the same trend is present across all configurations: wait time and slowdown do not vary when the node count is less than ten but tend to increase for larger node counts. Figure 5.5 shows how the wait time and slowdown vary with the requested CPU time (the product of the requested duration and the requested number of nodes), with trends similar to those when looking just at the requested duration, except that the upward trend as the CPU time increases is more pronounced in the NOVM–NOSR configuration. This upward trend is also evident in the suspend/resume configurations, but only for large values of CPU time.

The results for [10%/3H/medium] are shown in Figures 5.6, 5.7, and 5.8, and the results for [30%/1H/medium] are shown in Figures 5.9, 5.10, and 5.11. In general, we observe across all 72 workloads that, as ρ increases, wait times and slowdowns tend to increase more sharply in the NOVM–NOSR configurations, but tend not to vary in the configurations that use suspend/resume.

5.5 Experimental Results (scheduling with deadlines)

To evaluate the deadline scheduling algorithm, I once again used Haizea to simulate a month of lease requests to observe the effect of using these algorithms. These results are separate from the ones presented in the previous section, and use different workloads and metrics.

5.5.1 Workloads

I used two workloads in these experiments, based on job traces from the Parallel Workloads Archive[2] where certain attributes (such as the deadline) are added according to well-specified distributions.

2. http://www.cs.huji.ac.il/labs/parallel/workload/

1. SDSC Blue Horizon[3], or blue: 30 days of requests starting at request #218890[4]. This extract will be referred to as BLUE2 in the following chapters. The utilization of this workload is 84.48%. When using this workload, I simulate a site modelled after the trace's original cluster, with $P = 144$, $C = 1$, $M = 1024$ MB, $h_s = h_r = 64$ MB/s.

2. SDSC DataStar[5]: 30 days of requests starting at request #5043[6]. This extract will be referred to as DS in the following chapters. The utilization of this workload is 60.45%. When using this workload, I simulate a site modelled after the trace's original cluster, with $P = 164$, $C = 1$, $M = 1024$ MB, $h_s = h_r = 64$ MB/s.

Each job request in the trace is converted to a lease requesting the same duration and number of nodes as the original job request. Since the original requests are for best-effort jobs that do not request a specific starting time or have a deadline, I annotate each request with a starting time, a deadline, and a value for r_u.

I generate the starting time and deadline using two parameters, δ and ω:

$$start[l] = submit[l] + \delta$$

$$deadline[l] = start[l] + \omega + duration[l]$$

δ is the delay, in seconds, before the lease can start and ω is the maximum waiting time in seconds. When $\omega = 0$, the provider *must* schedule the lease exactly between the requested start time and deadline, whereas when $\omega > 0$, the provider has some flexibility when deciding what the actual start time of the lease will be, as long as the lease is completed before the

3. http://www.cs.huji.ac.il/labs/parallel/workload/l_sdsc_blue/index.html

4. Chosen randomly using Python's random number generator

5. http://www.cs.huji.ac.il/labs/parallel/workload/l_sdsc_ds/index.html

6. Chosen randomly using Python's random number generator.

deadline.

5.5.2 Parameters

A single experiment run involves simulating the scheduling of one of the two workloads. In each experiment run, I vary the following parameters:

- **The distribution of values of** δ. Each request in the workload is assigned a separate value between 0 and 86,400 (24 hours). I use three distributions of values:
  - EARLY START: Pareto distribution skewed toward 0.
  - UNIFORM START: Uniform distribution of values.
  - LATE START: Pareto distribution skewed toward 86,400.
- **The distribution of values of** ω. Each request in the workload is assigned a separate value between 0 and 604,800 (seven days). We use three distributions of values:
  - TIGHT DEADLINE: Pareto distribution skewed toward 0.
  - UNIFORM DEADLINE: Uniform distribution of values.
  - LOOSE DEADLINE: Pareto distribution skewed towards 604,800.
- **Preemption**: NO PREEMPTION, PREEMPTION WITHOUT SUSPEND/RESUME and PREEMPTION WITH SUSPEND/RESUME

The SLACK_THRESHOLD parameter in Algorithm 5 is set to 2.0 (i.e., if $deadline[l] - start[l]$ is less than twice the requested duration of the lease, we attempt to schedule the lease with preemption).

5.5.3 Metrics

In each experiment run I measure the following information:

- *Number of leases accepted*, *number of leases rejected by provider* because the lease could not be scheduled, and *Number of leases rejected by user* because the provider quoted a price that was too high for the user.

- *Resource utilization*, measured as percent of physical nodes that were used throughout the experiment run. To avoid ramp-up and ramp-down effects, the calculation omits the initial 5% time of the experiment run, and stops when the last request is submitted.

5.5.4 Results

The results focus on the effect that deadline tightness has on resource utilization, with and without using VM suspend/resume. Figure 5.12 shows the utilization for each workload and each distribution of values of δ and ω. In both workloads, as deadlines get tighter, utilization decreases as it becomes harder for the scheduler to accommodate the deadlines. Utilization also decreases, to a lesser extent, when later starting times are requested as (1) this further constrains the choice of resources for a lease, and (2) when a lease ends prematurely, there will be fewer leases that can be rescheduled to use the freed-up resources (since they are constrained to start at a later time).

Unlike the experiments where "extra work" was added in the form of advance reservation leases, the use of preemption has a relatively small effect on utilization, but a closer look at the characteristics of the accepted leases reveals that preemption favours leases with tighter deadlines. Table 5.2 shows the number of tight-deadlined leases (according to two criteria) accepted in three representative configurations. In the most constrained case, with Late Start and Tight Deadline, the use of preemption can increase the number of accepted leases from 51.68% to 78.09% when the leases' slack is below the scheduler's SLACK_THRESHOLD parameter. Although the effect of using VM suspend/resume is negligible, this result is consistent with the results show in the previous section, where VM suspend/resume is mostly beneficial when using high-utilization workloads (particularly in

Table 5.2: Effect of preemption on leases with tight deadlines

Workload	Start	Deadline	No Pr	Pr without S/R	Pr with S/R	Out of...
BLUE2	Late Start	Tight Deadline	597	873	902	1,155
	Uniform Start	Uniform Deadline	86	132	128	192
	Early Start	Loose Deadline	13	18	18	29
DS	Late Start	Tight Deadline	920	1,298	1,287	2,256
	Uniform Start	Uniform Deadline	129	262	270	468
	Early Start	Loose Deadline	22	29	33	50

(a) Number of accepted leases with slack ($\frac{deadline[l]-start[l]}{duration[l]}$) less than or equal to SLACK_THRESHOLD (2.0 in our experiments). The Out of... column is the number of leases requested with slack $\leqslant 2.0$.

Workload	Start	Deadline	No Pr	Pr without S/R	Pr with S/R	Out of...
blue	Late Start	Tight Deadline	2,553	2,973	3,049	3,387
	Uniform Start	Uniform Deadline	632	732	739	839
	Early Start	Loose Deadline	75	81	80	94
ds	Late Start	Tight Deadline	1,987	2,391	2,401	3,467
	Uniform Start	Uniform Deadline	400	569	580	774
	Early Start	Loose Deadline	51	62	63	74

(b) Number of accepted leases with $\omega \leqslant 120,960$ (20% of 604,800, the maximum value of ω). The Out of... column is the number of leases requested with $\omega \leqslant 120,960$

the presence of ARs). Since the utilization of the BLUE2 and DS workloads does not reach 100%, and are further constrained by deadlines, a high-utilization situation is not reached in these experiments.

5.6 Conclusions

This chapter has presented how leases can be scheduled and implemented using VMs, exploring the benefits and drawbacks of this approach, particularly when using the suspend/resume capability of VMs. The experimental results show that, when using workloads that combine best-effort and advance reservation leases, a VM-based approach with suspend/resume can overcome the utilization problems typically associated with the use of advance reservations. These results show that, even in the presence of the runtime overhead resulting from using VMs, a VM-based approach results in consistently better total execution time than a scheduler that does not support preemption, and only slightly worse performance than a scheduler that does support preemption. Measurements of wait time and slowdown for the same experiments show that, although the average values of these metrics increase when using VMs, this effect is due to short leases not being preferentially selected as backfill. In effect, a VM-based approach does not favour leases of a particular length over others, unlike systems that rely more heavily on backfilling. Additionally, the results have shown that, when scheduling leases with deadlines, the use of VM suspend/resume has a negligible effect on utilization, but allows a larger amount of leases with tight deadlines to be accepted.

Thus, this confirms that VMs are an adequate vehicle for implementing leases, reconciling the requirements of each type of lease (goal G3-Reconcile). However, the results at this point depend on a series of simplifying assumptions: VM disk images are predeployed, and there is at most one VM running in each physical node. Although these assumptions are not unrealistic, they don't meet goal G4-ModelVirt (virtual resources and, in particular, the overheads involved in using VMs, must be modelled accurately). The next two chapters

focus on meeting goal G4-ModelVirt by removing these assumptions.

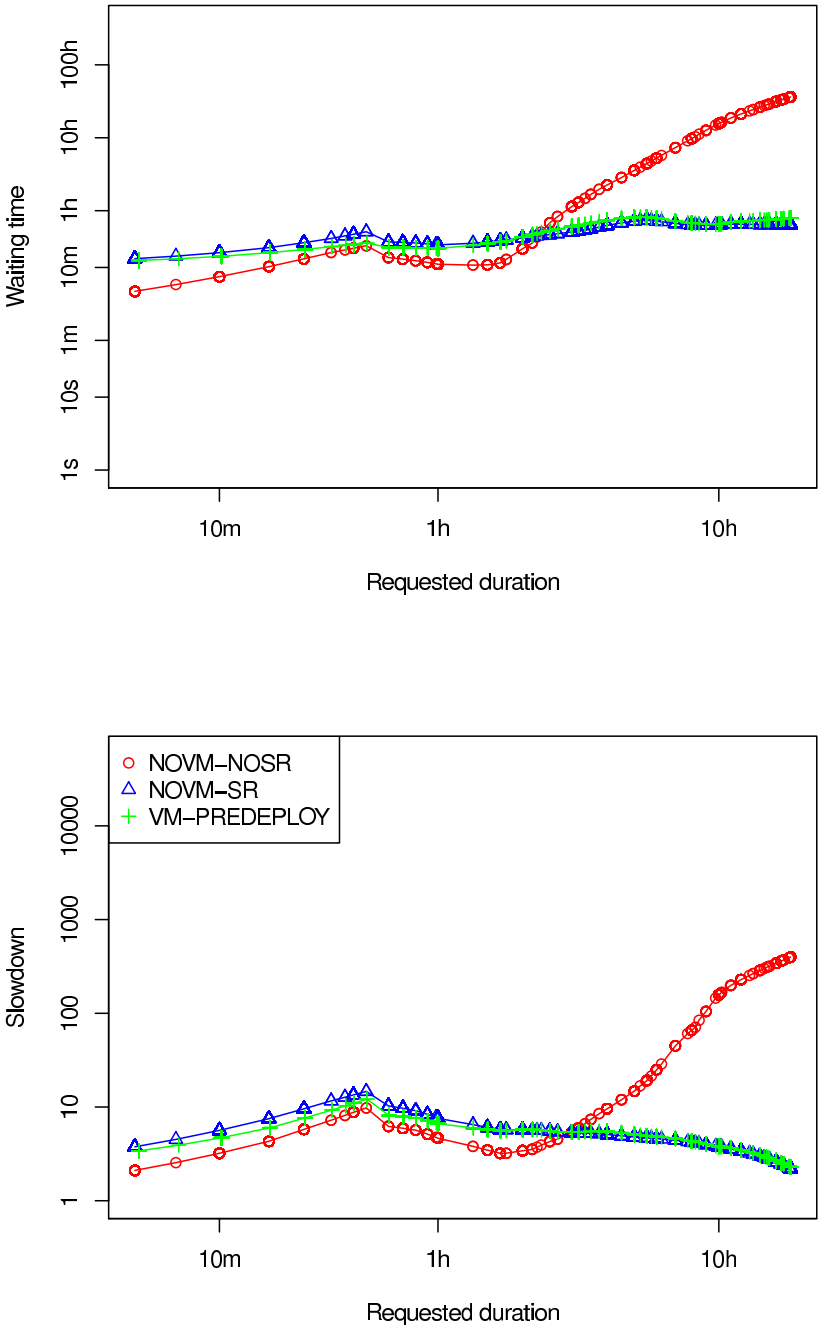


Figure 5.3: Effect of requested duration on waiting time and slowdown in [20%/3H/medium]

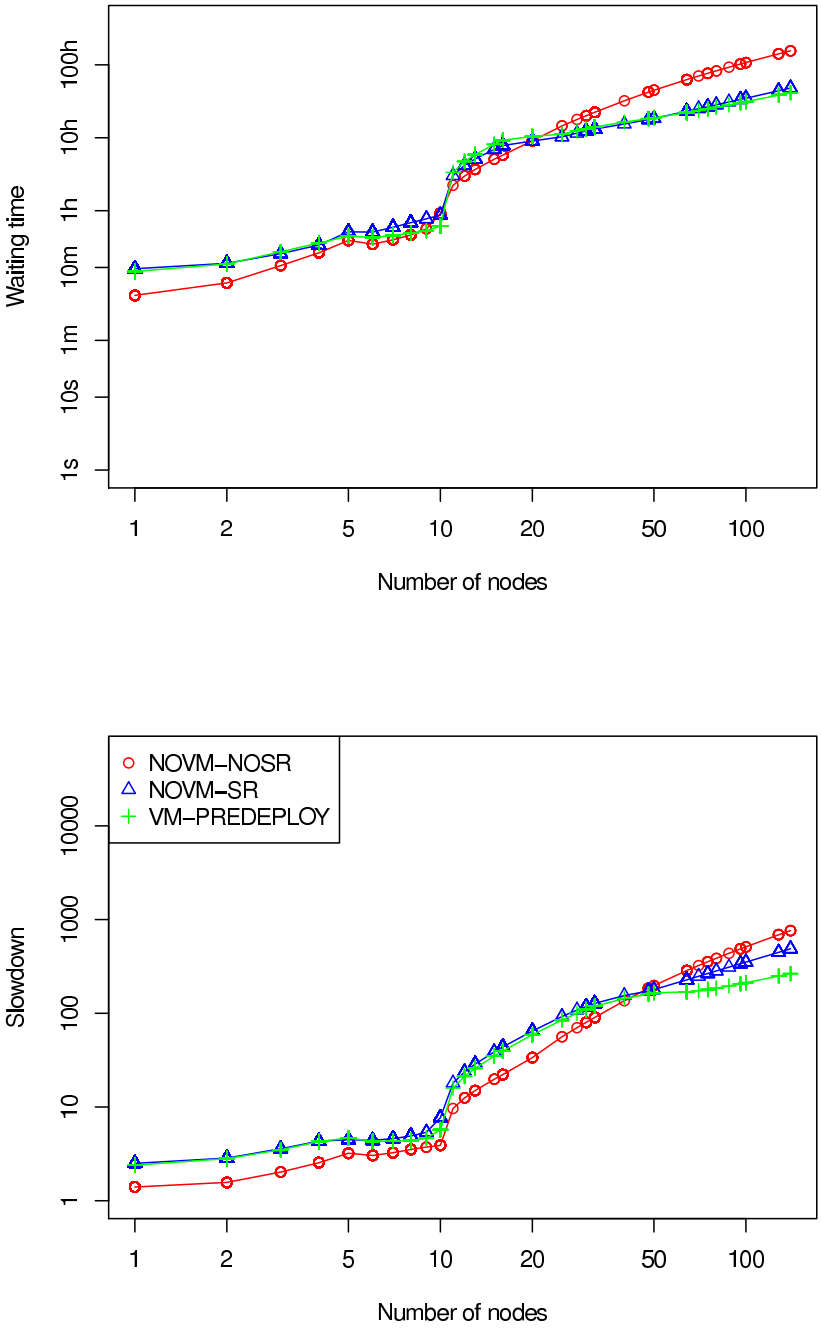
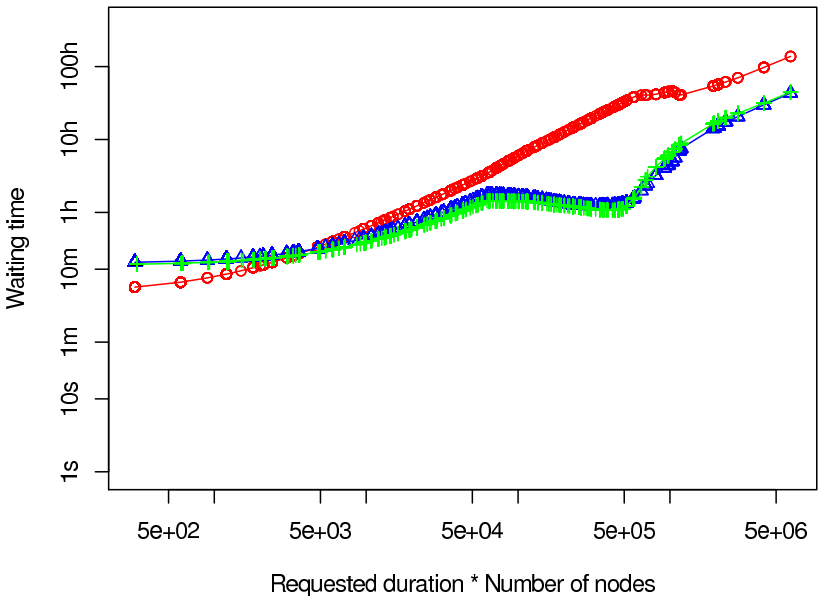


Figure 5.4: Effect of number of nodes on waiting time and slowdown in [20%/3H/medium]



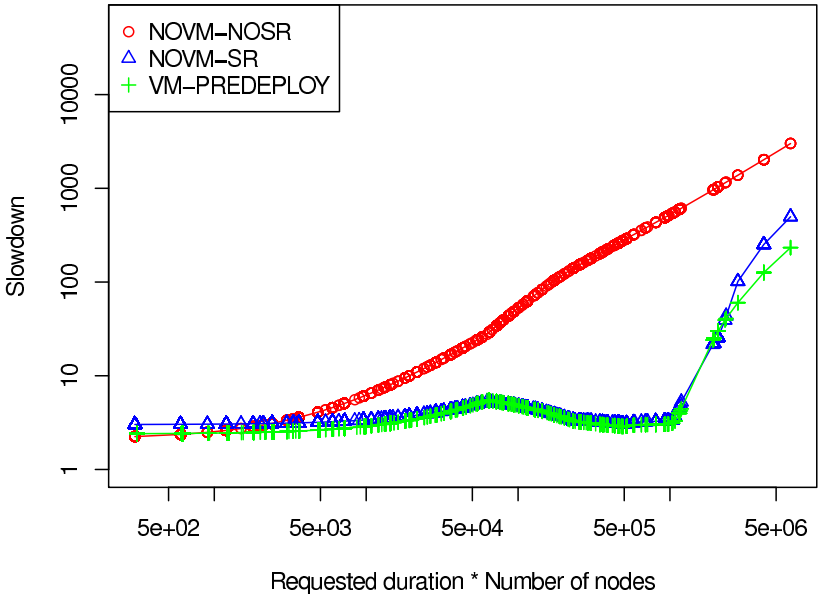


Figure 5.5: Effect of requested CPU hours on waiting time and slowdown in [20%/3H/medium]

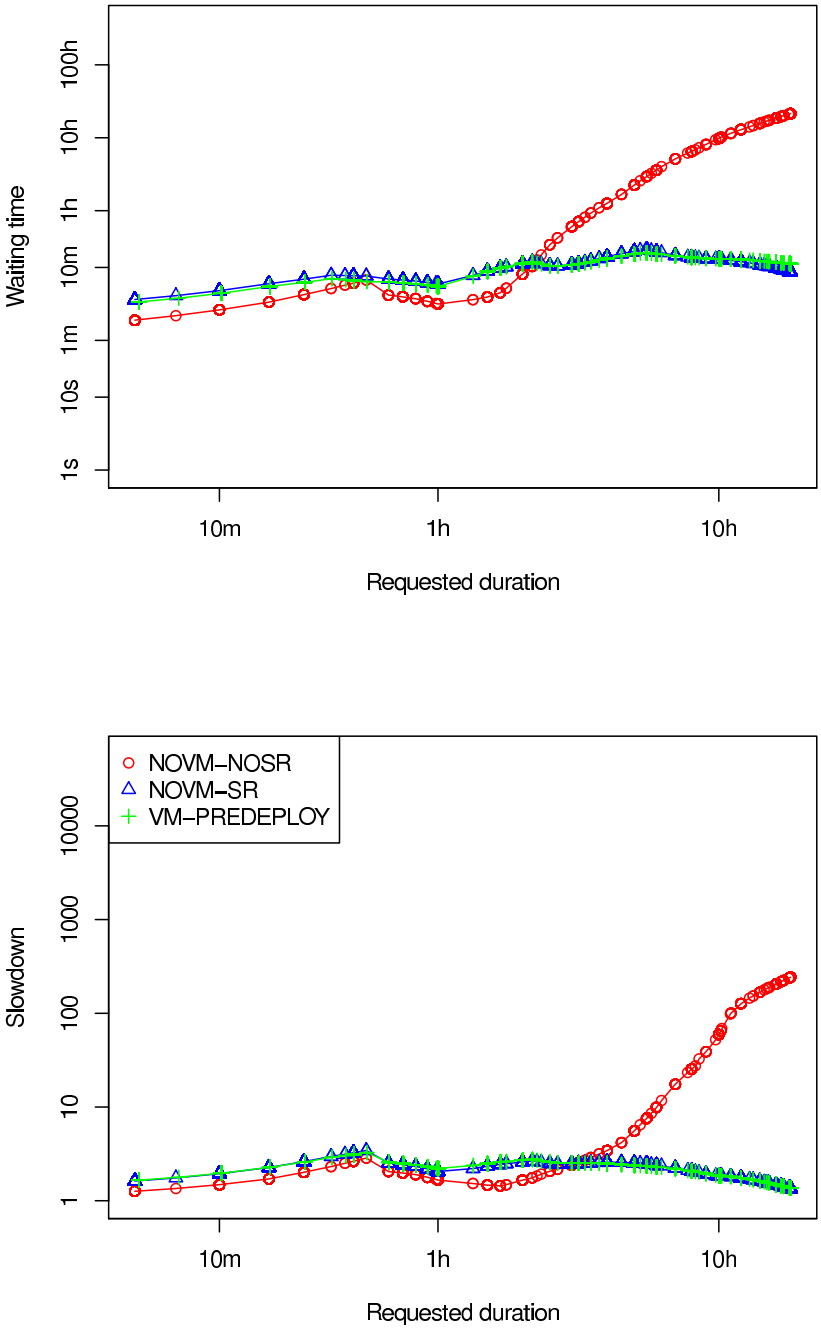


Figure 5.6: Effect of requested duration on waiting time and slowdown in [10%/4H/medium]

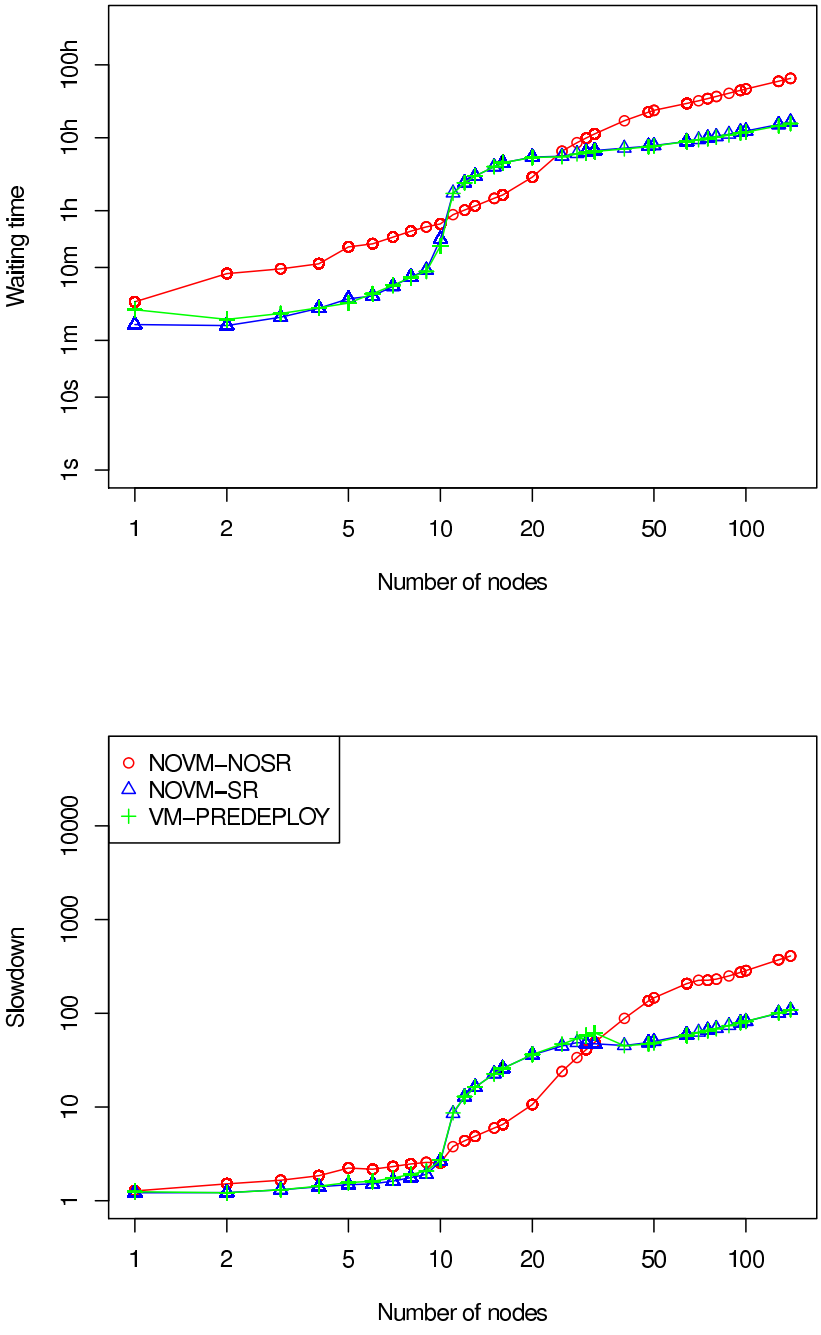
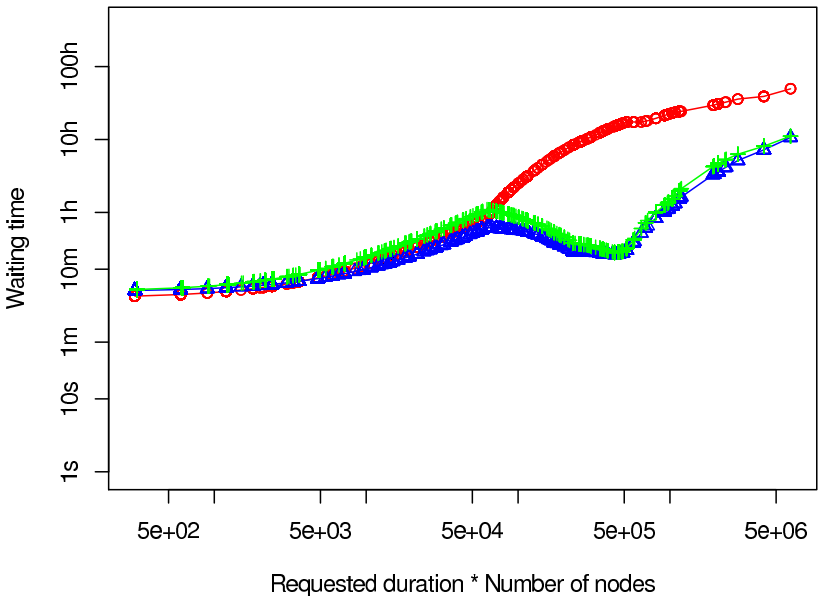


Figure 5.7: Effect of number of nodes on waiting time and slowdown in [10%/4H/medium]



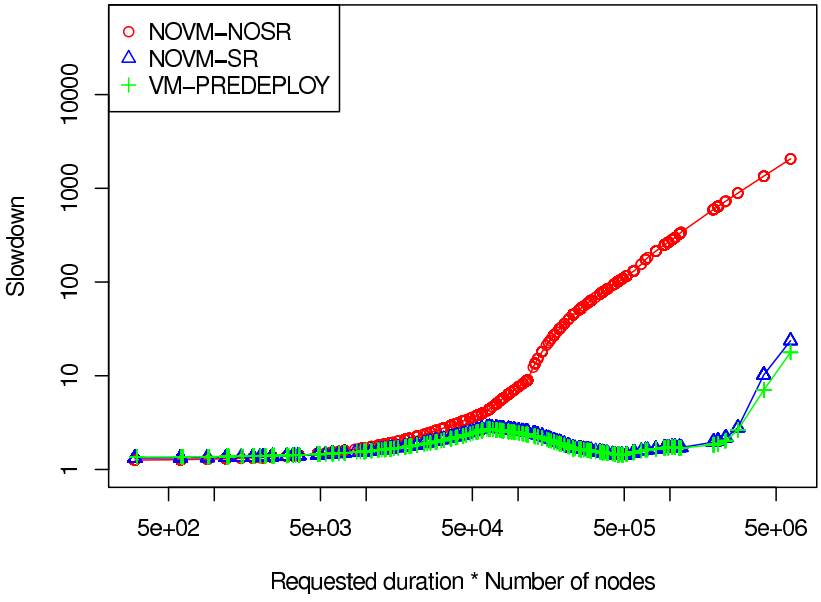


Figure 5.8: Effect of requested CPU hours on waiting time and slowdown in [10%/4H/medium]

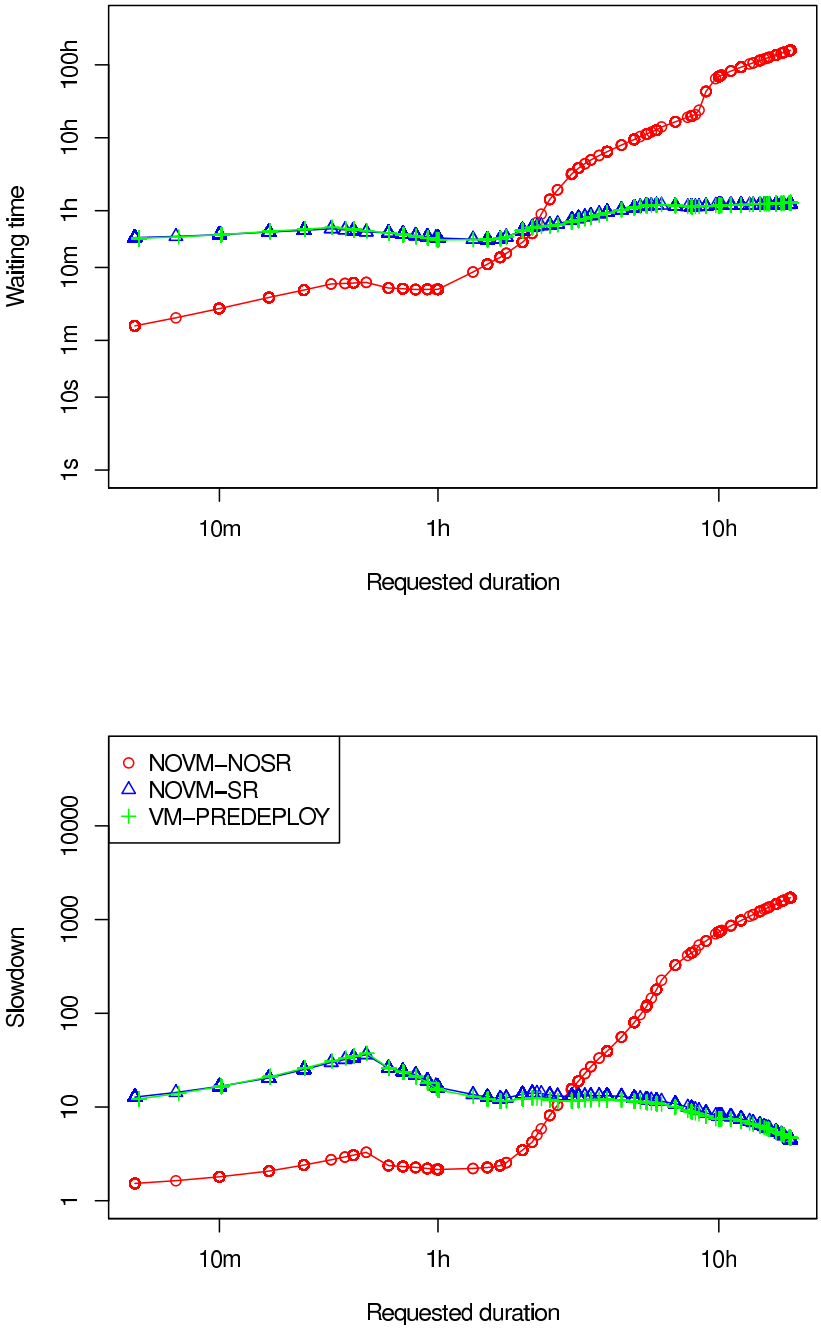


Figure 5.9: Effect of requested duration on waiting time and slowdown in [30%/2H/medium]

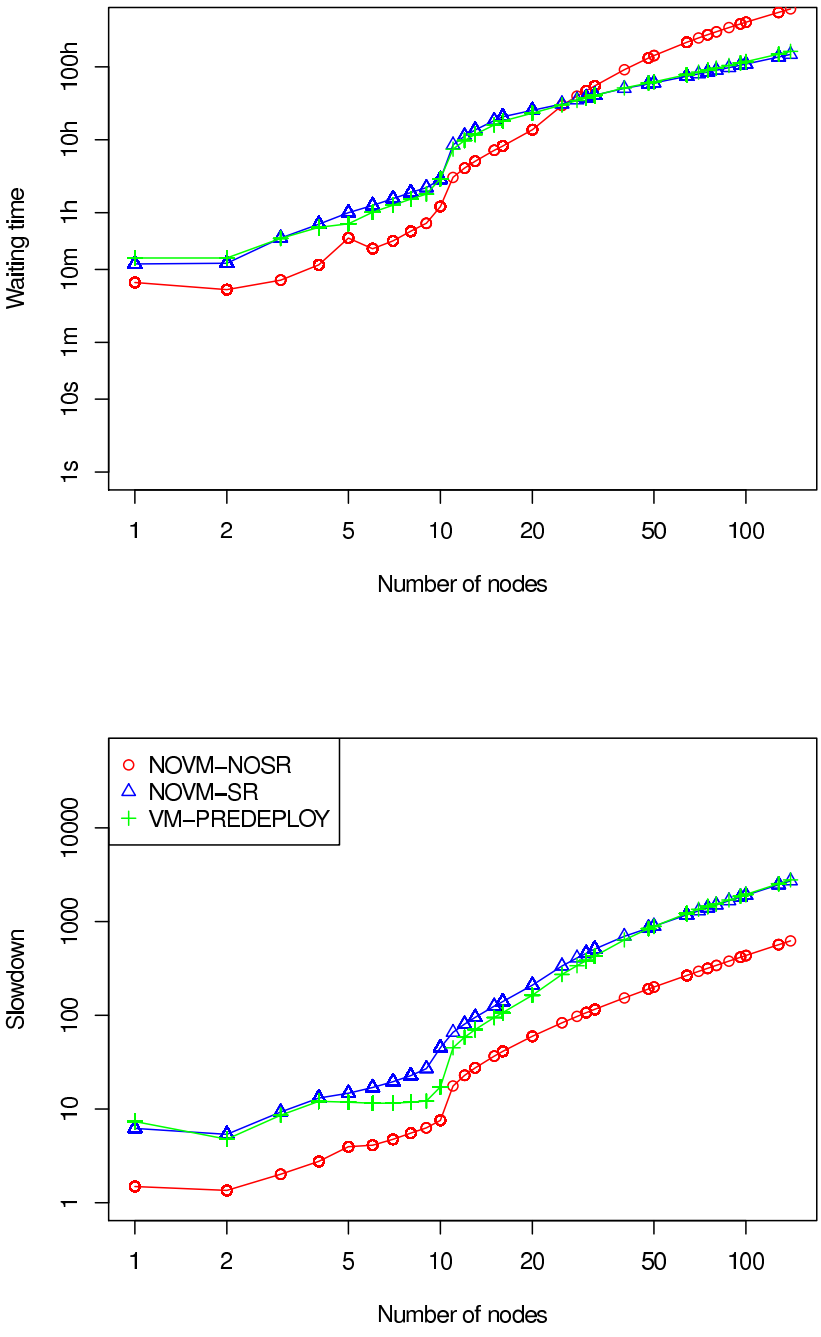
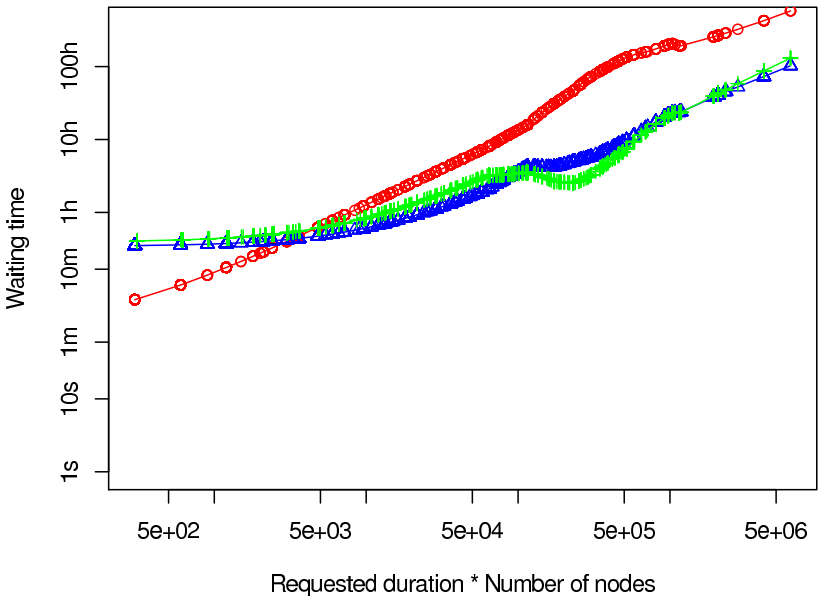


Figure 5.10: Effect of number of nodes on waiting time and slowdown in [30%/2H/medium]



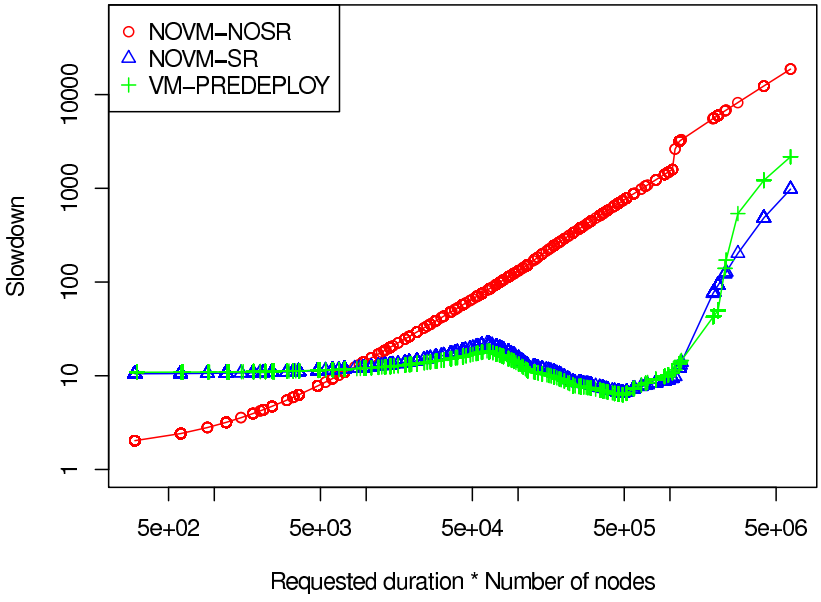


Figure 5.11: Effect of requested CPU hours on waiting time and slowdown in [30%/2H/medium]

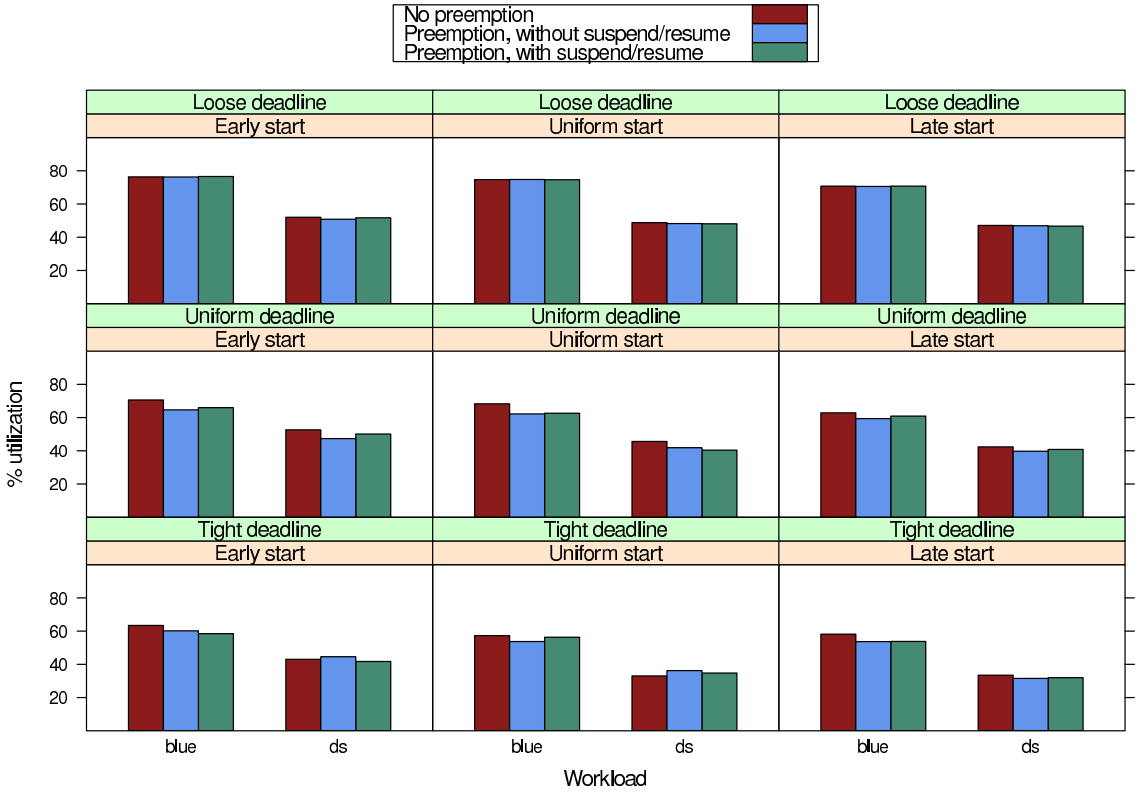


Figure 5.12: Utilization of resources in the BLUE2 and DS workloads, with and without preemption.

CHAPTER 6

SCHEDULING DISK IMAGE TRANSFERS

In the previous chapter, I presented how leases could be implemented with virtual machines, albeit assuming that any disk images required by the virtual machines were predeployed on all the physical nodes. Although this assumption may be reasonable for some use cases, such as compute clusters that have a small set of software environments that satisfy most users, it is not a general enough solution for most users. More specifically, if we want leases to encompass software environments, it must be possible for resource consumers to specify the exact software environment they want (in the form of a disk image), which will require transferring that disk image from a disk image repository to the physical nodes where the virtual machine will be executed.

An easy solution to this problem would be to let the overhead of deploying these disk images be borne by the resource consumer: the transfer of disk images would always take place at the start of a lease. However, doing so would be a breach of the lease terms, since the resources the consumer requested would not really be available until those transfers completed. Instead, I propose that this overhead be scheduled explicitly and separately from the VMs. In other words, the transfer of disk images would be scheduled in such a way that the terms of the lease are still met (e.g., the disk image transfers for an AR lease would be scheduled to complete before the start of the lease). Additionally, I claim that, by scheduling this overhead separately, we can take steps to reduce it.

In this chapter, I begin by presenting a VM disk image transfer strategy that guarantees that the lease terms (particularly the starting time) will not be breached (Section 6.1), followed by an algorithm for reusing disk images on physical nodes to reduce the number of transfers (Section 6.2), and a strategy for avoiding redundant transfers (Section 6.3. Finally, Section 6.4 extends the results from the previous chapter by showing how they are affected by the inclusion of disk image transfers.

6.1 Disk image transfer strategy

In Haizea, the transfer of disk images is scheduled separately from the VMs by the *preparation scheduler* (see Section 4.2.2). Given a lease, the preparation scheduler will tentatively schedule the transfers to determine the earliest time at which the disk image required by the lease can be transferred to each physical node. This time will be taken into account by the VM scheduler. For best-effort leases, the VM scheduler will not need to consider starting times before the images can be transferred. For other leases, the VM scheduler can reject a lease if the time to transfer the images does not allow the lease terms to be met (e.g., an AR lease can be rejected if the required disk images cannot be transferred before $start[l]$). Once the VMs have been scheduled, the disk image transfers are scheduled.

I assume that a site has a machine that acts as an image repository containing the disk images that can be deployed to the physical nodes, and that the repository and nodes are connected by a switched network with a bandwidth of B bytes/second. Furthermore, when a disk image must be transferred to several physical nodes, I assume that the transfer is done using multicasting. Thus, the time to deploy a lease's disk image is $\frac{size[software[l]]}{B}$ (irrespective of the number of physical nodes it is transferred to). Finally, I also assume that the image repository only multicasts one image at a time, so the schedule will be a sequence of disk image multicasts. The role of the preparation scheduler is, therefore, to determine the sequence in which these multicasts will happen.

If the disk image transfer schedule is empty, the transfer for best-effort and immediate leases is scheduled right away, and the transfer for AR and deadline leases is scheduled in such a way that the transfer will complete at exactly $start[l]$.

If the schedule already contains transfers, the scheduling of a new transfer depends on the type of lease:

- For best-effort leases, the scheduler tries to find the earliest gap in the schedule where

the transfer can fit (the worst case is at the end of the sequence)

- For immediate leases, the scheduler will only accept scheduling transfers at the present time. If that is not possible, the lease is rejected.

- For AR and deadline leases, the scheduler will attempt to schedule the transfer so it will finish at exactly $start[l]$. If that is not possible, all the transfers for AR and deadline leases are sorted by earliest deadline first, where the deadline for the image transfer is the start time of the lease. However, since the start time of an AR lease may occur long after the lease request, the transfers are pushed as close as possible to the start of the lease, preventing disk images from unnecessarily consuming disk space before the lease starts.

6.2 Reusing VM images

Scheduling disk image transfers separately can guarantee that lease terms are met, but the overhead of deploying many potentially large disk images (in the order of gigabytes) can delay or even prevent the start of leases. Reducing this overhead could allow the resource provider to reduce the waiting time for best–effort leases (by reducing the time spent waiting for image transfer to complete) and to accept AR and deadline leases with earlier start times.

To accomplish this goal I propose a disk image reuse mechanism, where each physical node has some disk space set aside for an *image pool* of frequently used disk images; when a lease is requested that requires one of these disk images, the preparation scheduler can use the disk image already deployed on the physical node, instead of scheduling a new transfer. The scheduler can use previously deployed images thanks to the reusability of most VM disk images; starting from a *master image* with an operating system and a desired software stack (e.g., a web server, a compute cluster node, etc.) multiple copies of the disk image can be created requiring only minor changes in each disk image (this process is commonly known as

contextualization). Additionally, even within the same physical node, this duplication can be done efficiently by accessing the master image using *copy–on–write* (COW), instead of creating an entire copy of the disk image. This can be done using LVM (Logical Volume Management), which most virtual machine hypervisors currently support.

Algorithm 6 outlines how the image pool in a node is used when a disk image is needed in that node. If a disk image required by a lease is not in that node's image pool, the disk image is first scheduled to be transferred to that node. Once the disk image is transferred, or if it was already in the image pool, the images are reference counted based on the leases that depend on them. An image's reference count is decremented each time a lease that depends on the image ends. If the image pool is full when an image is added, the scheduler removes the least recently used image with reference count equal to zero.

Algorithm 6 Image reuse

Input: A disk image $software[l]$ is required in node $n \in P$, for a VM v starting at t_{start} and ending at t_{end}. Each node has an image pool $imgpool[n]$. Each entry of $imgpool[n]$ is a disk image with an expiration time t_{expire} and a list of VMs vms that will be using that image.

if $imgpool[n]$ does *not* have a copy of $software[l]$ **then**
  Schedule transfer of $software[l]$ to n
  if disk image does not fit in image pool **then**
    while there is not enough space **do**
      Remove disk image with smallest t_{expire} and reference count equal to zero
    end while
  end if
  Add disk image to image pool, with $t_{\text{expire}} = t_{\text{end}}$ {This guarantees that the image will be available for the entire duration of the VM}
else
  if $t_{\text{start}} \leqslant t_{\text{expire}}$ of $software[l]$ in $imgpool[n]$ **then**
    {The image is guaranteed to be in the image pool at time t_{start}}
    Add v to $imgpool[img].vms$
    $t_{expire} \leftarrow \max(t_{\text{expire}}, t_{\text{end}})$
  else
    Transfer and add to pool as described above
  end if
end if

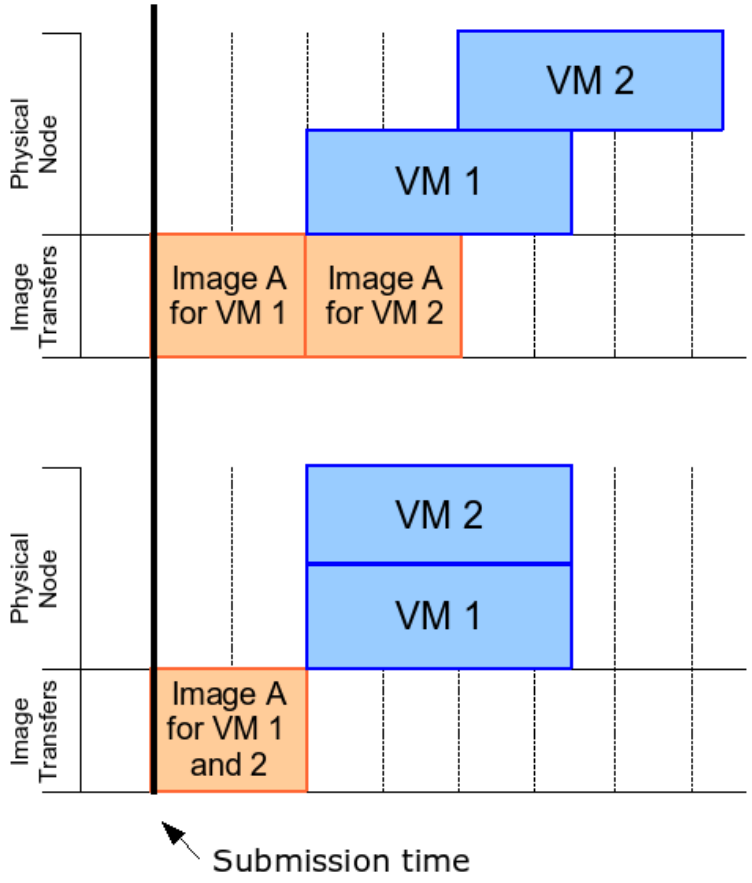


Figure 6.1: Avoiding redundant transfers

When scheduling leases, Haizea takes into account the state of the image pools in each node, and attempts to minimize the number of image transfers by mapping, whenever possible, VMs to nodes where the required image is already a part of the image pool.

6.3 Avoiding redundant transfers

As an additional optimization to the image reuse algorithm, image transfers are also scheduled in such a way that redundant transfers are avoided. Haizea avoids two types of redundant transfers:

Transfers for AR leases: Given a VM starting at time t_{start} and ending at time t_{end},

requiring image I, assigned to node n. If there is an image transfer of I scheduled to n, with deadline less than or equal to t_{start}, then we reuse that transfer.

For example, assume we have two leases, l_A and l_B. A transfer for image $software[l_A]$ has been scheduled to arrive on node n at time $start[l_A]$. At some point before $start[l_A]$, l_B is requested, requiring the same image as l_A, starting at $start[l_B]$ (where $start[l_B] > start[l_A] + duration[l_A]$). Haizea determines that l_B should be assigned to node n. Scheduling an additional transfer would be redundant, since there is already a transfer for that same disk image scheduled for that node. So, the existing transfer is tagged as carrying an image to be shared by l_A and l_B. Once the transfer is completed, the image's t_{expire} in the pool will be $start[l_B] + duration[l_B]$.

Transfers for best–effort leases: When scheduling the disk image transfer for a best-effort lease l, the preparation scheduler will check if any scheduled transfers will be multicasting image $software[l]$, so that the required transfer for l can simply "piggyback" on that existing transfer. For example, VWas shown in Figure 6.1 (top), assume that the last image transfer scheduled carries image A to node n, set to arrive at time t_{start} to be used by lease l_A and that the time to transfer image A is t_A. Now, a new best–effort lease l_B is requested, also requiring image A. If we scheduled a separate image transfer for l_B, it would start at $t_{\text{start}} + t_A$, despite the availability of resources at t_{start}. By allowing the transfer to piggyback on the previously scheduled transfer, l_B can start earlier, as show in Figure 6.1 (bottom)

In both cases, the scheduler will take into account existing transfers when mapping requests to nodes. Assuming availability of resources in the nodes, it is preferable to schedule a lease to a node with a reusable image transfer than scheduling it to a node where a new image transfer would necessarily have to be scheduled.

6.4 Experimental results

To evaluate the effect of adding disk image transfers and disk image reuse to lease scheduling, I extend the results presented in Section 5.4 by removing the assumption that all disk images are predeployed. The workloads, experiment setup, and metrics are the same as described in Section 5.4, except leases will be assigned a disk image, all with a size of 4GB, according to one of the following two distributions:

uniform: Each lease requests is assigned one of 40 disk images at random using a uniform distribution.

skewed: Each lease requests is assigned one of 40 disk images at random using a Pareto distribution, where eight disk images account for 80% of lease requests.

The workloads are simulated with the following three configurations:

VM–MULT — With VM, multiple images: Same as VM–PREDEPLOY, but using the *uniform* distribution of disk images and removing the assumption that images are predeployed (i.e., an image transfer has to be scheduled before the VM can start). Images are not reused on the nodes.

VM–REUSE–UNIFORM and VM–REUSE–SKEWED — With VM, multiple images with image reuse: Same as VM–MULT, but reusing images on the nodes. Using the *uniform* and *skewed* distribution of images, respectively.

As in the previous chapter, the above configurations assume a 1000Mbps network connection. Three additional configurations, VM–MULT–BW100, VM–REUSE–UNIFORM–BW100 and VM–REUSE–SKEWED–BW100 are defined as above, but assuming a 100Mbps network (these were only run with the [10%/3H/medium], [20%/2H/medium], and [30%/1H/medium] workloads).

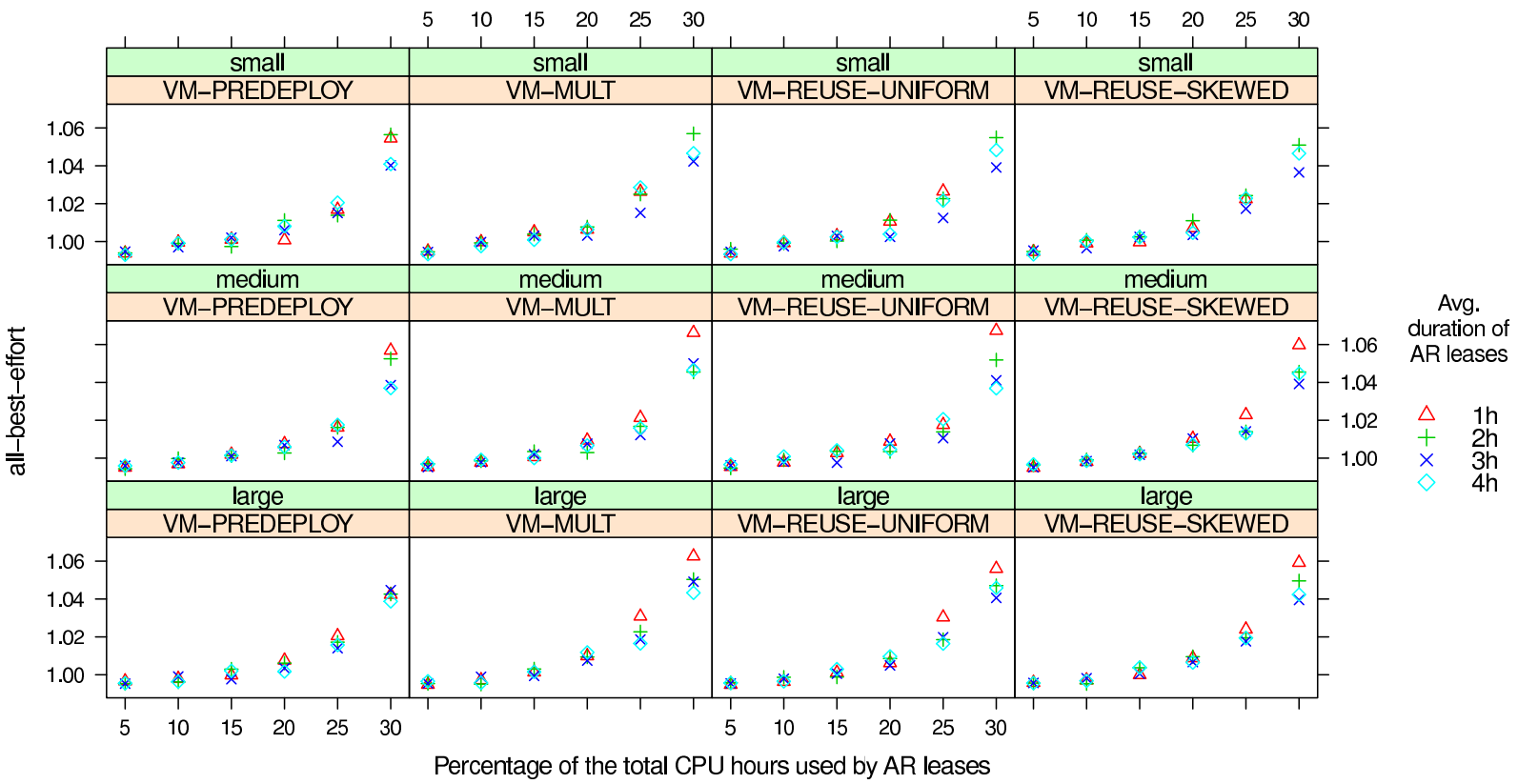


Figure 6.2: Each data point in this graph represents the all-best-effort metric in each experiment (the VM–PREDEPLOY configuration is included for comparison purposes). The x axis represents ρ and the y axis represents the value of all-best-effort (normalized as described in the text). The graphs are grouped by ν (the number of nodes requested by each lease) and the experiment configuration. The symbol of each point denotes the value of δ (see legend).

Figure 6.2 shows the values of the all-best-effort metric with the 1000Mps configurations. Similarly to Figure 5.2, when VMs are allowed to suspend and resume, the change in configuration has a negligible effect on this metric. Once again, looking at the waiting times and slowdown of the individual leases provides more information on the impact of disk image transfers. Table 6.1 shows the average waiting times and slowdowns for [10%/3H/medium], [20%/2H/medium], and [30%/1H/medium], and Figures 6.3 through 6.11 show how waiting time and slowdown vary according to requested duration and number of nodes.

Table 6.1: Experiment running times, average waiting times, and average slowdowns for [10%/4H/medium], [20%/3H/medium], and [30%/2H/medium]

Average waiting time for best-effort leases, in thousands of seconds

$\rho \rightarrow$	10%	20%	30%
VM–PREDEPLOY	6.18	10.88	27.43
VM–MULT	6.29	10.69	27.48
VM–REUSE–UNIFORM	5.76	9.81	24.63
VM–REUSE–SKEWED	5.67	10.30	22.96
VM–MULT–BW100	12.10	23.44	49.16
VM–REUSE–UNIFORM–BW100	7.05	12.35	30.74
VM–REUSE–SKEWED–BW100	5.13	9.98	24.96

Average bounded slowdown

$\rho \rightarrow$	10%	20%	30%
VM–PREDEPLOY	24.66	47.27	121.33
VM–MULT	30.36	49.06	144.39
VM–REUSE–UNIFORM	26.67	44.46	134.65
VM–REUSE–SKEWED	26.64	48.90	115.44
VM–MULT–BW100	74.95	132.78	292.96
VM–REUSE–UNIFORM–BW100	36.01	62.05	157.42
VM–REUSE–SKEWED–BW100	23.07	43.83	118.64

Interestingly, although adding image transfers does have an impact on all metrics, it is relatively small in the 1000Mbps configurations. Accordingly, the benefit of reusing images turns out to be small. However, the impact of transferring images is larger if we assume a

lower network bandwidth; the average waiting times and slowdowns are at least double those when not transferring images. In this case, adding image reuse (VM–REUSE–UNIFORM–BW100 and VM–REUSE–SKEWED–BW100) reduces wait times and slowdowns, although performance is still not as good as when using predeployed images (VM–PREDEPLOY).

6.5 Conclusions

This chapter has removed the assumption that VM disk images are predeployed on physical nodes, and has presented algorithms that allow a lease's requested disk image to be transferred from a disk image repository in such a way that the lease's terms are not broken. Additionally, I have presented strategies for optimizing these disk image transfers by reusing frequently used images and reducing the number of redundant transfers. The results show that, when transferring 4GB images through a 1000Mbps network, the impact of adding image transfers, regardless of whether disk images are reused, turns out to be relatively small. However, when those same images are transferred through a 100Mbps network (with each transfer taking five minutes, instead of thirty seconds; of course, a five minute transfer time could also happen when transferring 40GB images through a 1000Mbps network), the impact on performance is considerable, and disk image reuse proves to be an effective technique to palliate, although not entirely eliminate, the overhead of transferring disk images.

Thus, by modelling and scheduling the preparation overhead involved in using virtual machines, the work presented in this chapter is a step towards meeting G4-ModelVirt. However, I have explored only the case where disk images are transferred from a disk image repository. Although this is a common use case, more work can be done to explore other deployment strategies, such as the use of P2P mechanisms. Additionally, the results highlight how the impact of image transfers does not depend solely on the network bandwidth or on the image size, but on the additional preparation overhead per lease. Thus, it may be interesting to explore reuse algorithms that, instead of using a simple least-recently-used policy, have

the specific goal of reducing the amount of preparation overhead (e.g., by targeting images that take five minutes to transfer instead of thirty seconds).

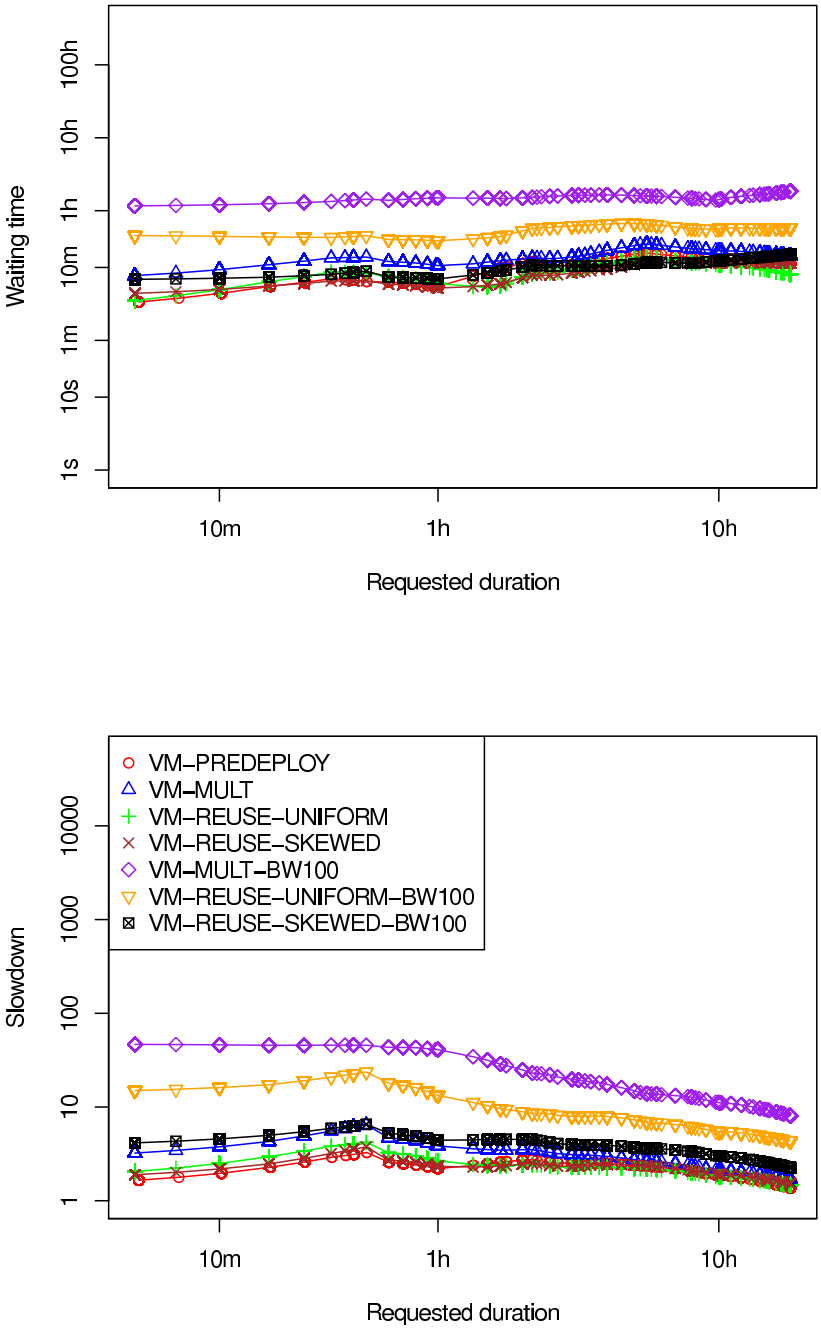


Figure 6.3: Effect of requested duration on waiting time and slowdown in [10%/4H/medium]

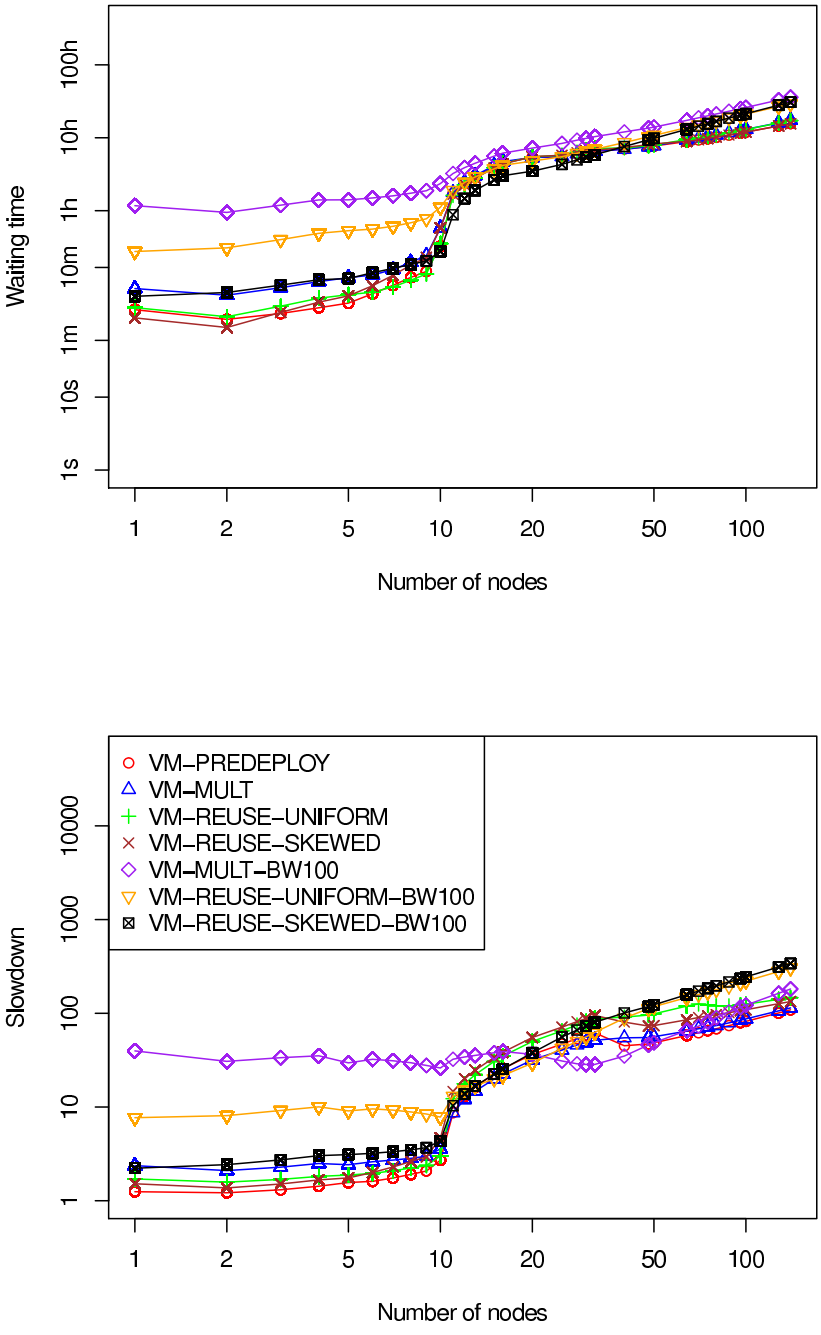
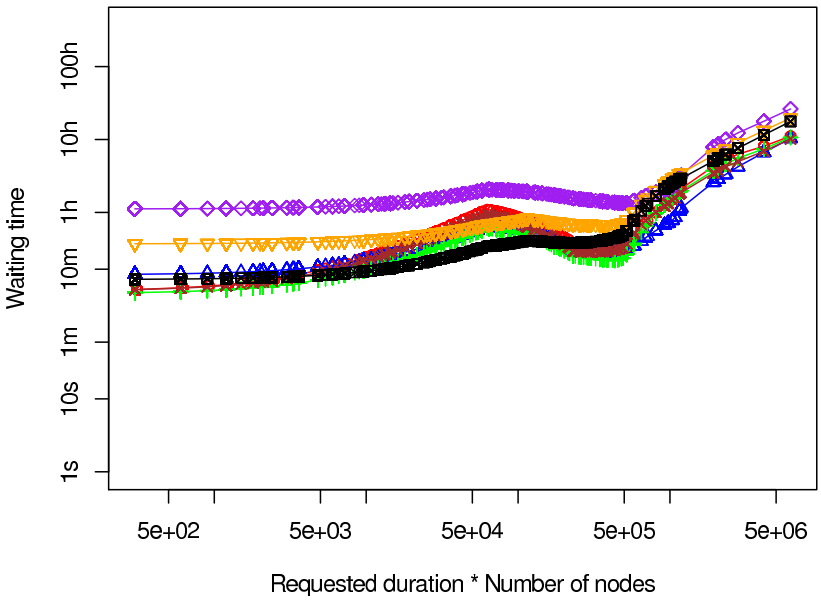


Figure 6.4: Effect of number of nodes on waiting time and slowdown in [10%/4H/medium]



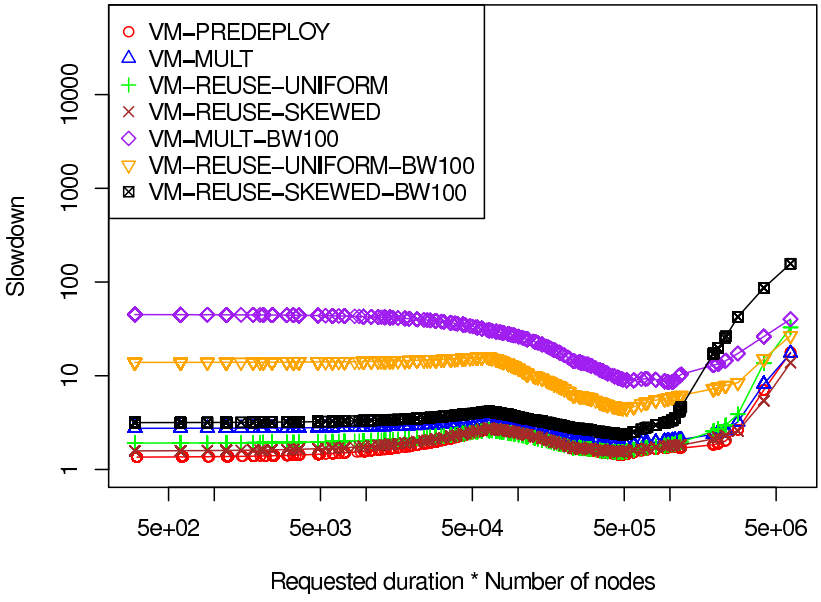


Figure 6.5: Effect of requested CPU hours on waiting time and slowdown in [10%/4H/medium]

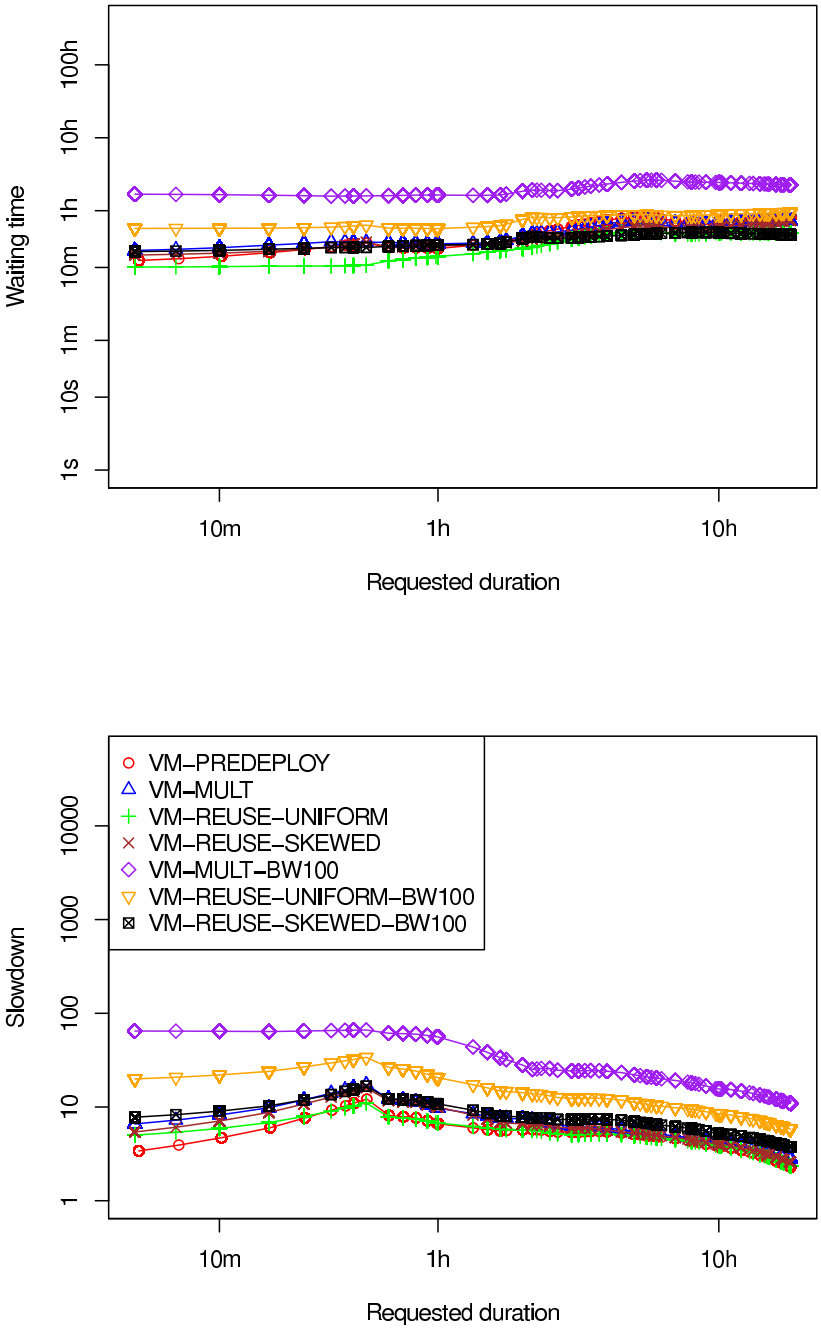


Figure 6.6: Effect of requested duration on waiting time and slowdown in [20%/3H/medium]

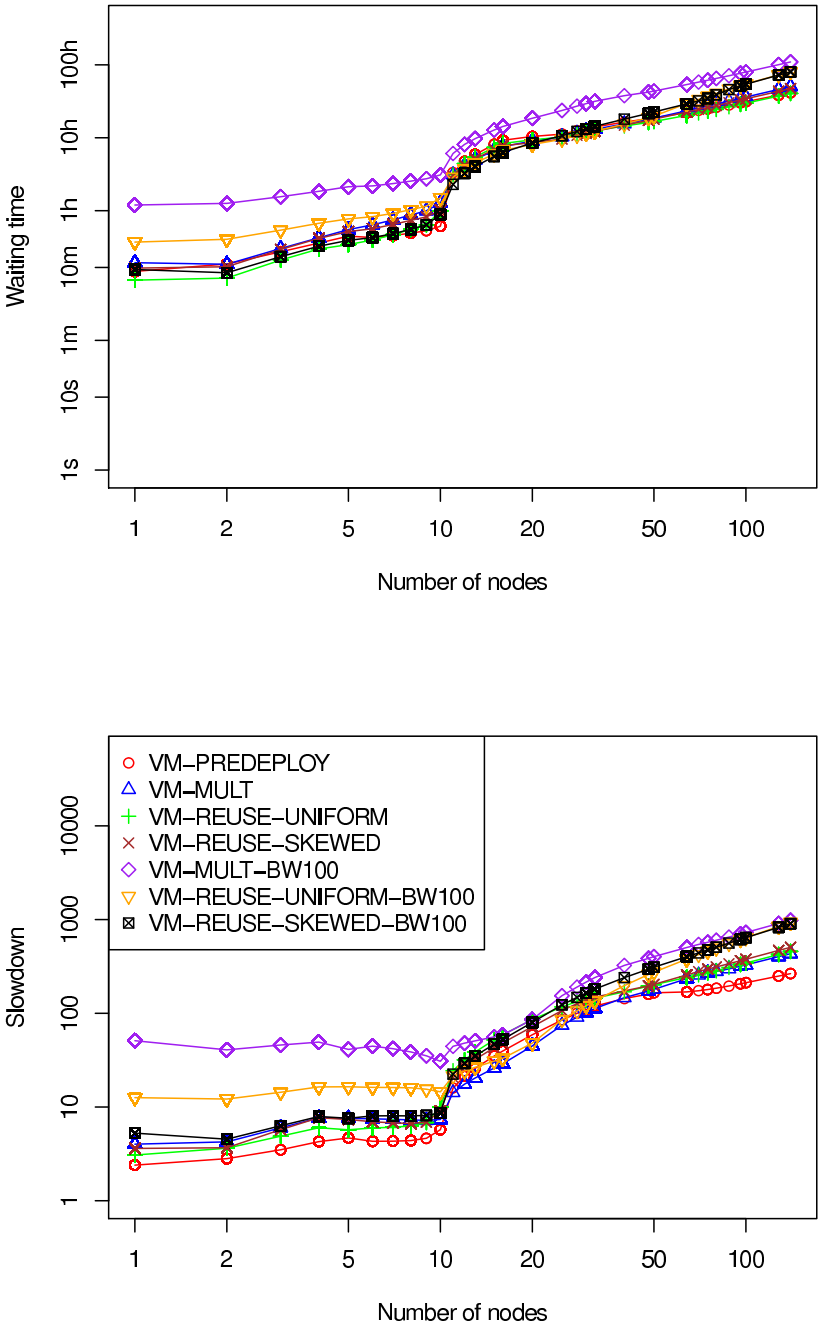
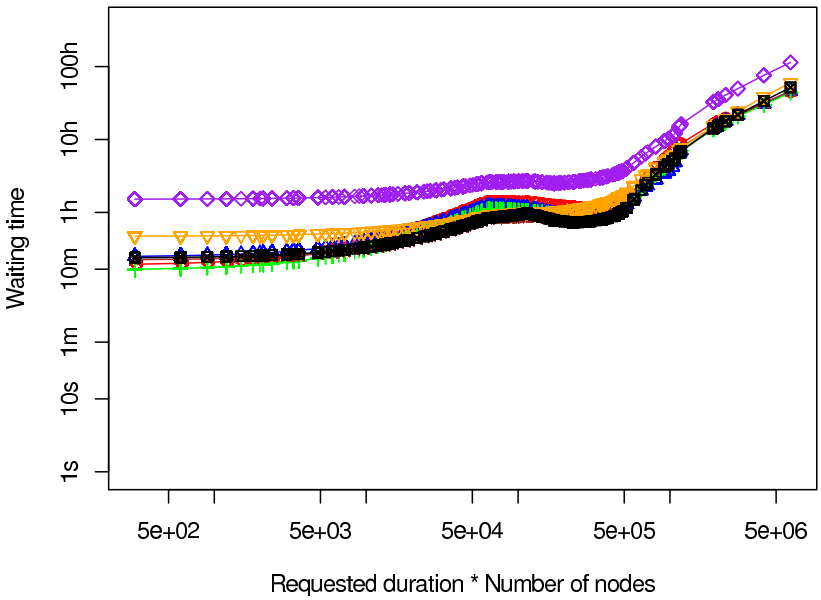


Figure 6.7: Effect of number of nodes on waiting time and slowdown in [20%/3H/medium]



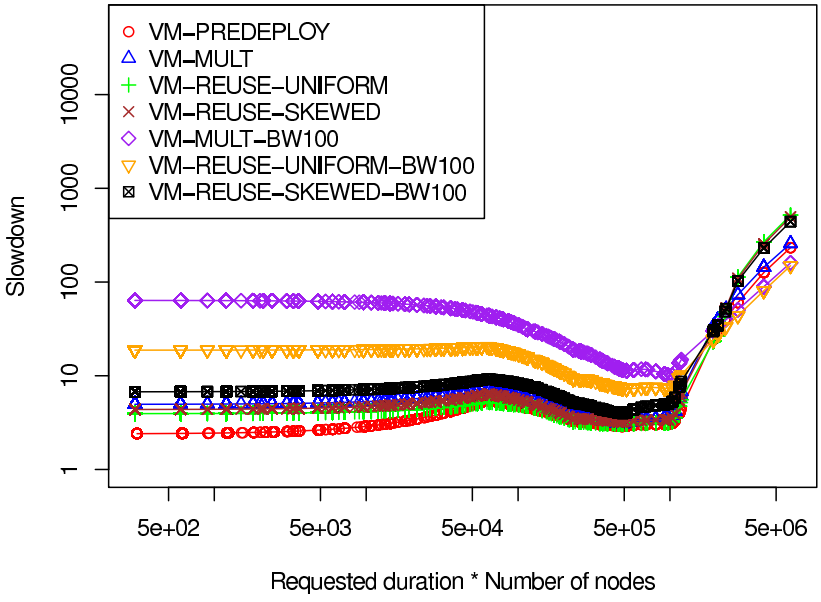


Figure 6.8: Effect of requested CPU hours on waiting time and slowdown in [20%/3H/medium]

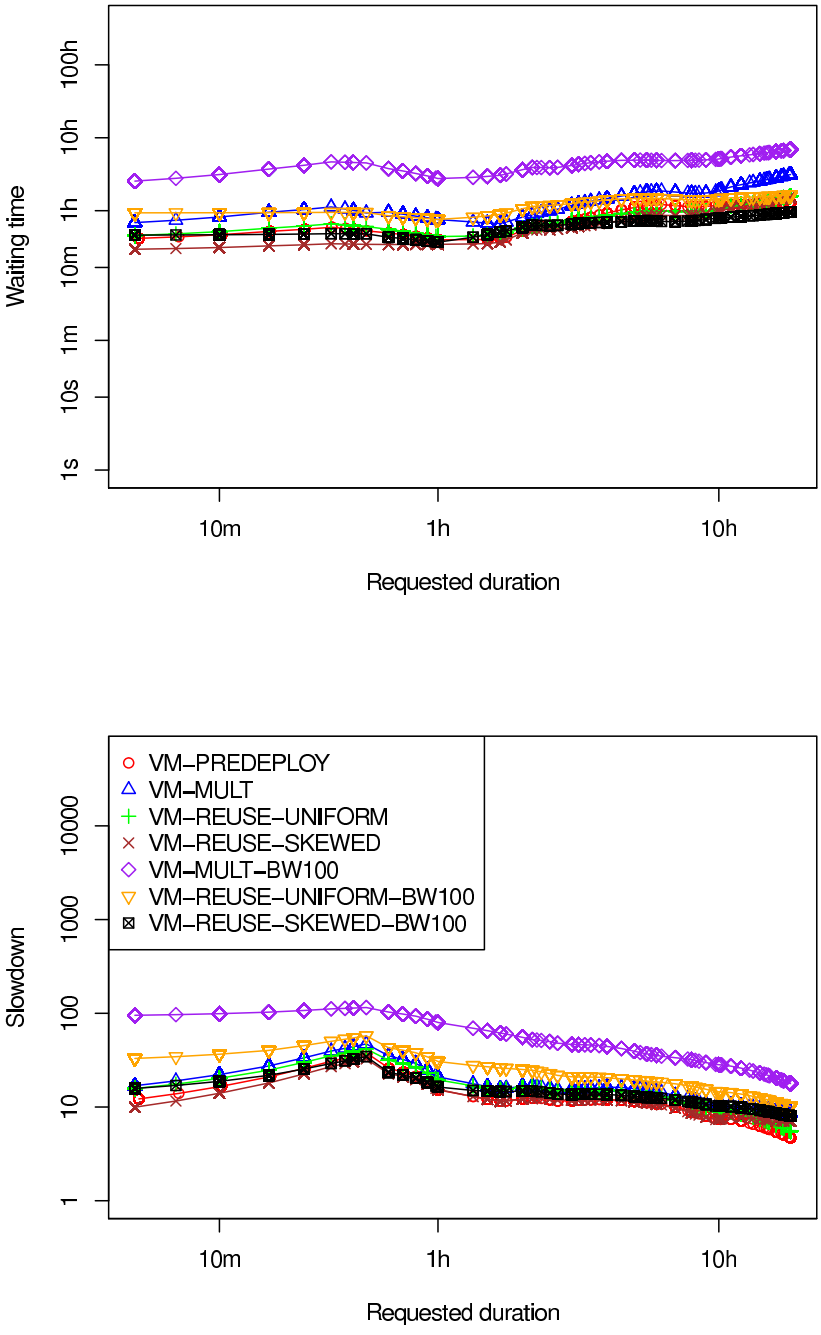


Figure 6.9: Effect of requested duration on waiting time and slowdown in [30%/2H/medium]

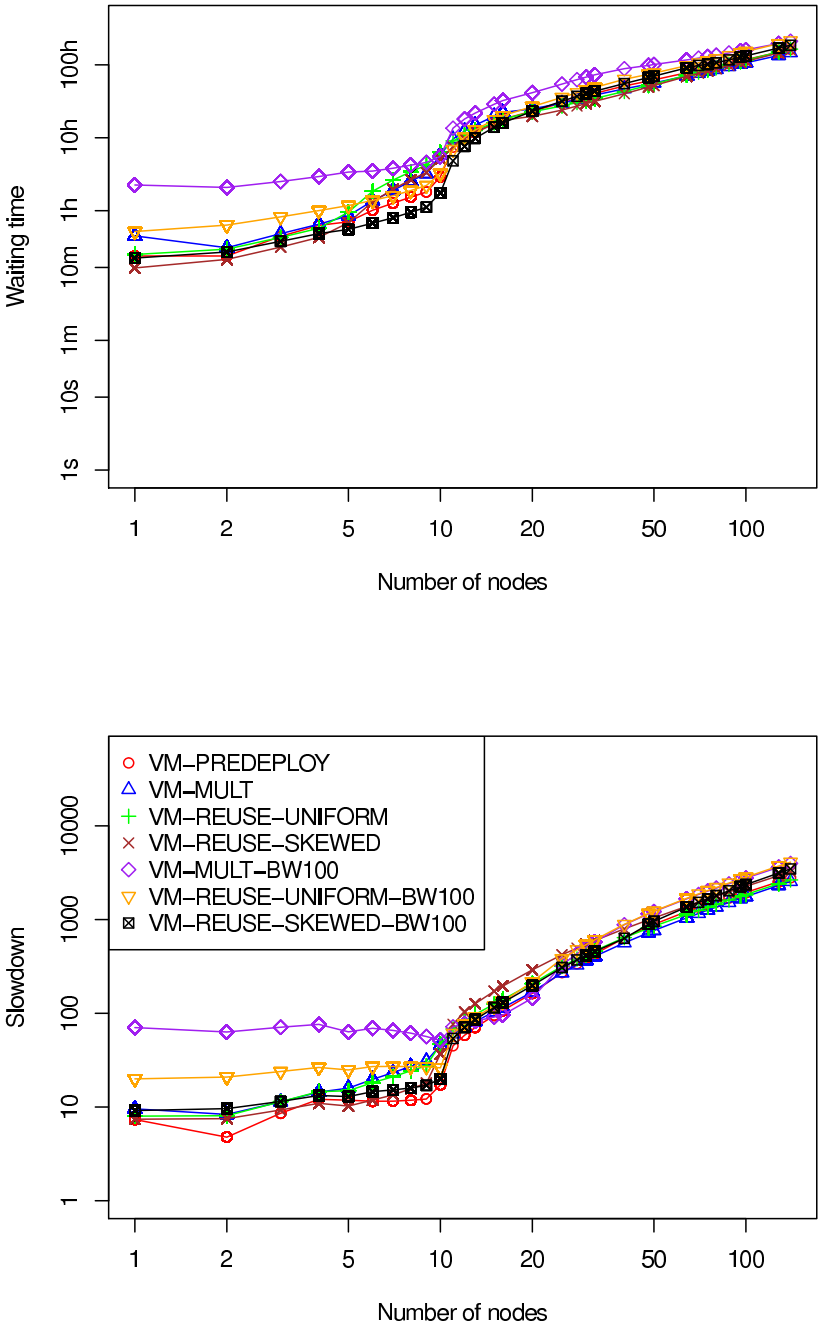
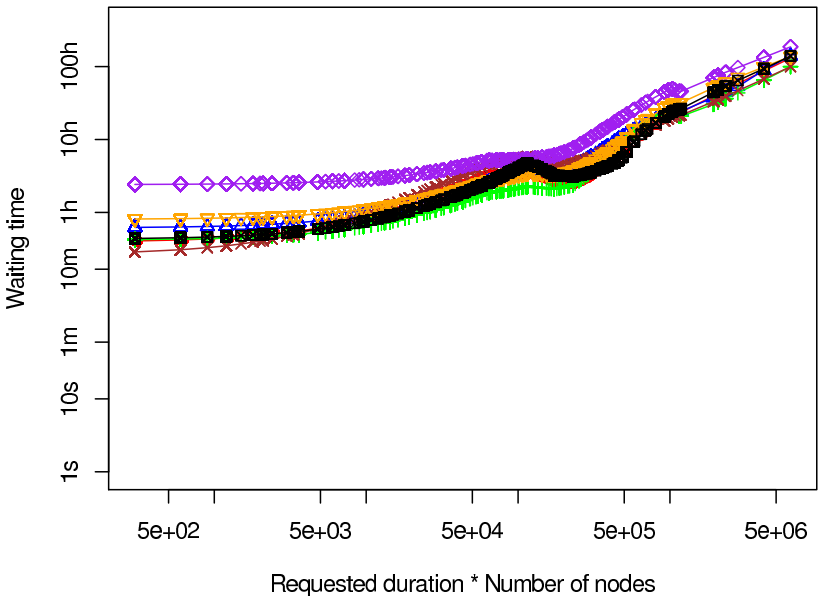


Figure 6.10: Effect of number of nodes on waiting time and slowdown in [30%/2H/medium]



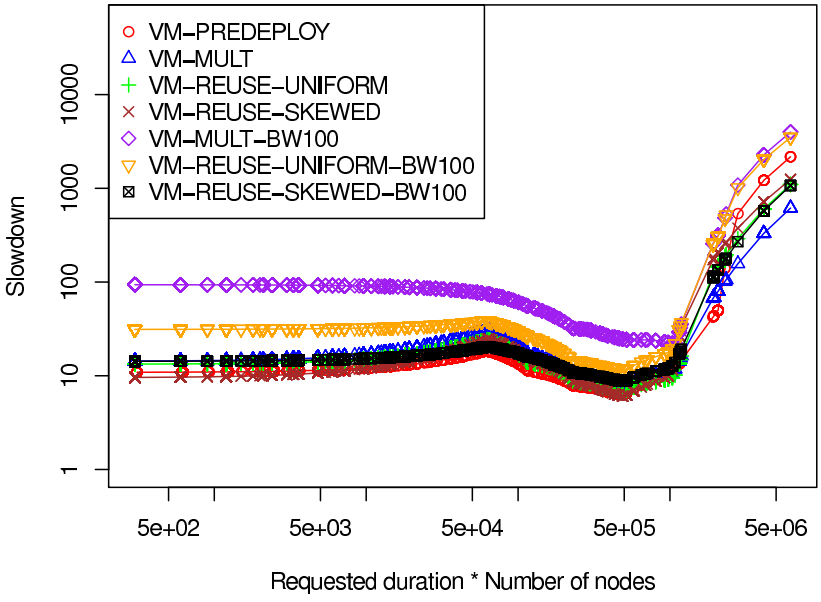


Figure 6.11: Effect of requested CPU hours on waiting time and slowdown in [30%/2H/medium]

CHAPTER 7

MODELLING AND SCHEDULING VIRTUAL MACHINE SUSPENSION AND RESUMPTION TIMES

The previous chapter focused on modelling *preparation overhead*, or the actions that must take place before the start of a lease, such as transferring VM disk images. However, at this point we still have a simple model for the *runtime overhead* of suspending and resuming VMs, based on the assumptions that only allowing a single VM can be deployed per physical node and requiring that memory state images (resulting from suspensions) be saved to the local filesystem of the physical node, without allowing for the option of using a global filesystem. This assumption made VM scheduling simpler, but did not capture the interactions that may arise when multiple VMs are scheduled on a same physical server.

In this chapter, I explore a more comprehensive model of VM suspension/resumption that removes the assumptions made in Chapter 5. I present this model in Section 7.1, followed by experimental results in Section 7.2 testing the degree of accuracy of this model on physical hardware and augmenting the simulation results from Chapter 5 by exploring the long-term effects of modifying certain variables in the model.

7.1 Modelling VM suspension/resumption times

In chapter 5, I assumed that the time to suspend and resume a VM was $t_s = m/h_s$ and $t_r = m/h_r$, respectively (where m is the amount, in megabytes, of memory the VM has and h_s and h_r are the rates, in megabytes of VM memory per second, to suspend/resume the VM). If we remove the assumption that each physical node can only run one VM at a time, a lease may have more than one VM in a physical node, and these formulas become invalid since we can no longer assume that all the VMs can be suspended in parallel. Although each physical node can suspend VMs independently of what is happening on other nodes, the

time to suspend will now depend on the number of VMs scheduled on each physical node. Additionally, in preliminary experiments I observed that suspending or resuming multiple VMs simultaneously on the same physical node results in longer and more unpredictable suspension times due to contention for I/O. Thus, suspensions and resumptions must happen sequentially, so each has exclusive access to the filesystem. More specifically, if n_i is the number of VMs mapped to physical node i, then suspending all the VMs in i will require $n_i \cdot \frac{m}{h_s}$. Furthermore, if we remove the assumption that the memory state file resulting from suspension is saved to a local filesystem, and allow them to be saved to a global filesystem, the VMs in each physical node can no longer be suspended independently of others, and the scheduler needs to account for contention when accessing the shared filesystem.

Thus, the time to suspend an entire lease becomes the following:

$$t_s = \begin{cases} \max(n_0, n_1, \ldots, n_P) \cdot \frac{m}{h_s} & \text{if } f = \mathsf{local} \\ N \cdot \frac{m}{h_s} & \text{if } f = \mathsf{global} \end{cases} \tag{7.1}$$

t_r is defined similarly.

Next, each suspend/resume command sent to physical nodes will incur a communication overhead. In preliminary experiments I observed that this overhead cannot be assumed away, since OpenNebula sends commands to physical nodes sequentially over TCP and, thus, even if a command only takes 1-2 seconds to be processed, suspending 64 VMs would result in a total overhead of 64-128 seconds. Thus, if e is the enactment overhead of sending a suspend/resume command, the formula becomes:

$$t_s = N \cdot e + \begin{cases} \max(n_0, n_1, \ldots, n_P) \cdot \frac{m}{h_s} & \text{if } f = \mathsf{local} \\ N \cdot \frac{m}{h_s} & \text{if } f = \mathsf{global} \end{cases} \tag{7.2}$$

Finally, this model also takes into account sh, the time to shutdown a VM. When suspending/resuming a lease A to free resources for another lease B, the end of B is, in general,

followed by the resumption of A. Previously, this shutdown time did not figure into the calculation of the suspend/resume time, since resumption was assumed to begin as soon as the ending lease started shutting down. However, I observed in preliminary experiments that allowing B's shutdown to overlap with the resumption of A would noticeably delay the first resumption operations, resulting in a longer t_r than expected. Although modelling h does not affect the formulas for t_s and t_r, it is taken into account by the scheduler.

7.2 Experimental results

To evaluate the accuracy and effects of the model described in this chapter, I carried out two series of experiments on physical hardware and in simulation. The first set of experiments, carried out on physical cluster running the Xen hypervisor[1], focuses on determining what factors can affect the accuracy of the model, and showing the effect of underestimating and overestimating the values of h_s and h_r in practice. The other set of experiments, carried out by simulating 30 days of lease requests, shows the long-term effects of different parameter values in our model.

7.2.1 Experiment setup

The testbed used for the first set of experiments is made up by five SunFire x4150 servers, each with two Intel Xeon QuadCore L5335 2GHz processors (i.e., 8 cores per server, $p = 8$) and 8GB of RAM ($m = 8192$). All the nodes are connected with a switched Gigabit Ethernet network. One node is used as a head node that hosts a shared NFS filesystem for all the nodes, while the remaining four nodes are used to run virtual machines (i.e., 32 cores available to run VMs). The head node also runs OpenNebula 1.0 and Haizea, which manage all the VMs during the experiments. The testbed is configured to operate both with Xen 3.2 or

1. http://www.xen.org/

KVM, although the current results are based on Xen 3.2. All virtual machines used in the experiments have 2GB disk images on the shared NFS filesystem[2]. When suspending or resuming a virtual machine, OpenNebula can be instructed to save the file to the NFS filesystem or to the local filesystem of the node where the virtual machine is running.

For the first set of experiments, Haizea needs values for h_s and h_r. I determined these values by suspending and resuming a single virtual machine (with $m = 1024$) 25 times, and measuring the times v_s and v_r to suspend and resume that single VM (these times were extracted by parsing the Xen logs). When $f = \mathsf{local}$, $\overline{v_s} = 15.4$ ($\sigma = 0.58$) and $\overline{v_r} = 14.12$ ($\sigma = 0.67$). When $f = \mathsf{global}$, $\overline{v_s} = 14.00$ ($\sigma = 1.04$) and $\overline{v_r} = 11.60$ ($\sigma = 0.50$). I conservatively estimate h_s to be $\frac{m}{\overline{v_s}+2\cdot\sigma_{v_s}}$, and estimate h_r similarly. Thus, for $f = \mathsf{local}$, h_s is 61.86 MB/s and h_r is 66.27 MB/s and, for $f = \mathsf{global}$, h_s is 63.67 MB/s and h_r is 81.27 MB/s.

7.2.2 Experiment #1: Suspending and resuming leases

In this experiment, two leases are scheduled on the cluster: a best-effort lease (Be_Lease) initially scheduled to use all available resources, which is preempted to free up resources for an AR lease (Ar_Lease), which involves suspending all the VMs in Be_Lease, and then resuming them once Ar_Lease ends. The purpose of this experiment is to measure how accurately lease suspension and resumption times are estimated, comparing the value predicted by the model to the actual times measured on our testbed. Although these results only highlight how these leases are scheduled on this particular testbed, they also provide insights into what factors can have an impact on the accuracy of the model.

Both leases are requested at the start of the experiment and require all the physical resources on the testbed. Ar_Lease has a duration of 5 minutes, and must start 15 minutes

2. Since one of the experiments involves running 32 VMs, each of which needs a separate disk image, 2GB was the largest image size that we could accommodate in the shared NFS filesystem.

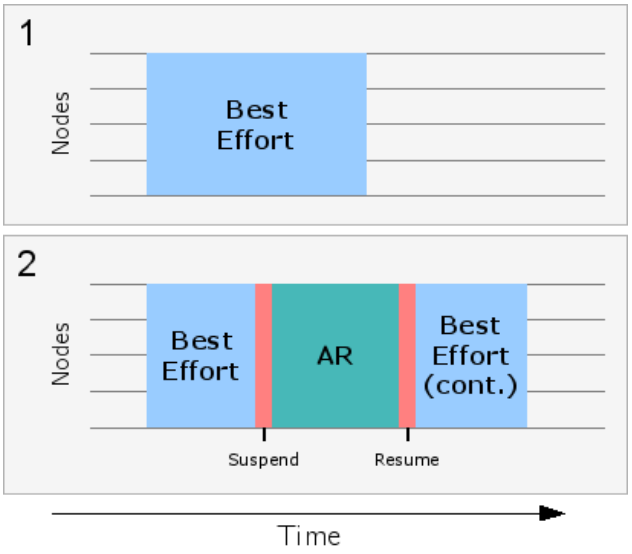


Figure 7.1: (1) A best-effort lease is scheduled to use all available nodes. (2) An AR lease is requested at the time when the best-effort lease is using the testbed resources, necessitating preempting the best-effort lease. The best-effort lease is suspended before the AR and then resumed once the AR ends.

into the experiment. Be_Lease has a duration of 20 minutes, and since there are no other leases at the start of the experiment, Be_Lease can start immediately (but will be preempted by Ar_Lease when it starts 15 minutes into the experiment; see Figure 7.1). In this experiment I explore the following three parameters:

- $c \in \{1, 2, 4, 8\}$: The number of VMs per physical node. Since each lease uses all available resources, then N, the total number of VMs, will be $c \cdot 4$.
- $f \in \{\mathsf{local}, \mathsf{global}\}$: Whether the memory state files are saved to a local or global filesystem.
- $m \in \{512, 768, 1024\}$: Amount of memory per VM.

This experiment is performed in 22 configurations (one for each combination of parameters, except those with $c = 8$ and $m = 1024$; since the Xen Dom0 domain uses 512 MB of

memory, VMs that consume a total of 8192MB cannot be started), and each configuration is run five times. In each run, I measure the following metrics:

- $\tilde{t_s}$: The *observed* time used to suspend Be_Lease. I obtain this time by pinging each VM in the lease every 2 seconds and recording whether it responds or not. I analyse the sequence of responses to determine how much time the lease took to suspend.

- a_s: The *accuracy* of suspension, or how close the observed time to suspend was to the predicted time. Thus, I define a_s as $\frac{\tilde{t_s}}{t_s}$. A value of 1.0 indicates perfect accuracy. Values less than 1.0 indicate the time was *overestimated*, meaning that the VMs in the lease finished suspending earlier than the time predicted by the model; overestimation has the effect of leaving resources idle between the end of the suspension and the start of the AR. Values larger than 1.0 indicate the time was *underestimated*, meaning the lease took longer to suspend than estimated; underestimation has the effect of delaying the start of the AR.

- $\tilde{v_s}$: The *observed* time used to suspend a single VM in Be_Lease. I parse the Xen log files to determine this time.

- $\tilde{t_r}, a_r, \tilde{v_r}$: Defined similarly for resumption.

Figure 7.2 shows the average values for $\tilde{t_s}$, $\tilde{t_r}$, a_s, and a_r in each experiment configuration. The averages are taken from the five runs of each configuration. The standard deviation is not shown in the graphs but is at most 13.9% of the average (with most values below 10%). These graphs show that, as expected, the time to suspend and resume increases with the amount of memory requested, with the rate of increase being more pronounced when f = global. The observed times show that the model tends to overestimate the time to suspend, with six configurations having an accuracy larger that 1.0 (overestimation) and a maximum value of 1.39 (i.e., the lease took 39% longer to suspend than estimated). On the

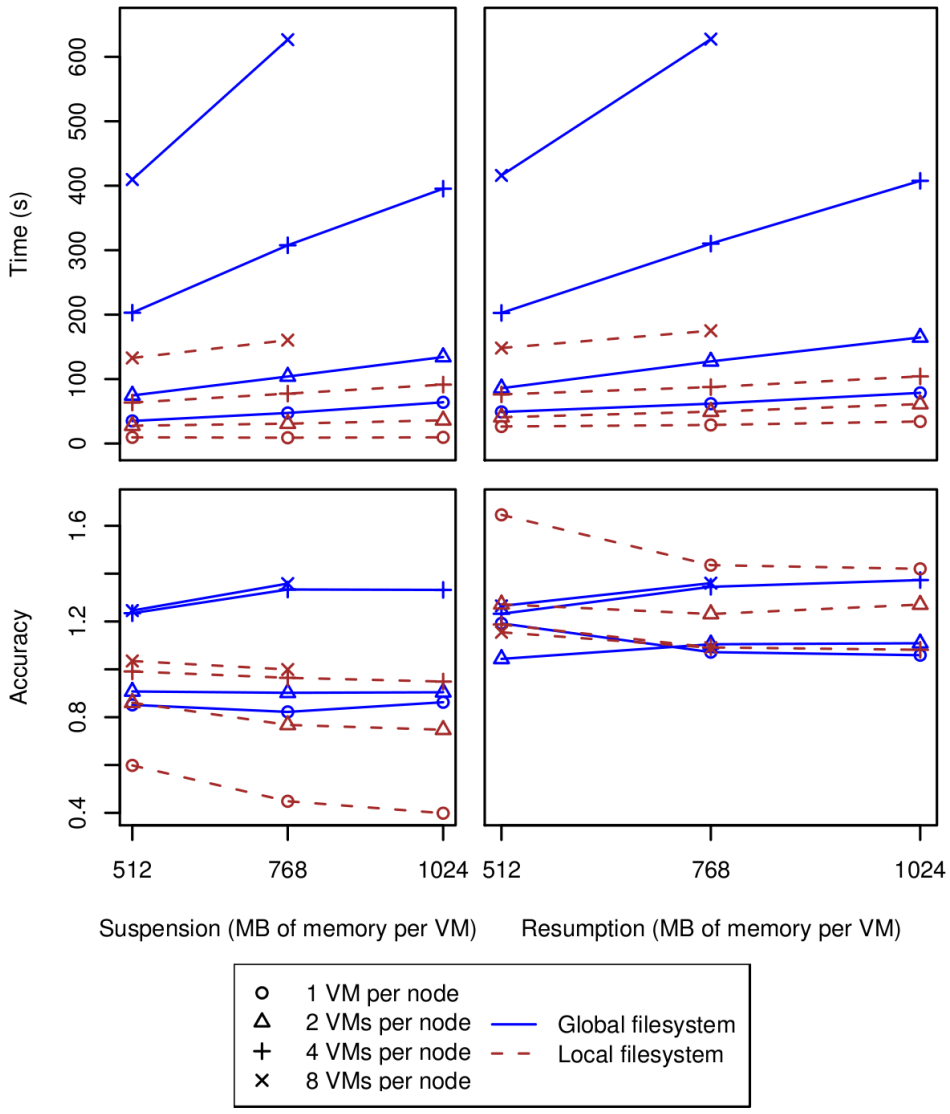
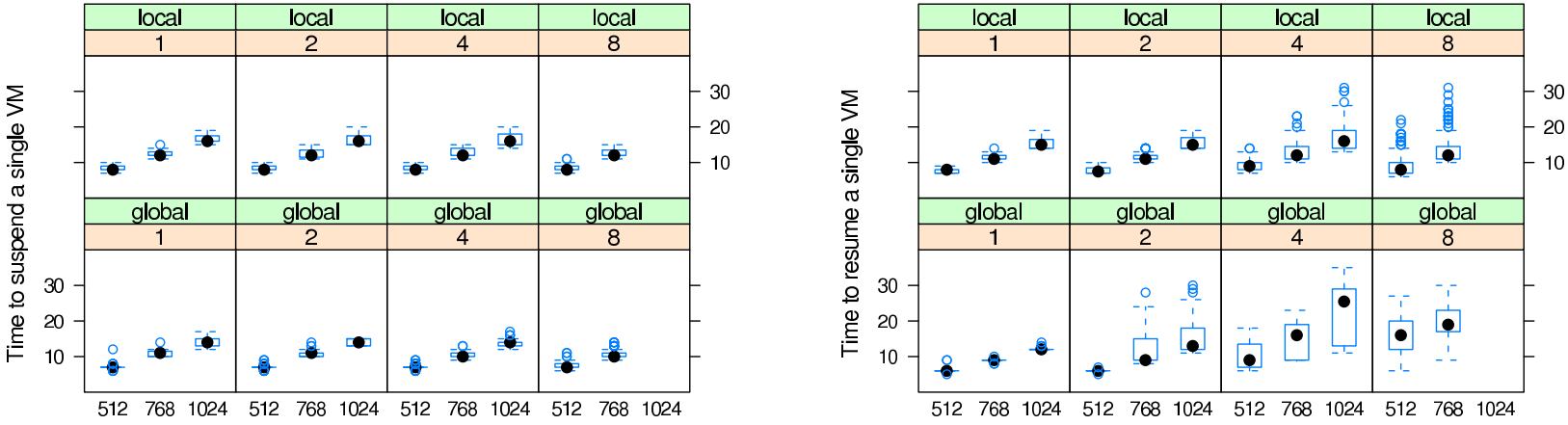


Figure 7.2: Experiment #1: Observed time to suspend and resume an entire lease (top) and accuracy of estimation (bottom)

other hand, resumption times are all underestimated, with all accuracies above 1.0 and a maximum value of 1.64.



(a) Suspension times

(b) Resumption times

Figure 7.3: Experiment #1: Distribution of times required to suspend and resume individual VMs, for all combinations of cores per physical nodes (1, 2, 4, 8), memory used by each VM (1024, 2048, 3072, 4096), and filesystem used (local or global).

The explanation for these values can be found by looking at how individual VMs behave when suspending and resuming. Figures 7.3a and 7.3b show the distribution of values for $\tilde{v}_s$ and $\tilde{v}_r$, respectively. Values of $\tilde{v}_s$ show little dispersion, although the graph does not show 19 outliers (out of 1388 measurements) with times ranging between 45 seconds and 370 seconds. By inspecting the Xen logs, I found that these outliers are caused by suspensions that, for no apparent reason, Xen blocked on. In other words, Xen correctly receives and processes the suspension command, including pausing the VM, but does not actually save the memory state to disk until an arbitrary amount of time has passed (although it does not block all other operations; e.g., other suspensions are processed correctly). I do not know why this blocking occurs and have not found any mention of it in other work.

On the other hand, the values of $\tilde{v}_r$ tend to become more dispersed as c increases. The direct cause for this dispersion is resource contention between overlapping resumptions. Although the scheduler plans the resumptions in such a way that they will not overlap (either globally or locally, depending on f), sometimes a resumption might take slightly longer than estimated, affecting the time of the next planned resumption, delaying both in the process. I found that the underlying cause for these delays is the following:

1. *The shutdown of* Ar_Lease *can overlap with the first resumptions.* If Ar_Lease is still in the process of shutting down when the resumptions start, there will be contention for resources, delaying the first resumption. In this experiment, h is a fixed value (15) regardless of the number of nodes in a lease, and this value turned out to be insufficient when $c = 8$ (even assuming that a single VM can be shutdown in 1 second, with a 1 second enactment overhead, that still adds up to 64 seconds).

2. *Delays in enactment commands.* Occasionally, enactment commands sent from OpenNebula (which uses SSH to send commands) would be delayed by the SSH server itself, sometimes up to 10 seconds.

3. *The larger c is, the greater the likelihood of a cascade effect.* On top of the previous two, for larger values of c, a delayed resumption is more likely to affect other resumptions, resulting in more dispersed values overall. This effect is especially apparent when the shutdown overlaps with the first resumption, which could delay up to 31 other resumptions thereafter (when $c = 8$ and $f = \mathsf{global}$).

The above factors can have an impact on the time to suspend and resume leases, and can potentially delay other leases, such as AR leases that depend on other leases being preempted before they can start.

7.2.3 Experiment #2: Long-term effects

The previous experiment explored small self-contained cases on real hardware to validate the accuracy of the model in predicting and scheduling the overhead of preempting a lease. However, it reveals nothing about the long-term implications of using resource leases and how changes in the model's parameters can compound over time, particularly those that affect suspension/resumption times. In this final set of experiments, I run Haizea in simulation mode to process 30 days of lease requests.

The workloads I have used are a subset of those used in the previous two chapters. More specifically, I use the BLUE1 workload with the following injected AR leases: [10%/3H/-medium], [20%/2H/medium], [30%/1H/medium] (hereafter T10, T20, and T30).

These three workloads were chosen because they test two extremes of ARs. Given that BLUE1 has a utilization of 69.60%,, T10 introduces a relatively small number of ARs, with more time between each reservation, whereas T30 introduces a large number of ARs. T20 is meant as an intermediate point between the two. I limit the discussion to these three workloads because the effect of varying ρ, δ, and ν was already explored in the previous chapter, and the focus of this experiment is on the effect of other parameters.

As in the previous two chapters, the simulated cluster in the experiment is modelled after

the SDSC Blue Horizon cluster. However, whereas I previously assumed a single processor per physical node ($p = 1$) and that memory state files are saved to a local filesystem ($f = \mathsf{local}$), here p will equal 1, 2, 4, or 8 (and vary the number of nodes accordingly so the total number of processors in the cluster is 144) and will also allow memory state files to be saved to a global filesystem. Furthermore, I assume the values for h_s and h_r are the same as the ones in the Xen testbed (indicated in Section 7.2.1). Finally, I assume that VM disk images do not need to be deployed before the start of a lease (e.g., because they are on a global filesystem or are predeployed to the local filesystems, similar to configuration VM–PREDEPLOY in Chapter 5).

In sum, the parameters in this experiment are the following:

- $\mathsf{workload} \in \{\mathsf{No\ ARs}, \mathsf{T10}, \mathsf{T20}, \mathsf{T30}\}$
- $C \in \{1, 2, 4, 8\}$ (number of cores per physical node)
- $f \in \{\mathsf{local}, \mathsf{global}\}$
- $m \in \{1024, 2048, 3072, 4096\}$

In this experiment, I explore all 128 combinations of the above four parameters. In each run, I record the all-best-effort, waiting time, and slowdown metrics, as defined in Section 5.4.3. However, in this experiment, the all-best-effort metric is normalized differently. Instead of normalizing with the time to process the workload without any ARs, here it is normalized, for each workload, relative to the time required to process the workload under the assumption that suspension and resumption can be done instantaneously (e.g., a value of 1.1 indicates that processing the entire workload took 10% longer than if we could suspend and resume virtual machines instantaneously).

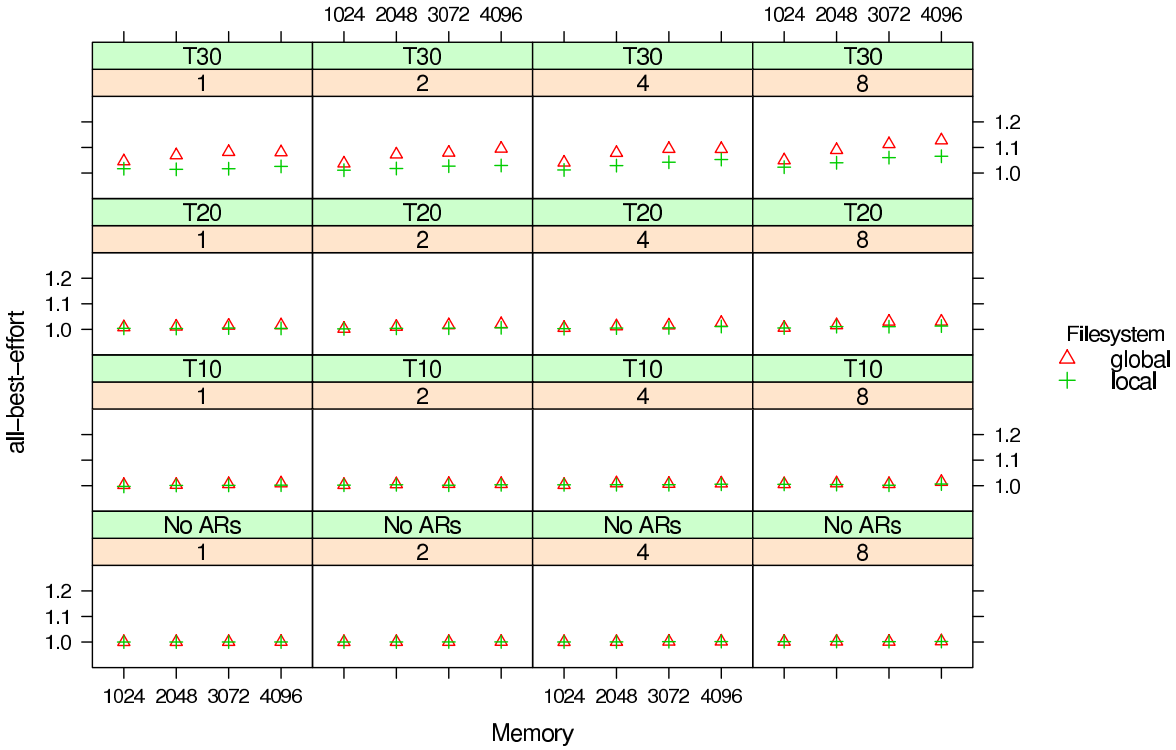


Figure 7.4: Experiment #2: Values of the all-best-effort metric for all the combinations of workloads (No ARs,T10,T20,T30), cores per physical nodes (1, 2, 4, 8), memory used by each VM (1024, 2048, 3072, 4096), and filesystem used (local or global).

Figure 7.4 shows the values of all-best-effort for each combination of the experiment parameters. Interestingly enough, when using workloads T10 and T20, the effect of all parameters on the running time of the best-effort leases is relatively small, with all-best-effort being at most 1.01 and 1.05 respectively. However, with workload T30, where preemptions are more likely to happen, the effect is more noticeable, particularly when using a global filesystem, with values of all-best-effort up to 1.14. In this case, the amount of memory per VM has a more noticeable effect on performance, especially when suspensions and resumptions have to be globally exclusive. This effect is even more apparent when in the average waiting times of the leases (Figure 7.4).

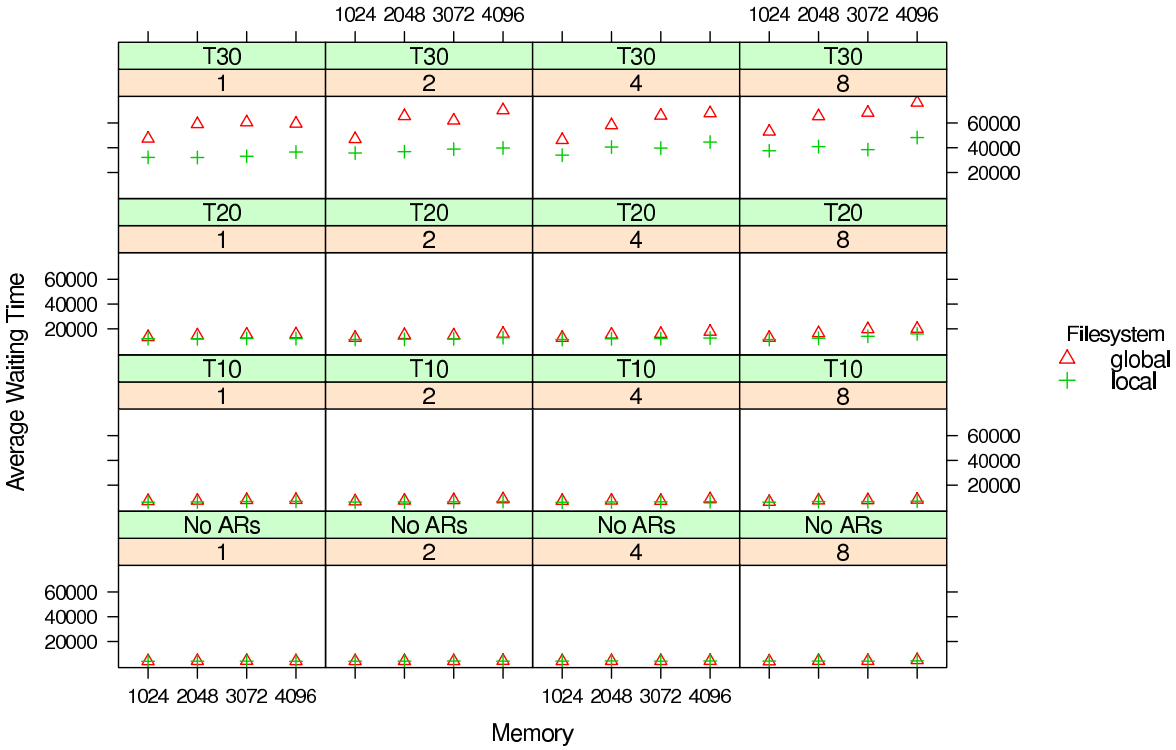


Figure 7.5: Experiment #2: Values of the average waiting time for all the combinations of workloads (No ARs,T10,T20,T30), cores per physical nodes (1, 2, 4, 8), memory used by each VM (1024, 2048, 3072, 4096), and filesystem used (local or global).

Table 7.1: Effect of network bandwidth on all-best-effort

Workload	1000Mbps	100Mbps	Difference
No ARs	1.00	1.00	+0.39%
T10	1.00	1.00	+0.61%
T20	1.00	1.04	+3.81%
T30	1.04	1.14	+8.98%

Another parameter that may have an effect on performance is the network bandwidth, which directly affects h_s and h_r when f = global. Fixing $C = 1$ and $m = 1024$, I reran the simulations assuming a 100Mbps Ethernet network. Since the physical testbed does not have a 100Mbps network with which to determine the values of h_s and h_r, I assume that they will be one tenth of what they are in the 1000Mbps network (i.e., $h_s = 6.367$ MB/s and h_r is 8.127 MB/s). Table 7.1 summarizes the results of these simulations. The effect on all-best-effort is still small with workload T10, but more noticeable when using workloads T20 and T30.

7.3 Conclusions

This chapter has described a model for estimating the time required to suspend and resume a lease implemented with virtual machines, removing the assumptions made in Chapter 5. The experimental results show that, while this model only underestimates suspension times in a few cases (which would delay the start of an AR lease), it also tends to overestimate suspension times, resulting in idle times between the end of the suspension and the start of the AR, and also shows that the model tends to underestimate resumption times. However, these inaccurate estimations where due to specific factors –overlapping of shutdowns and resumptions, delays in enactment commands, and cascade effects–, highlighting how an incomplete model can have a significant impact not just on lease resumption times but also on the rest of the schedule. On the other hand, the simulation experiments show that varying

some of the parameters in the model has relatively small long-term effects on performance, as measured by the all-best-effort metric and the average waiting times, except when using workloads with a large number of ARs, which results in a larger number of lease preemptions.

Thus, although the model and results shown here get us closer to meeting goal G4-ModelVirt, more work is needed to refine this model and minimize the impact of the factors mentioned above. In particular, at the time of this writing, work is already underway to make Haizea more adaptive to unexpected events, such as suspensions/resumptions that take longer than expected (e.g., because of a delay in an enactment command), to avoid contention for resources when these operations take place. Additionally, this chapter has explored only one model, along with fixed values for h_s and h_r; more work is needed to explore different estimation models and variable values of h_s and h_r, and showing their effect on the probability that ARs will be delayed (instead of presenting only the degree of accuracy of the estimations).

CHAPTER 8

PRICING STRATEGIES FOR LEASES

Up to this point, I have assumed an *accept all* policy, whereby a provider never declines a lease request if there are enough resources to satisfy that request. Such a policy provides no incentives for users *not* to use AR leases rather than best-effort leases, and also provides a resource provider with no mechanism other than lease refusal to signal that a site is overloaded. In this chapter, I explore price-based policies for lease admission, hypothesizing that prices can be used both to provide an incentive for the use of best-effort leases (if ARs cost more than best effort) and to signal (via higher prices) when a resource is overloaded.

First, Section 8.1 describes how leases are priced and the interaction between resource consumers and providers in the presence of prices. Next, Section 8.2 presents several pricing strategies, and Section 8.3 presents an experimental evaluation of these pricing strategies.

8.1 User model

As described in Chapter 3, leases can have a price $price[l]$. I assume that resource providers set this price using a *rate* r (measured in $\frac{\text{Monetary unit}}{\text{node·second}}$):

$$price[l] = duration[l] \cdot nodes[l] \cdot r$$

Although a provider can change r, once a price for a lease is set, future changes to r will not affect that price.

Assume there is a set of users U that request leases. A user $u \in U$ requesting a lease l ($user[l] = u$) is willing to pay at a rate no higher than r_u, the user's *acceptable rate*. Thus, a user is willing to pay no more than $p_{u,l} = duration[l] \cdot nodes[l] \cdot r_u$ for the lease.

The interaction between the user and the resource provider is the following:

1. The user u sends a request for a lease l to the resource provider.

2. Based on the current lease schedule...

   (a) ...if the lease can be scheduled, the provider determines the price $price[l]$ it will charge for the lease (pricing strategies are discussed in the next section). This price is sent to the user.

   (b) ...if the lease is not feasible, the provider rejects the lease.

3. The user evaluates $price[l]$, and...

   (a) ...if $price[l] > p_{u,l}$, the user rejects the lease.

   (b) ...if $price[l] \leqslant p_{u,l}$, the user accepts the lease.

8.2 Pricing strategies

A resource provider implements a *pricing strategy* to meet its individual goal(s). For example, some providers may set rates with the goal of maximizing revenue, while others seek to maximize resource utilization. Here I explore four pricing strategies. The first three are used for comparison purposes:

- CONSTANT: The provider always uses the same rate r_i.

- RANDOM: For each requested lease, the provider uses a random rate, chosen from a uniform distribution (r_{min}, r_{max})

- MAXIMUM: The provider has access to a magical oracle that enables it to divine the exact value of r_u for every user. For each requested lease, the provider charges the maximum rate the requesting user is willing to pay. Thus, a lease is only rejected if the provider is unable to schedule the lease (and never because the consumer rejects the price).

The fourth strategy, ADAPTIVE, initializes r to an initial value r_i, but then adjusts the value of r based on user behaviour with the goal of maximizing revenue. This strategy entails not just maximizing the number of accepted leases, but also minimizing the amount of revenue lost to *undercharging* (the difference between what the provider charged and what the user was actually willing to pay). To do this, the ADAPTIVE strategy looks at the sequence of leases accepted and rejected by each user u to estimate the user's acceptable rate, and then uses these estimated values to approximate the median value of r_u. The procedure used to update r after a lease is accepted or rejected is shown in Algorithm 7.

8.3 Experimental Evaluation

To evaluate the model and algorithms presented in this chapter, I have performed simulation experiments with Haizea using the BLUE2 and DS workloads presented in Chapter 5. As described in that chapter, these workloads are taken from the Parallel Workloads Archive, and are annotated with a deadline and starting time. Additionally, each request is annotated with a value for r_u.

Besides the deadline and starting time (parameters δ and ω as defined in Section 5.5), I vary the following parameters in each experiment run:

- **The distribution of values of r_u.** Each user in the workload is assigned a separate value between \$0.10 and \$10.00 (i.e., between 10^{-1} and 10^{1}), according to four distributions of values:
  - CHEAP USERS: Pareto distribution skewed toward \$0.10.
  - CONSTANT USERS: Always \$1.00 (i.e., 10^{0})
  - UNIFORM USERS: Uniform distribution of values.
  - RICH USERS: Pareto distribution skewed toward \$10.00.

Algorithm 7 Update r after a lease l has been accepted or rejected by a user. Parameters UP_MULTIPLIER and DOWN_MULTIPLIER control the factors by which the estimated rate for a user has to be updated when the user accepts or rejects a lease. Parameter N controls how r is chosen from the estimated rates of the users (if N=0.5, the median value is used).

$u \leftarrow user[l]$
if we haven't encountered u yet **then**
  $accept_{high}[u] \leftarrow \varnothing$ {Highest rate at which user accepted a lease}
  $reject_{low}[u] \leftarrow \varnothing$ {Lowest rate at which user rejected a lease}
  $estimate[u] \leftarrow \varnothing$
  $done[u] \leftarrow$ **False**
end if
if l was rejected by user **then**
  if $reject_{low}[u] = \varnothing$ **then**
    $reject_{low}[u] \leftarrow r$
  else
    $reject_{low}[u] \leftarrow min(r, reject_{low}[u])$
  end if
else if l was accepted by user **then**
  if $reject_{low}[u] \neq \varnothing$ **and** $accept_{high}[u] \neq \varnothing$ **then**
    $done[u] \leftarrow$ **True**
  else if $accept_{high}[u] = \varnothing$ **then**
    $accept_{high}[u] \leftarrow r$
  else
    $accept_{high}[u] \leftarrow max(r, accept_{high}[u])$
  end if
end if
for all u encountered so far such that $done[u] =$ **False do**
  if $accept_{high}[u] = \varnothing$ **then** {User has rejected every rate so far}
    $estimate[u] \leftarrow reject_{low}[u] \cdot$ DOWN_MULTIPLIER
  else if $reject_{low}[u] = \varnothing$ **then** {User has accepted every rate so far}
    $estimate[u] \leftarrow accept_{high}[u] \cdot$ UP_MULTIPLIER
  else {Estimate the user's rate as the midpoint between the highest recorded rate at which the user accepted a lease and the lowest recorded rate at which the user rejected a lease}
    $estimate[u] \leftarrow \frac{reject_{low}[u]+accept_{high}[u]}{2}$
  end if
end for
$r \leftarrow$ N-th percentile of all estimates

- **Pricing strategy**: CONSTANT, RANDOM, MAXIMUM, or ADAPTIVE (as described in Section 8.2)

Random numbers are generated using the Python random number generator, with known seeds so as to make the results reproducible. I simulated the two workloads under every combination of these parameters, and fixing the DOWN_MULTIPLIER, UP_MULTIPLIER, and N parameters in Algorithm 7 to 0.5, 1.5, and 0.5, respectively (i.e., prices are adapted downwards by half, upwards by half, and we choose the median of the estimated rates). In all cases, Haizea is allowed to preempt VMs with suspend/resume.

8.3.1 Metrics

In each experiment run I measure the following information:

- *Revenue*, the sum of the price of all the leases sold during the experiment. In each run, this value is normalized relative to the *total possible revenue*, or revenue the provider would get if it were somehow able to satisfy every single demand in the workload *and* charge the maximum price that the user is willing to pay.

- *Missed revenue (underpricing).* The missed revenue for a lease is the difference between the maximum price for a lease (assuming we had charged each user up to his acceptable rate) and the actual price. The total missed revenue is the sum of the missed revenue of all leases.

- *Missed revenue (price rejected by user)*: The sum of the price of all the leases that were rejected by the user, priced at the user's acceptable rate (not at the rate at which they were offered to the user).

- *Missed revenue (lease could not be scheduled)*: The sum of the price of all the leases, priced at the user's acceptable rate, rejected by the provider.

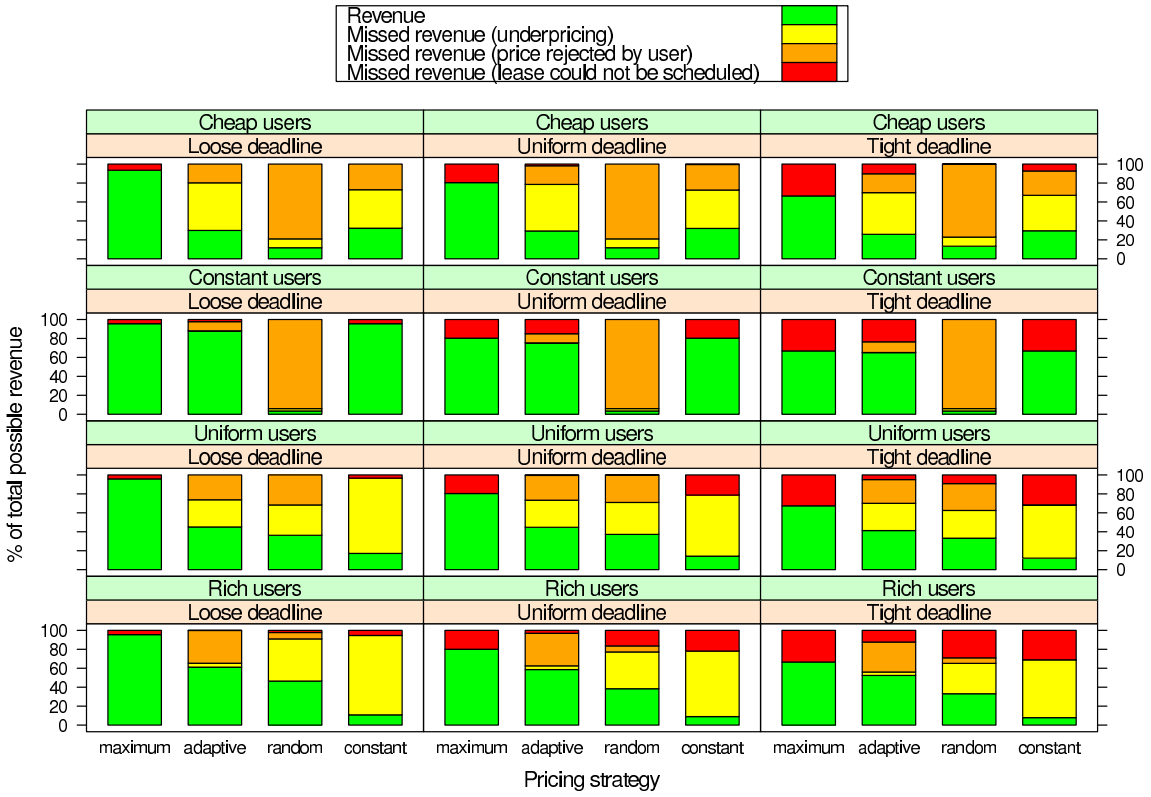


Figure 8.1: Effect of pricing strategy, distribution of r_u, and distribution of ω on revenue (BLUE2 workload).

8.3.2 Results

Figures 8.1 and 8.2 show the effect of the pricing strategies and the distribution of values of r_u on the revenue metrics. To focus on the effect of pricing strategies, I limit the discussion to the results with the least restrictive EARLY START distribution of δ. In both the BLUE2 and DS workloads, the ADAPTIVE strategy performs better than CONSTANT and RANDOM when dealing with UNIFORM USERS and RICH USERS, obtaining between 41.28% (TIGHT DEADLINE, UNIFORM USERS) and 61.17% (LOOSE DEADLINE, RICH USERS) of the total possible revenue in the BLUE2 workloads (61.25% and 64.06%, respectively, of the revenue obtained using the magical MAXIMUM strategy), and between 36.26% (TIGHT DEADLINE, UNIFORM USERS) and 65.74% (LOOSE DEADLINE, RICH USERS) in the DS workload (55.49% and

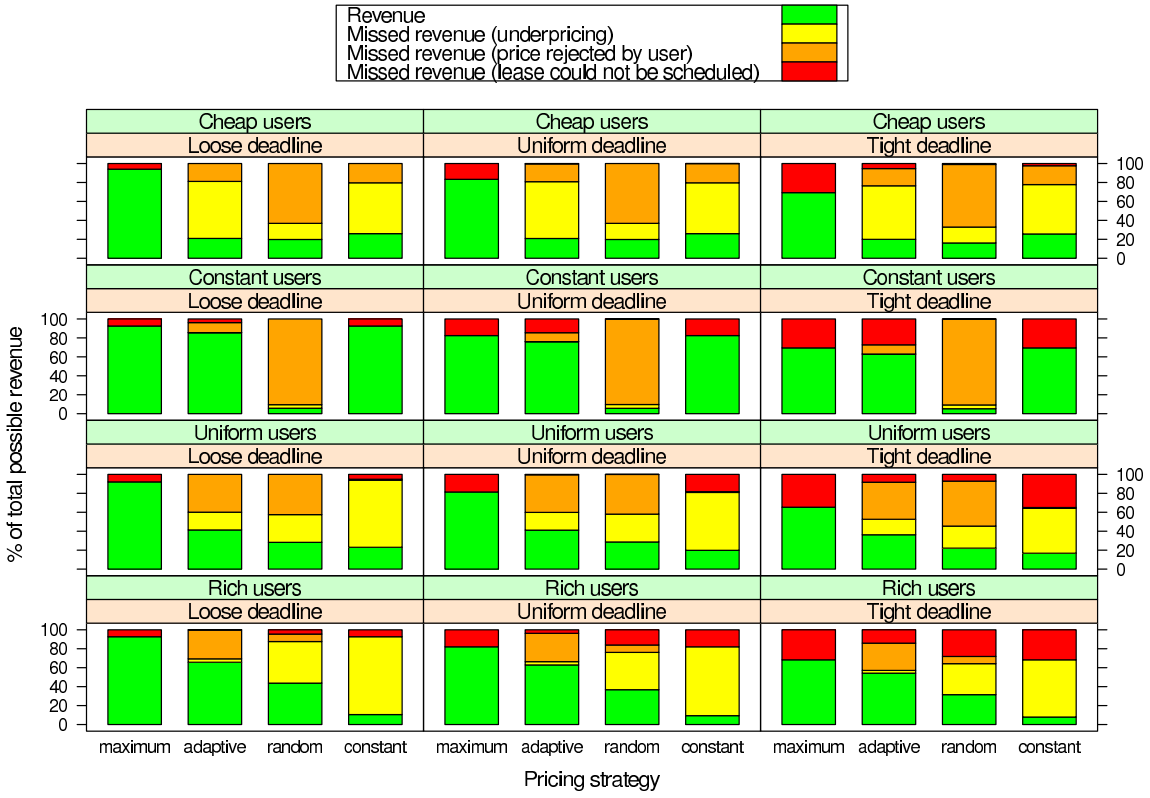
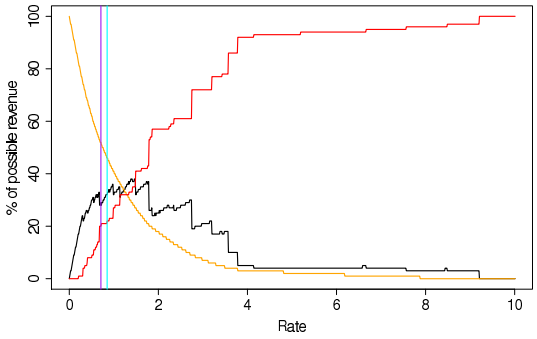


Figure 8.2: Effect of pricing strategy, distribution of r_u, and distribution of ω on revenue (DS workload).

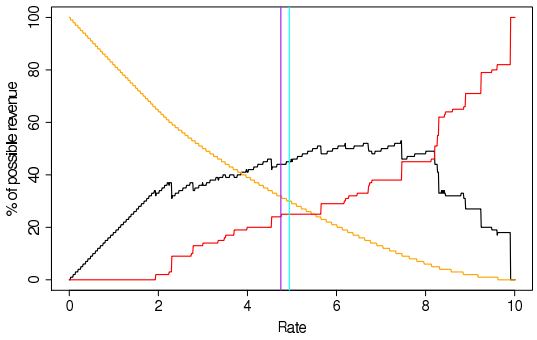
70.92% of the MAXIMUM revenue). With CONSTANT USERS, the ADAPTIVE strategy is closer to MAXIMUM, since the lack of variability in the user rates allows the ADAPTIVE algorithm to arrive at a rate that maximizes revenue for all requests, although some leases are rejected by users at the start of the workload as the algorithm converges on this constant rate (which includes raising the rate to see if users will accept it).

However, in the CHEAP USERS cases, more revenue is lost to underpricing than is actually earned. Although this effect could be caused by the adaptive algorithm estimating a rate lower than what most users are willing to pay, an inspection of what happens when the entire workload is priced at a constant rate between 0.1 and 10.0 in 0.01 increments (Figures 8.3 and 8.4) reveals the actual cause. In the CHEAP USERS case or, in general, when most user rates are skewed toward a small value, the rate at which we gain additional revenue from leases we are accepting (by not underpricing them) is not enough to compensate the increasing number of leases rejected by users. Thus, the equilibrium point is reached at a smaller total revenue than the other cases. These graphs also highlight that aiming for the median of the values of r_u may not necessarily be a good strategy, as shown in the RICH USERS case, where higher revenue is possible by setting a rate smaller than that arrived at by the adaptive algorithm.

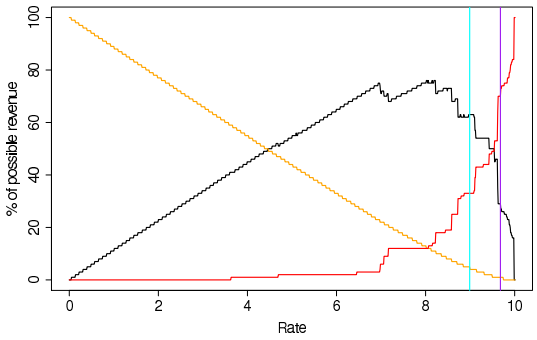
Finally, Figure 8.5 shows the relation between revenue and utilization for each pricing strategy. These results emphasize that high utilization does not necessarily imply high revenue. If we ignore CONSTANT USERS, given that it is arguably unrealistic to expect the entire user population to choose the same rate, the adaptive pricing strategy shows the least variability in terms of utilization; regardless of user population, and values of δ and ω, utilization remains between 35.17% and 47.54% (in the BLUE2 workload) and between 17.25% and 33.83% (in the DS workload). Thus, in these workloads, the adaptive pricing strategy is able to obtain higher revenue than the other strategies, while still leaving more than half of the resources available for other uses.



(a) Cheap users



(b) Uniform users



(c) Rich users

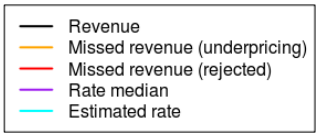
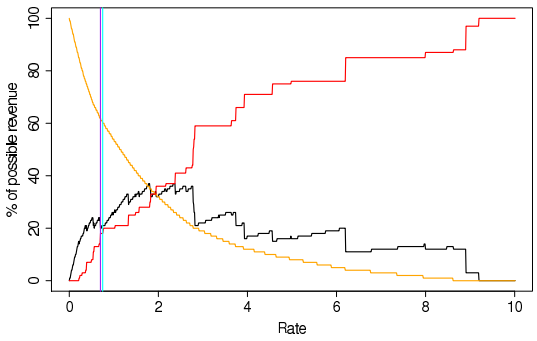
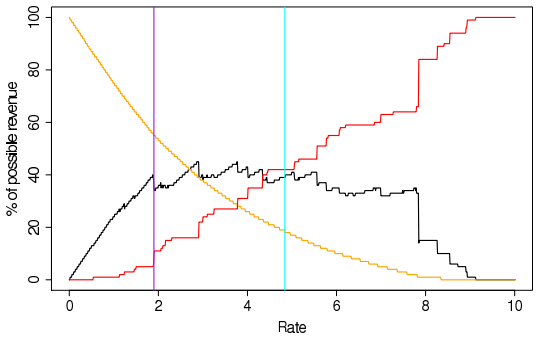


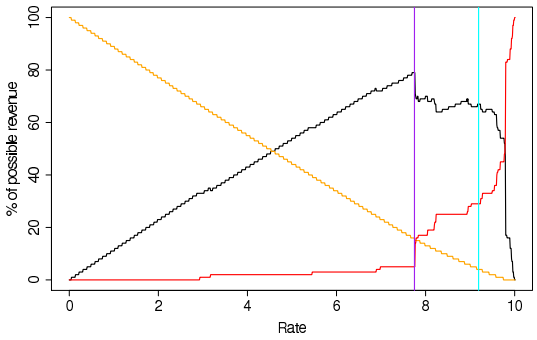
Figure 8.3: Effect on revenue when pricing the DS workload at a constant rate, in the best-case scenario that every single lease can be satisfied (making the value of δ and ω irrelevant).



(a) Cheap users



(b) Uniform users



(c) Rich users

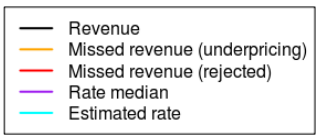
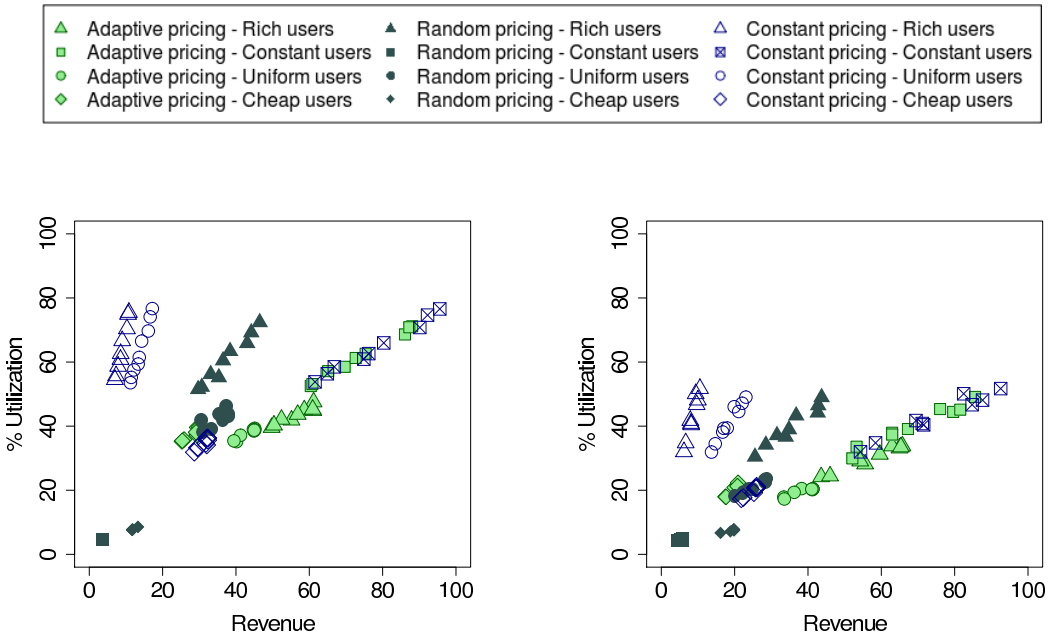


Figure 8.4: Effect on revenue when pricing the DS workload at a constant rate, in the best-case scenario that every single lease can be satisfied (making the value of δ and ω irrelevant).



(a) BLUE2 workload

(b) DS workload

Figure 8.5: Revenue and utilization categorized according to pricing strategy and distribution of r_u. Each point within a category represents a run with PREEMPTION WITH SUSPEND/RESUME and each of the possible combinations of values of δ and ω.

8.4 Conclusions

This chapter has presented a model for pricing leases and an adaptive pricing strategy by which resource providers determine lease prices via analysis of user behaviour. The experimental evaluation shows that, in most cases, this strategy uses fewer resources to obtain more revenue than other baseline pricing strategies. Thus, there may be value to adaptive resource pricing.

However, in this current model, resource providers adapt to resource consumer behaviour, but resource consumers do not respond to resource provider behaviour. Adaptation by consumers to resource provider pricing strategies seems likely to occur in practice, and has the potential to alter the results shown here. I hope to explore these issues in future work, perhaps via direct experimentation with real users.

Another future step will be dealing with unsold capacity. As highlighted in this work,

supporting QoS and prices can limit the number of leases a provider is willing to accept, resulting in unused capacity. However, the experiments rely only on existing workloads and, although the model includes the rate that users are willing to pay for their leases, it does not model how other users might consume leftover capacity. A recent relevant development in this line is Amazon EC2's spot pricing, which permits unused capacity in EC2 to be purchased at a lower rate but, accordingly, provides fewer QoS guarantees. However, unlike Amazon, I am interested in exploring a model that can combine both high QoS leases, such as ARs or leases with deadlines, and lower (but not too low) QoS leases that are used as 'economic backfilling' for unused capacity and which leverage VM suspend/resume/migrate to survive preemptions (unlike EC2 spot instances, which are killed when the spot rate rises above the consumer's rate).

I will also explore additional pricing strategies, including some that consider the requested QoS level when setting prices –by, for example, charging a higher rate for leases with tighter deadlines. A possible strategy would be by applying a surcharge for preempting other leases. Even if a preempted lease still meets its deadline, the preemption involves a period of time spent suspending and resuming the lease and during which the physical resources are essentially idling (as they are occupied with overhead tasks, not actual computation). Since a tighter deadline increases the likelihood that other leases will have to be preempted to accommodate it, this surcharge is an indirect way of charging for higher QoS.

Finally, I currently make the assumption that, once a lease is accepted, the agreement to satisfy the terms of the lease cannot be broken. In other words, the scheduler will not take any action that violates any lease terms. In the future, I will relax this assumption by incorporating penalties into the model, allowing the scheduler to breach a lease in exchange for paying a penalty to the user, enabling the provider to offer the user several possible prices, each with a different penalty: higher penalties would be charged at a higher price but would make it less likely that the provider would breach the lease terms.

CHAPTER 9

CONCLUSIONS

At the outset of this dissertation, I presented four goals that a resource provisioning model and architecture supporting multiple resource provisioning scenarios should meet:

G1-ResProv **Provide an abstraction focused solely on resource provisioning**

G2-HwSwAvail **Provision hardware, software, and availability**

G3-Reconcile **Reconcile requirements of different types of leases**

G4-ModelVirt **Model virtual resources accurately and schedule them efficiently**

In this dissertation, I have presented work that meets, to some degree, all these goals. Goal G1-ResProv is met by the choice of a leasing abstraction that is not tied to a specific provisioning use case but, instead, provides a general-purpose abstraction for resource provisioning. Goal G2-HwSwAvail is met by the choice of virtual machines as an implementation vehicle, since they can be used to provision software, hardware, and availability. However, as pointed out in the Preface, this was the "easy" part. The crux of this dissertation was to show that the use of virtual machines and leases can support multiple resource provisioning scenarios simultaneously and efficiently.

As far as meeting goal G3-Reconcile, I have shown that the use of leases and virtual machines, while leveraging their suspend/resume capability, can overcome the utilization problems typically encountered when combining best-effort leases and advance reservation leases. The results presented in this dissertation show that, even when assuming that workloads will run 5% slower on VMs, a VM-based approach still results in consistently better performance, in terms of the time required to run the entire workload, than using a resource scheduler that does not support preemption. Compared to a scheduler that does support preemption, the performance is only slightly worse, but allows resource providers to offer

more flexible lease terms, particularly concerning the software environments they can provide to resource consumers, given that non-VM-based schedulers with preemption require either making applications checkpointing-aware or using a checkpointing-capable OS.

However, although the use of VMs proves to be an effective vehicle for meeting goal G3-RECONCILE, they introduce new overheads, such as transferring potentially large disk images and, when using suspend/resume, saving and restoring a VMs entire memory state to and from disk. This dissertation has shown that these overheads cannot be ignored, and must be accurately modelled and scheduled (G4-MODELVIRT). Ignoring the overhead of transferring disk images can result in longer waiting times than can be unacceptable in the case of advance reservations. I have shown that treating this overhead as a separate scheduling problem can guarantee that disk images arrive on time, and that disk image reuse strategies can palliate, although not entirely eliminate, this overhead. On the other hand, ignoring the overhead of suspension and resumption can delay the start of leases that depend on those operations being completed by a certain time (e.g., when preempting resources for an advance reservation). I have presented a model for predicting these suspension and resumption times under a variety of conditions (multiple VMs per physical node, local and global filesystems, etc.), allowing the scheduler to allocate enough time for these operations, although my model still results in inaccurate estimations in certain cases.

Finally, meeting goal G3-RECONCILE also requires providing some form of lease admission control. Otherwise, users would have no incentive to request anything other than leases with the highest QoS guarantees (such as advance reservation leases or deadline leases). The results in Chapter 5 highlighted how the injection of AR lease requests only had a significant impact on performance once the site became overutilized. Of course, in these experiments the number of injected AR leases was controlled by me; in a real-world scenario, the resource provider will need some way of deciding what leases to reject and thus avoid the site from becoming overutilized. I proposed a price-based approach, where the resource provider

prices leases, which can then be accepted or rejected by users. So far, I have only explored strategies for maximizing revenue, showing that, in most cases, an adaptive pricing strategy uses fewer resources to obtain more revenue than other baseline pricing strategies.

So, have I met the four goals I set for this dissertation? There still remains much work to be done, and I cannot claim to have addressed these goals definitively. However, this dissertation has shown that a VM-based approach to supporting multiple types of leases has merit, and I have provided substantial advances towards a resource provisioning solution that meets these four goals.

9.1 Future work

As I continue the research presented in this dissertation, my future efforts will focus on three lines of work, described below.

9.1.1 Improving the model

In my work so far, I have presented a resource model that can be used by a scheduler (such as Haizea) to schedule certain operations more accurately, like disk image transfers and suspend/resume operations. However, there are several ways in which this model can be refined.

First of all, as highlighted in Chapter 7, sometimes the model will incorrectly predict the time to perform an operation because of unforeseeable factors, like an enactment command that takes an inordinately long time to be processed. One part of addressing this issue is to make Haizea more tolerant to failures. As currently implemented, if an operation takes longer than expected, Haizea just marches along merrily, as if nothing had happened. Instead, it should detect the failure, and modify the schedule if necessary. As of this writing, work is already underway to add this functionality to Haizea.

However, the model itself should account for the uncertainty in predicting the time to

complete an operation. Even if Haizea can detect and react to operations that take longer to complete, a delay in suspending a VM or in transferring a disk image can still result in an AR being unable to start on time. This can be palliated by allocating some buffer time to account for possible delays, with the tradeoff that, when operations complete on time, resources will be left idling during that buffer time. Thus, I am interested in researching how the probability of not breaching lease agreements varies with the aggressiveness of the model (i.e., how much buffer time is allocated) and ways in which the number of lease breaches can be minimized.

Next, my model currently assumes that software environments are only deployed as disk images transferred from a single image repository. Although this is a reasonable assumption, I am interested in exploring a more general model of "lease preparation", including systems that prepare disk images on-the-fly and distribution of disk images via P2P mechanisms (such as BitTorrent) that do not depend on a single image repository. Additionally, the disk transfer preparation model has been implemented and tested in simulation, but is not currently integrated with OpenNebula (when Haizea runs in OpenNebula mode, images must be predeployed on the physical nodes, or accessible via a global filesystem). I will pursue this integration so that, similar to our experiences testing the suspension/resumption mechanisms with OpenNebula, I can further refine the disk image transfer mode based on observations on real hardware.

Finally, although I have run Haizea with OpenNebula on modest sites (of up to 64 cores), I am interested in researching if the model, and Haizea itself, can scale to larger sites (in the order of thousands of nodes), and what challenges arise when supporting many different types of leases at those scales.

9.1.2 Further exploration of price-based policies

Towards the end of my dissertation work, I started to explore the use of price-based policies to determine if a lease should be accepted or not. My first approach, however, is rather limited, as it looks at this problem mostly from the perspective of how a resource provider can maximize revenue. However, some resource providers may be interested in maximizing utilization, while generating enough revenue to amortize their resources. The adaptive pricing policy presented in this dissertation determines the price based on the amount of resources requested (at a rate determined based on users' past behaviour), but not on whether the lease has a tight deadline or would involve preempting other resources (factors that, as shown in previous chapters, can impact utilization).

Thus, I intend to explore more pricing policies, particularly those where the price is determined based on the level of quality of service requested by the user. I am particularly interested in incorporating a negotiation phase to the lifecycle of leases, where a user requests a lease, and the resource provider provides a price for that lease, but also provides alternative prices with different levels of service. If leases can be breached, with a penalty paid to the user when this happens, the resource provider can also provide different prices based on the amount of the penalty: leases with a higher penalties would be charged at a higher price but would make it less likely that the provider would breach the lease terms.

Whereas my future work on refining the resource model can result in specific improvements on how leases are scheduled in Haizea, and the quality of service that users receive, the work on price-based policies is more speculative. It will be hard to reach conclusive results about the effectiveness of these policies until they can be tested with real users, but I believe that future results, even just in simulation, can provide valuable insights to resource providers on how to price leases.

9.1.3 Haizea "in the wild"

I also plan to continue work on Haizea and to push for its further adoption in production environments. Although this involves more engineering work than research work, I believe it will ultimately uncover interesting research questions that only arise once we leave the comfortable shell of assumptions and controlled environments, and have to deal with real users in the wild. Some of the issues I intend to address in Haizea have already been mentioned above: making Haizea more tolerant to failures and integrating the disk image transfer scheduling functionality with OpenNebula. Another issue affecting adoption in production environments that I am particularly interested in is the lack of leasing semantics in remote cloud interfaces, such as EC2 and OCCI, which are geared towards immediate provisioning of VMs. I am interested in exploring how these interfaces, but specially OCCI as an open standard, could be extended to support more use cases, such as requesting advance reservations on a cloud. This, in turn, may result in interesting problems related to meta-scheduling and co-allocations across clouds, similar to those that have already been explored in computational grids.

APPENDIX A

REPRODUCIBILITY OF RESULTS

This appendix describes how to reproduce the simulation results presented in this dissertation. First, Section A.1 describes how to set up the environment to run the experiments, and describes some naming and directory structure conventions followed in this appendix. Next, Section A.2 how to obtain the version of Haizea used in the experiments. Sections A.3 explains to generate the workloads used in the experiments. Section A.4 explains how to generate the configuration files for the experiments. Section A.5 describes how to run the experiments and generate the raw data. Section A.6 describes the format of the raw data produced by the experiments, and provides a link to the raw data I obtained in our own run of the experiments, and Section A.7 explains how to generate the dissertation's graphs and tables from the raw data. Finally, Section A.8 explains how to reproduce the non-simulated results in the dissertation.

To reproduce the results and graphs by rerunning all the simulations, all the instructions must be followed sequentially from beginning to end. To analyse and visualize my copy of the data without having to rerun all the simulations, skip to Section A.6.

A.1 Prerequisites

The experiments require a Linux environment with the following software installed:

- Python (at least 2.5)
- mxDateTime 3.1.0 (http://www.egenix.com/products/python/mxBase/mxDateTime/), part of the eGenix.com mx Base Distribution.
- Mako Templates for Python 0.2.2 (http://www.makotemplates.org/).
- R 2.8.x (http://www.r-project.org/).

Although it is possible to run all the simulations on a single machine, I recommend running them on a compute cluster; as a reference point, I used a 200-core Condor pool to run the experiments, and the time to complete all the simulations in this dissertation takes roughly five days. The instructions provided here are valid both for running in a single node (which should still allow a small sample of results to be produced) or in a compute cluster (in which case I recommend the following instructions be done on a shared filesystem; instructions on how to generate a Condor submission file for the simulations are provided below).

First, you will need to download some supplementary scripts I used to run the experiments. Create an empty directory, which we will refer to as `$EXP_DIR`. Download http://people.cs.uchicago.edu/~borja/dissertation/dissertation-exp.tgz and un-tar it in that directory. This will create the following directory structure:

- `bin`: Scripts.
- `configs`: Generated configuration files. Initially empty.
- `data`: Data files generated by experiments. Initially empty.
- `data_processed`: Data files generated by experiments. Initially empty.
- `etc`: Master configuration files.
- `graphs`: Graphs. Initially empty.
- `scripts`: Generated scripts. Initially empty.
- `traces`: Workload files. Initially empty.

Unless otherwise indicated, all instructions in this document are relative to `$EXP_DIR`. I will use the following notation to denote commands that must be run in the shell:

```
mkdir dissertation
cd dissertation
export EXP_DIR=‘pwd‘
wget http://people.cs.uchicago.edu/~borja/dissertation/dissertation-exp.tgz
tar xvzf dissertation-exp.tgz
```

A.2 Haizea

I ran the simulations using an experimental branch of Haizea. To download the exact version used to produce the results in this dissertation, run the following:

```
svn co https://phoenixforge.cs.uchicago.edu/svn/haizea/branches/1.1@839 haizea
```

And set the following environment variables

```
export PATH=$EXP_DIR/haizea/bin:$PATH
export PYTHONPATH=$EXP_DIR/haizea/src:$PYTHONPATH
```

A.3 Workloads

The workloads are based on job traces available on the Parallel Workloads Archive. You will need to download the SDSC Blue and DataStar traces:

```
cd traces
wget http://www.cs.huji.ac.il/labs/parallel/workload/l_sdsc_blue/\
SDSC-BLUE-2000-3.1-cln.swf.gz
gunzip SDSC-BLUE-2000-3.1-cln.swf.gz
wget http://www.cs.huji.ac.il/labs/parallel/workload/l_sdsc_ds/SDSC-DS-2004-1.swf.gz
gunzip SDSC-DS-2004-1.swf.gz
cd ..
```

Next, we need to create the BLUE1 and BLUE2 workloads by taking two 30 day segments of job submissions from the Blue trace and converting them to LWF files:

```
./bin/common/gen_trace_blue.sh 5:02:14:30 30 BLUE1
./bin/common/gen_trace_blue.sh 811:23:22:33 30 BLUE2
```

This script is a wrapper that instructs the `haizea-swf2lwf` command to:

- Select only the jobs from queues 1, 2, 3, 4. These are the cluster's main queues. Jobs submitted to these queues have access to up to 144 nodes.
- Scale the number of processors by 8. Each Blue node has eight processors and, as specified in the Parallel Workloads Archive, "The original log specifies the number of nodes each job requested and received. In the conversion this was multiplied by 8 to get the number of processors." Thus, we divide the number of processors by 8 to obtain the number of nodes.

We do the same for the DS trace:

```
./bin/common/gen_trace_ds.sh 49:16:42:21 30 DS
```

In this case, we instruct `haizea-swf2lwf` to do the following:

- Select only the jobs from queues 1, 2 (the cluster's main queues). Jobs submitted to these queues have access to up to 164 8-processor nodes.
- Scale the number of processors by 8. The nodes in the queues we used have eight processors and, as specified in the Parallel Workloads Archive, "The original log specifies the number of nodes each job requested and received. In the conversion this was multiplied by 8 or by 32 to get the number of processors, depending on which type of nodes was used." Since we are only using queues with 8-processor nodes, we divide the number of processors by 8 to obtain the number of nodes.

The generated LWF files will be in the `traces` directory:

```
traces/BLUE1.lwf
traces/BLUE2.lwf
traces/DS.lwf
```

A.3.1 AR injections

The experiments in Chapters 5 and 6 inject an artificially-generated workload of advance reservation leases into the `BLUE1` workload. These are generated using the `haizea-lwf--generate` command. We first create the configuration files needed by this command:

```
./bin/common/gen_inj_confs.py BLUE-INJ                              \
                              30:00:00:00                           \
                              05,10,15,20,25,30                     \
                              60                                    \
                              1h:3600,2h:7200,3h:10800,4h:14400     \
                              1800                                  \
                              small:1-24,medium:25-48,large:49-72   \
                              CPU:100,Memory:1024                   \
                              etc/blue.site                         \
                              819018246
```

The meaning of each parameter is the following:

- `BLUE-INJ`: Identifier.
- `30:00:00:00`: Duration of the workload (30 days).
- `05,10,15,20,25,30`: Values for ρ, as defined in Section 5.4.1.
- `60`: When generating the lease arrival interval, a random value (uniformly chosen) of seconds, between 0 and 60 (this parameter, is added to each interval.
- `1h:3600,2h:7200,3h:10800,4h:14400`: The values of δ, as defined in Section 5.4.1.
- `1800`: When generating the individual durations of the leases, a value is chosen in from a range spanning $\delta \pm 1800$s.
- `small:1-24,medium:25-48,large:49-72`: The values of ν, as defined in Section 5.4.1
- `CPU:100,Memory:1024`: Resources requested by each lease.

- `etc/blue.site`: XML file describing the resources in the Blue site.
- `819018246`: Random seed.

The configuration files are generated in the `configs` directory:

```
configs/BLUE-INJ__U05_Nlarge_D1h.conf
configs/BLUE-INJ__U05_Nlarge_D2h.conf
configs/BLUE-INJ__U05_Nlarge_D3h.conf
configs/BLUE-INJ__U05_Nlarge_D4h.conf
configs/BLUE-INJ__U05_Nmedium_D1h.conf
configs/BLUE-INJ__U05_Nmedium_D2h.conf
configs/BLUE-INJ__U05_Nmedium_D3h.conf
configs/BLUE-INJ__U05_Nmedium_D4h.conf
configs/BLUE-INJ__U05_Nsmall_D1h.conf
configs/BLUE-INJ__U05_Nsmall_D2h.conf
...
```

Each file specifies how an injected AR workload must be generated. The file name can be used to identify how each attribute is generated: `U` is ρ, `N` is ν and `D` is δ.

Next, we generate the injected AR workloads:

```
./bin/common/gen_injs.sh BLUE-INJ
```

The LWF files will be generated in the `traces/inj/` directory:

```
traces/inj/BLUE-INJ__U05_Nlarge_D1h.lwf
traces/inj/BLUE-INJ__U05_Nlarge_D2h.lwf
traces/inj/BLUE-INJ__U05_Nlarge_D3h.lwf
traces/inj/BLUE-INJ__U05_Nlarge_D4h.lwf
traces/inj/BLUE-INJ__U05_Nmedium_D1h.lwf
traces/inj/BLUE-INJ__U05_Nmedium_D2h.lwf
traces/inj/BLUE-INJ__U05_Nmedium_D3h.lwf
traces/inj/BLUE-INJ__U05_Nmedium_D4h.lwf
traces/inj/BLUE-INJ__U05_Nsmall_D1h.lwf
traces/inj/BLUE-INJ__U05_Nsmall_D2h.lwf
...
```

A.3.2 Image files

The experiments in Chapter 6 require that each lease specify a disk image to use. Instead of adding this information directly to the LWF file, I keep this information in separate *annotation files* which specify extra attributes that should be added to a lease (such as a disk image to use. We first generate the configuration files:

```
./bin/2imagetransfer/gen_img_annot_confs.py 4GB-img   \
                                            40        \
                                            4096      \
                                            2776613659
```

- `4GB-img`: Identifier.
- `40`: Total number of disk images.
- `4096`: Disk image size.
- `2776613659`: Random seed.

Next, we generate the annotation files:

```
./bin/2imagetransfer/gen_imgs.sh 4GB-img 50000
```

This script is a wrapper over `haizea-lwf-annotate`. The `50000` parameter specifies the number of entries each file should contain (this is an arbitrarily large number, as no workload has more than 50,000 leases). The annotation files are generated in the `traces` directory:

```
traces/img/4GB-img__pareto.lwfa
traces/img/4GB-img__uniform.lwfa
```

A.3.3 Deadline, start time, and price annotations

The experiments in Chapters 5 and 8 require that the BLUE2 and DS workloads specify the earliest start time of jobs, deadlines, or user rates. Since the logs from the Parallel Workloads Archive do not include this information, it is artificially generated. As with the disk images, this information is specified in an annotation file. To generate these annotation files, we first have to generate the configuration files that specify how the values in each file should be generated (e.g., the distribution of values of the deadlines, etc.)

We do this using the following script:

```
./bin/4pricing/gen_price_annot_confs.py BLUE2-ANNOT     \
                                        absolute 86400  \
                                        absolute 604800 \
                                        10              \
                                        276100674
```

The meaning of each parameter is the following:

- `BLUE2-ANNOT`: Identifier.
- `absolute 86400`: The start time of each lease is at most a day (86,400 seconds).
- `absolute 604800`: The deadline of each lease is at most a week (604,800 seconds).
- `10`: The highest user rate is $10.00
- `276100674`: Random seed.

The configuration files Annotation files are generated in the `configs` diractory:

```
configs/BLUE2-ANNOT__Sinvpareto_Dinvpareto_Mconstant.conf
configs/BLUE2-ANNOT__Sinvpareto_Dinvpareto_Minvpareto.conf
configs/BLUE2-ANNOT__Sinvpareto_Dinvpareto_Mpareto.conf
configs/BLUE2-ANNOT__Sinvpareto_Dinvpareto_Muniform.conf
configs/BLUE2-ANNOT__Sinvpareto_Dpareto_Mconstant.conf
configs/BLUE2-ANNOT__Sinvpareto_Dpareto_Minvpareto.conf
configs/BLUE2-ANNOT__Sinvpareto_Dpareto_Mpareto.conf
configs/BLUE2-ANNOT__Sinvpareto_Dpareto_Muniform.conf
configs/BLUE2-ANNOT__Sinvpareto_Duniform_Mconstant.conf
configs/BLUE2-ANNOT__Sinvpareto_Duniform_Minvpareto.conf
...
```

Each file specifies how an annotation file must be generated. The file name can be used to identify how each attribute is generated:

- Start time:
  - `Spareto`: Pareto distribution, skewed towards 0. Referred to as "Early Start" in Chapter 8.
  - `Suniform`: Uniform distribution. Referred to as "Uniform Start".
  - `Sinvpareto`: Pareto distribution, skewed towards the maximum (in this case, 24 hours). Referred to as "Late Start".
- Deadline:
  - `Dpareto`: Pareto distribution, skewed towards 0. Referred to as "Tight Deadline".
  - `Duniform`: Uniform distribution. Referred to as "Uniform Deadline".
  - `Dinvpareto`: Pareto distribution, skewed towards the maximum (in this case, a week). Referred to as "Late Deadline".
- User rate:
  - `Mconstant`: Users are always willing to pay at most $1.00 per hour. Referred to as "Constant Users".

- `Mpareto`: Pareto distribution, skewed towards $0.10. Referred to as "Cheap Users".
- `Muniform`: Uniform distribution. Referred to as "Uniform Users".
- `Minvpareto`: Pareto distribution, skewed towards the maximum ($10.00 in this case). Referred to as "Rich Users".

Next, we generate the annotation files themselves:

```
./bin/4pricing/gen_annots.sh BLUE2 BLUE2-ANNOT
```

This script is a wrapper over the `haizea-lwf-annotate` command, and should take a few minutes to run. The annotation files will be generated in the `traces/annot/` directory:

```
traces/annot/BLUE2-ANNOT__Sinvpareto_Dinvpareto_Mconstant.lwfa
traces/annot/BLUE2-ANNOT__Sinvpareto_Dinvpareto_Minvpareto.lwfa
traces/annot/BLUE2-ANNOT__Sinvpareto_Dinvpareto_Mpareto.lwfa
traces/annot/BLUE2-ANNOT__Sinvpareto_Dinvpareto_Muniform.lwfa
traces/annot/BLUE2-ANNOT__Sinvpareto_Dpareto_Mconstant.lwfa
traces/annot/BLUE2-ANNOT__Sinvpareto_Dpareto_Minvpareto.lwfa
traces/annot/BLUE2-ANNOT__Sinvpareto_Dpareto_Mpareto.lwfa
traces/annot/BLUE2-ANNOT__Sinvpareto_Dpareto_Muniform.lwfa
traces/annot/BLUE2-ANNOT__Sinvpareto_Duniform_Mconstant.lwfa
traces/annot/BLUE2-ANNOT__Sinvpareto_Duniform_Minvpareto.lwfa
...
```

Finally, we do the same thing for the DS workload:

```
./bin/4pricing/gen_price_annot_confs.py DS-ANNOT        \
                                        absolute 86400  \
                                        absolute 604800 \
                                        10              \
                                        2448108845
./bin/4pricing/gen_annots.sh DS DS-ANNOT
```

A.4 Experiment configuration files

Now that we have the workloads and the annotation files for the workload, we need to generate the Haizea configuration files for each simulation:

A.4.1 Chapters 5 (no deadlines) and 6

We generate the configuration files for these chapters using the following command:

```
./bin/1scheduling/gen_configfiles.sh BLUE1 BLUE-INJ
```

This command generates a "multiconfiguration" file from a configuration template file in `etc/template_1scheduling.conf` (see Haizea documentation for more details on "multiconfiguration" files) and then uses the `haizea-generate-configs` command to generate the individual configuration files, which are generated in the `configs/1scheduling` and `configs/2disktransfer` directories:

```
configs/1scheduling/BLUE1+BLUE-INJ.conf
configs/1scheduling/NOVM-NOSR_BLUE1+BLUE-INJ__U05_Nlarge_D1h.conf
configs/1scheduling/NOVM-NOSR_BLUE1+BLUE-INJ__U05_Nlarge_D2h.conf
configs/1scheduling/NOVM-NOSR_BLUE1+BLUE-INJ__U05_Nlarge_D3h.conf
configs/1scheduling/NOVM-NOSR_BLUE1+BLUE-INJ__U05_Nlarge_D4h.conf
...
configs/1scheduling/VM-PREDEPLOY_BLUE1+BLUE-INJ__U30_Nsmall_D1h.conf
configs/1scheduling/VM-PREDEPLOY_BLUE1+BLUE-INJ__U30_Nsmall_D2h.conf
configs/1scheduling/VM-PREDEPLOY_BLUE1+BLUE-INJ__U30_Nsmall_D3h.conf
configs/1scheduling/VM-PREDEPLOY_BLUE1+BLUE-INJ__U30_Nsmall_D4h.conf
configs/1scheduling/VM-PREDEPLOY_BLUE1.conf
```

A.4.2 Chapter 6

We generate the configuration files:

```
./bin/2imagetransfer/gen_configfiles.sh       BLUE1 BLUE-INJ 4GB-img
./bin/2imagetransfer/gen_configfiles-bw100.sh BLUE1                          \
                                              "BLUE-INJ__U10_Nmedium_D4h.lwf  \
                                               BLUE-INJ__U20_Nmedium_D3h.lwf  \
                                               BLUE-INJ__U30_Nmedium_D2h.lwf" \
                                               4GB-img
```

Which will be placed in `configs/2imagetransfer`. The files are created based on templates `etc/template_2imagetransfer_noreuse.conf` and `etc/template_2imagetransfer_reuse.conf`.

```
configs/2imagetransfer/BLUE1+4GB-img_noreuse-bw100.conf
configs/2imagetransfer/BLUE1+4GB-img_reuse-bw100.conf
configs/2imagetransfer/BLUE1+BLUE-INJ+4GB-img_noreuse.conf
configs/2imagetransfer/BLUE1+BLUE-INJ+4GB-img_reuse.conf
configs/2imagetransfer/VM-MULT_BLUE1+4GB-img__uniform.conf
...
configs/2imagetransfer/VM-REUSE-BW100_BLUE1+BLUE-INJ__U10_Nmedium_D4h+4GB-img__pareto.conf
configs/2imagetransfer/VM-REUSE-BW100_BLUE1+BLUE-INJ__U10_Nmedium_D4h+4GB-img__uniform.conf
configs/2imagetransfer/VM-REUSE-BW100_BLUE1+BLUE-INJ__U20_Nmedium_D3h+4GB-img__pareto.conf
configs/2imagetransfer/VM-REUSE-BW100_BLUE1+BLUE-INJ__U20_Nmedium_D3h+4GB-img__uniform.conf
configs/2imagetransfer/VM-REUSE-BW100_BLUE1+BLUE-INJ__U30_Nmedium_D2h+4GB-img__pareto.conf
configs/2imagetransfer/VM-REUSE-BW100_BLUE1+BLUE-INJ__U30_Nmedium_D2h+4GB-img__uniform.conf
```

There is one configuration file for each combination of workload and configuration profile. The multiconfiguration file specifies the NOVM–NOSR, VM–NOSR, VM–PREDEPLOY, VM–MULT, and VM–REUSE, as defined in Sections 5.4.3 and 6.4. The VM–REUSE–UNIFORM, VM–REUSE–SKEWED, VM–MULT–BW100, VM–REUSE–UNIFORM–BW100, and VM–REUSE–SKEWED–BW100 profiles are derived from certain configuration options (e.g., VM–MULT–BW100 is the VM–MULT profile with the bandwidth option in the configuration file set to 100).

A.4.3 Chapter 7

We generate the configuration files:

```
./bin/3suspendresume/gen_configfiles.sh BLUE1                              \
                                        "BLUE-INJ__U10_Nmedium_D4h.lwf  \
                                         BLUE-INJ__U20_Nmedium_D3h.lwf  \
                                         BLUE-INJ__U30_Nmedium_D2h.lwf"
```

Which will be placed in `configs/3suspendresume`. The files are created based on template `etc/template_3suspendresume.conf`.

```
configs/3suspendresume/baseline_BLUE1+BLUE-INJ__U10_Nmedium_D4h.conf
configs/3suspendresume/baseline_BLUE1+BLUE-INJ__U20_Nmedium_D3h.conf
configs/3suspendresume/baseline_BLUE1+BLUE-INJ__U30_Nmedium_D2h.conf
configs/3suspendresume/baseline_BLUE1.conf
configs/3suspendresume/BLUE1.conf
...
configs/3suspendresume/mem4096_pernode8_exclglobal_sh10_bw1000_BLUE1.conf
configs/3suspendresume/mem4096_pernode8_excllocal_sh10_bw1000_BLUE1+BLUE-INJ__U10_Nmedium_D4h.conf
configs/3suspendresume/mem4096_pernode8_excllocal_sh10_bw1000_BLUE1+BLUE-INJ__U20_Nmedium_D3h.conf
configs/3suspendresume/mem4096_pernode8_excllocal_sh10_bw1000_BLUE1+BLUE-INJ__U30_Nmedium_D2h.conf
configs/3suspendresume/mem4096_pernode8_excllocal_sh10_bw1000_BLUE1.conf
```

There is one configuration file for each combination of workload and configuration profile. The multiconfiguration file specifies the following profiles:

- `baseline`: The baseline used in the experiments, assuming that suspend/resume is instantaneous.
- `mem`m`_pernode`C`_excl`f`_sh`h`_bw`b: Every combination of parameters m, C, and f as defined in Section 7.2.3. For every combination, a shutdown time of $h = 10$ and a bandwidth of $b = 1000$ (Mbps) is used. A few profiles were also selected to explore other values of h and b.

A.4.4 Chapters 5 (with deadlines) and 8

We generate the configuration files:

```
./bin/4pricing/gen_conf.sh BLUE2 BLUE2-ANNOT
./bin/4pricing/gen_conf.sh DS DS-ANNOT
```

Which will be placed in `configs/4pricing`. The files are created based on template `etc/template_4pricing.conf`.

```
configs/adaptive_blue_30days+blue-annot__Sinvpareto_Dinvpareto_Mconstant.conf
configs/adaptive_blue_30days+blue-annot__Sinvpareto_Dinvpareto_Minvpareto.conf
configs/adaptive_blue_30days+blue-annot__Sinvpareto_Dinvpareto_Mpareto.conf
configs/adaptive_blue_30days+blue-annot__Sinvpareto_Dinvpareto_Muniform.conf
configs/adaptive_blue_30days+blue-annot__Sinvpareto_Dpareto_Mconstant.conf
...
configs/maximum_blue_30days+blue-annot__Sinvpareto_Dinvpareto_Mconstant.conf
configs/maximum_blue_30days+blue-annot__Sinvpareto_Dinvpareto_Minvpareto.conf
configs/maximum_blue_30days+blue-annot__Sinvpareto_Dinvpareto_Mpareto.conf
configs/maximum_blue_30days+blue-annot__Sinvpareto_Dinvpareto_Muniform.conf
configs/maximum_blue_30days+blue-annot__Sinvpareto_Dpareto_Mconstant.conf
...
```

There is one configuration file for each combination of workload, annotation file, and configuration profile. The master configuration file specifies the following profiles:

- `maximum-NOP`: Maximum pricing strategy (as described in the paper), no preemption
- `maximum-NOSR`: Maximum pricing strategy, preemption without suspend/resume
- `maximum`: Maximum pricing strategy, preemption with suspend/resume
- `constant-NOSR`: Constant pricing strategy (as described in the paper), preemption without suspend/resume
- `constant`: Constant pricing strategy, preemption with suspend/resume
- `random-NOSR`: Random pricing strategy (as described in the paper), preemption without suspend/resume
- `random`: Random pricing strategy, preemption with suspend/resume

- `adaptive-NOSR`: Adaptive pricing strategy (as described in the paper), preemption without suspend/resume
- `adaptive`: Adaptive pricing strategy, preemption with suspend/resume

A.5 Running the experiments

Next, we generate the scripts that will run Haizea with these configuration files:

```
./bin/common/gen_scripts.sh configs/1scheduling/BLUE1+BLUE-INJ.conf \
                            configs/1scheduling/ \
                            1scheduling_BLUE1

./bin/common/gen_scripts.sh configs/2imagetransfer/BLUE1+BLUE-INJ+4GB-img_noreuse.conf \
                            configs/2imagetransfer  \
                            2imagetransfer_BLUE1-noreuse
./bin/common/gen_scripts.sh configs/2imagetransfer/BLUE1+BLUE-INJ+4GB-img_reuse.conf \
                            configs/2imagetransfer  \
                            2imagetransfer_BLUE1-reuse
./bin/common/gen_scripts.sh configs/2imagetransfer/BLUE1+4GB-img_noreuse-bw100.conf \
                            configs/2imagetransfer  \
                            2imagetransfer_BLUE1-noreuse-bw100
./bin/common/gen_scripts.sh configs/2imagetransfer/BLUE1+4GB-img_reuse-bw100.conf \
                            configs/2imagetransfer  \
                            2imagetransfer_BLUE1-reuse-bw100

./bin/common/gen_scripts.sh configs/3suspendresume/BLUE1.conf \
                            configs/3suspendresume \
                            3suspendresume_BLUE1

./bin/common/gen_scripts.sh configs/4pricing/BLUE2+BLUE2-ANNOT.conf \
                            configs/4pricing \
                            4pricing_BLUE2
./bin/common/gen_scripts.sh configs/4pricing/DS+DS-ANNOT.conf \
                            configs/4pricing \
                            4pricing_DS
```

This will generate shell scripts that runs all the simulations sequentially:

```
scripts/run_1scheduling_BLUE1.sh
scripts/run_2imagetransfer_BLUE1-noreuse.sh
scripts/run_2imagetransfer_BLUE1-reuse.sh
scripts/run_2imagetransfer_BLUE1-noreuse-bw100.sh
scripts/run_2imagetransfer_BLUE1-reuse-bw100.sh
scripts/run_3suspendresume_BLUE1.sh
scripts/run_4pricing_BLUE2.sh
scripts/run_4pricing_DS.sh
```

Condor scripts are also generated:

```
scripts/condor_submit_1scheduling_BLUE1
scripts/condor_submit_2imagetransfer_BLUE1-noreuse
scripts/condor_submit_2imagetransfer_BLUE1-reuse
scripts/condor_submit_2imagetransfer_BLUE1-noreuse-bw100
scripts/condor_submit_2imagetransfer_BLUE1-reuse-bw100
scripts/condor_submit_3suspendresume_BLUE1
scripts/condor_submit_4pricing_BLUE2
scripts/condor_submit_4pricing_DS
```

Note that the Condor script is generated from a very basic template and will probably require some tweaks to work on your own Condor pool.

Each run of Haizea will generate a data file with information collected during the simulation. These files are placed in three directories:

- `data/1scheduling+2imagetransfer/`: Should contain XXX files.
- `data/3suspendresume/`: Should contain XXX files.
- `data/4pricing/`: Should contain 648 files.

These files contain Python-pickled objects. For ease of processing, we convert them into CSV files:

```
./bin/1scheduling/gen_csv.sh data/1scheduling+2imagetransfer/ BLUE1 BLUE-INJ
./bin/3suspendresume/gen_csv.sh data/3suspendresume/ BLUE1 BLUE-INJ
./bin/4pricing/gen_csv.sh data/4pricing/ BLUE2 BLUE2-ANNOT
./bin/4pricing/gen_csv.sh data/4pricing/ DS DS-ANNOT
```

This script is a wrapper over the `haizea-convert-data` command. The CSV files will be generated in the `data_processed` directory. The content of this files is described in Section A.6 below.

Finally, we need to generate the data used for Figures 8.3 and 8.4 (showing the types of revenue obtained when a constant rate is used, assuming no leases are rejected). We use a script that invokes Haizea's pricing policies over an entire workload, but without doing a full simulation (since we assume that no leases are rejected):

```
./bin/4pricing/rate_sweep.py traces/BLUE2.lwf \
                             traces/annot/BLUE2-ANNOT__Spareto_Dinvpareto_Mpareto.lwfa \
                             data_processed/4pricing/blue_ratesweep_cheapusers.csv
./bin/4pricing/rate_sweep.py traces/BLUE2.lwf \
                             traces/annot/BLUE2-ANNOT__Spareto_Dinvpareto_Muniform.lwfa \
                             data_processed/4pricing/blue_ratesweep_uniformusers.csv
./bin/4pricing/rate_sweep.py traces/BLUE2.lwf \
                             traces/annot/BLUE2-ANNOT__Spareto_Dinvpareto_Minvpareto.lwfa \
                             data_processed/4pricing/blue_ratesweep_richusers.csv
./bin/4pricing/rate_sweep.py traces/DS.lwf \
                             traces/annot/DS-ANNOT__Spareto_Dinvpareto_Mpareto.lwfa \
                             data_processed/4pricing/ds_ratesweep_cheapusers.csv
./bin/4pricing/rate_sweep.py traces/DS.lwf \
                             traces/annot/DS-ANNOT__Spareto_Dinvpareto_Muniform.lwfa \
                             data_processed/4pricing/ds_ratesweep_uniformusers.csv
./bin/4pricing/rate_sweep.py traces/DS.lwf \
                             traces/annot/DS-ANNOT__Spareto_Dinvpareto_Minvpareto.lwfa \
                             data_processed/4pricing/ds_ratesweep_richusers.csv
```

This script will also print out the median user rate and the rate the adaptive policy estimated. Make a note of these values, as they are used when generating the graphs.

A.6 Raw data

After following the above steps, you will have all the raw data used to produce the results in this dissertation. If you cannot run the simulations yourself, you can also download the

raw data obtained in my own run of the experiments:

```
wget http://people.cs.uchicago.edu/~borja/dissertation/raw_data_dissertation.tar
tar xvf raw_data_dissertation.tar
```

After running the simulations or downloading the raw data you should have the following files in the `data_processed` directory:

```
data_processed/1scheduling+2imagetransfer/BLUE1+BLUE-INJ_perlease.csv.gz
data_processed/1scheduling+2imagetransfer/BLUE1+BLUE-INJ_perrun.csv

data_processed/3suspendresume/BLUE1+BLUE-INJ_perlease.csv.gz
data_processed/3suspendresume/BLUE1+BLUE-INJ_perrun.csv

data_processed/4pricing/BLUE2+BLUE2-ANNOT_perlease.csv.gz
data_processed/4pricing/BLUE2+BLUE2-ANNOT_perrun.csv
data_processed/4pricing/BLUE2+BLUE2-ANNOT_utilization.csv.gz
data_processed/4pricing/DS+DS-ANNOT_perlease.csv.gz
data_processed/4pricing/DS+DS-ANNOT_perrun.csv
data_processed/4pricing/DS+DS-ANNOT_utilization.csv.gz
data_processed/4pricing/blue_ratesweep_cheapusers.csv
data_processed/4pricing/blue_ratesweep_richusers.csv
data_processed/4pricing/blue_ratesweep_uniformusers.csv
```

Before reading the files, or using them to generate graphs, some of these files need to be gunzipped:

```
gunzip data_processed/1scheduling+2imagetransfer/BLUE1+BLUE-INJ_perlease.csv.gz
gunzip data_processed/3suspendresume/BLUE1+BLUE-INJ_perlease.csv.gz
gunzip data_processed/4pricing/BLUE2+BLUE2-ANNOT_perlease.csv.gz
gunzip data_processed/4pricing/BLUE2+BLUE2-ANNOT_utilization.csv.gz
gunzip data_processed/4pricing/DS+DS-ANNOT_perlease.csv.gz
gunzip data_processed/4pricing/DS+DS-ANNOT_utilization.csv.gz
```

All the files are CSV files. The `perlease` files contains one row per lease per experiment run. The `perrun` files contain one row per experiment run. The `utilization` files contain utilization information for all the experiment runs (every time the utilization changes, an entry is added). The fields in each file are described next.

A.6.1 Common fields

The following fields are common to all the files:

- `profile`: Configuration profile (NOVM–NOSR, maximum-NOP, etc.)
- `tracefile`: Path of the workload file used.
- `annotationfile`: Path of the annotation file used.
- `injectfile`: Path of the injections file used.

A.6.2 `1scheduling+2imagetransfer` fields

The following fields appear only in files in the `data_processed/1scheduling+2imagetransfer/` directory:

- `utilization`: Parameter ρ, as defined in Section 5.4.1.
- `numnodes`: Parameter ν
- `duration`: Parameter δ
- `img_distr`: Distribution of disk images: `uniform` or `pareto`.
- `bandwidth`: Network bandwidth assumed in simulated resources

A.6.3 `3suspendresume` fields

The `utilization`, `numnodes`, `duration` fields, defined above, and the following fields appear only in files in the `data_processed/3suspendresume/` directory:

- `mem`: Parameter m, as defined in Section 7.2.3.
- `pernode`: Parameter C.

- `excl`: Parameter f.
- `shutdown`: Parameter h.

The `utilization`, `numnodes`, `duration` fields, defined earlier, are also used in the files found in the

A.6.4 `4pricing` fields

The following fields appear only in files in the `data_processed/4pricing/` directory:

- `tau`: Distribution of deadlines (pareto, invpareto, uniform; as described when the annotation files where generated)
- `delta`: Distribution of start times (pareto, invpareto, uniform; as described when the annotation files where generated)
- `mu`: Distribution of user rates (constant, pareto, invpareto, uniform; as described when the annotation files where generated)

A.6.5 `perrun` fields

The following fields appear only in `perrun` files. Note that some of these fields my be missing in some files (e.g., the `Revenue` field only appears in files in the `data_processed/4pricing/` directory).

- `Total requested leases`: Number of leases requested in this experiment.
- `Total accepted AR`: Number of AR leases accepted (and completed successfully).
- `Total best-effort completed`: Number of best-effort leases completed successfully.
- `Total completed leases`: Number of leases completed successfully.

- `Total rejected AR`: Number of leases completed successfully.
- `Total rejected leases (by user)`: Number of leases rejected because the user rejected the price offered by the provider.
- `Total rejected leases`: Number of leases rejected because they could not be scheduled.
- `all-best-effort`: Time at which the last best-effort lease was completed.
- `Revenue`: As defined in the paper. Note that this value, and all revenue values, are not normalized in this file relative to the total possible revenue (as described in the paper).
- `Surcharge`: This field can be ignored (we are not currently using surcharges in our experiments).
- `Missed revenue (undercharging)`: As defined in paper.
- `Missed revenue (reject)`: As defined in paper. Note that the paper refers to this metric as "Missed revenue (lease could not be scheduled)".
- `Missed revenue (reject by user)`: As defined in paper. Note that the paper refers to this metric as "Missed revenue (price rejected by user)".
- `Missed revenue (total)`: This field can be ignored, as we are not using "injection files" in these experiments (see Haizea documentation for more details on what these are)

A.6.6 `perlease` fields

The following fields appear only in `perlease` files. Note that some of these fields my be missing in some files.

- `lease_id`: Lease ID. Same as the Job Number in the original SWF file from the Parallel Workloads Archive.
- `Waiting time`: In best effort leases, the time between the submission time and the time the lease started.
- `Slowdown`: In best-effort leases, bounded slowdown of the lease (time the lease took to run from the requested start time to the actual end time, divided by the running time of the lease). Any lease with a duration less than 10 seconds is counted as having a duration of 10 seconds (for the purpose of computing the slowdown; this is how the "bounded" slowdown metric differs from the regular slowdown metric). This field will be empty if the lease was rejected.
- `State`: Final state of the lease. Can be "Done", "Rejected", or "Rejected by user".
- `Number of nodes`: Number of nodes requested.
- `Requested duration`: Requested duration in seconds.
- `Actual duration`: Actual duration in seconds. This field will be empty if the lease was rejected.
- `Start`: Start delay (parameter δ as defined in Section 5.5)
- `Deadline`: Maximum waiting time, which is used to determine the deadline (parameter ω)
- `Price`: Price charged for the lease. Empty if no pricing was done or if the lease was rejected.
- `simul_start_delta`: Same as "Deadline".
- `simul_userrate`: User rate.

- `simul_deadline_tau`: Deadline slack, as defined in the paper.
- `rate`: Rate that was used by the provider to compute the price of the lease.
- `rejected_price`: If the lease was rejected, this was the price the user rejected.
- `SWF_group`: For lease requests that were converted from an SWF file, this is field 13 in the original SWF file ("Group ID")
- `SWF_waittime`: Field 3 in the original SWF file ("Wait Time", the waiting time in the original workload)
- `SWF_runtime`: Field 4 in the original SWF file ("Run Time", the same as "Actual Duration", except this field always has a value, whereas "Actual Duration" will be empty if the lease was rejected)
- `SWF_execnumber`: Field 14 in the original SWF file ("Executable (Application) Number")
- `SWF_avgcputime`: Field 6 in the original SWF file ("Average CPU Time Used")
- `SWF_queue`: Field 15 in the original SWF file ("Queue Number")

A.6.7 `utilization` fields

The following fields appear only in `utilization` files.

- `time`: Time at which this utilization measurement was taken.
- `value`: Utilization, as defined in the paper.
- `average`: Time-weighted average utilization from time 0 up to this time.

A.6.8 `ratesweep` fields

These files don't have any of the common fields listed earlier, just the following fields:

- `rate`: Constant rate used to price the leases.
- `revenue`: Revenue, as defined in the paper. Note that this value is normalized relative to the total possible revenue.
- `underpricing`: Missed revenue (underpricing), as defined in the paper.
- `rejected`: Missed revenue (price rejected by user), as defined in the paper.

A.7 Graphs and tables

I used R to generate the graphs in the paper. To speed up the R scripts, we first convert the CSV files to R data files (this assumes the files have been ungzipped, as described in the previous section):

```
./bin/csv2rda.R data_processed/1scheduling+2imagetransfer/BLUE1+BLUE-INJ_perlease.csv \
                data_processed/1scheduling+2imagetransfer/BLUE1+BLUE-INJ_perlease.rda
./bin/csv2rda.R data_processed/1scheduling+2imagetransfer/BLUE1+BLUE-INJ_perrun.csv \
                data_processed/1scheduling+2imagetransfer/BLUE1+BLUE-INJ_perrun.rda

./bin/csv2rda.R data_processed/3suspendresume/BLUE1+BLUE-INJ_perlease.csv \
                data_processed/3suspendresume/BLUE1+BLUE-INJ_perlease.rda
./bin/csv2rda.R data_processed/3suspendresume/BLUE1+BLUE-INJ_perrun.csv \
                data_processed/3suspendresume/BLUE1+BLUE-INJ_perrun.rda

./bin/csv2rda.R data_processed/4pricing/BLUE2+BLUE2-ANNOT_perlease.csv \
                data_processed/4pricing/BLUE2+BLUE2-ANNOT_perlease.rda
./bin/csv2rda.R data_processed/4pricing/BLUE2+BLUE2-ANNOT_perrun.csv \
                data_processed/4pricing/BLUE2+BLUE2-ANNOT_perrun.rda
./bin/csv2rda.R data_processed/4pricing/BLUE2+BLUE2-ANNOT_utilization.csv \
                data_processed/4pricing/BLUE2+BLUE2-ANNOT_utilization.rda
./bin/csv2rda.R data_processed/4pricing/DS+DS-ANNOT_perlease.csv \
                data_processed/4pricing/DS+DS-ANNOT_perlease.rda
./bin/csv2rda.R data_processed/4pricing/DS+DS-ANNOT_perrun.csv \
                data_processed/4pricing/DS+DS-ANNOT_perrun.rda
./bin/csv2rda.R data_processed/4pricing/DS+DS-ANNOT_utilization.csv \
                data_processed/4pricing/DS+DS-ANNOT_utilization.rda
```

Since the utilization data files are rather big, and we're mostly interested in what the final utilization is, we extract that information and store it in a separate (smaller) file:

```
./bin/common/compute_util.R data_processed/4pricing/BLUE2+BLUE2-ANNOT_utilization.rda \
                            data_processed/4pricing/blue_util.rda 129600 2591876 blue
./bin/common/compute_util.R data_processed/4pricing/DS+DS-ANNOT_utilization.rda \
                            data_processed/4pricing/ds_util.rda 129600 2590365 ds
```

A.7.1 Chapter 5

To generate the graph for Figure 5.2, run the following:

```
./bin/1scheduling/graph_all-best-effort.R \
      graphs/1scheduling/all-best-effort.pdf \
      data_processed/1scheduling+2imagetransfer/BLUE1+BLUE-INJ_perrun.rda \
      notransfer
```

To generate the graphs for Figures 5.3 through 5.11, run the following:

```
./bin/1scheduling/graph_wait+slowdown.R \
      graphs/1scheduling \
      data_processed/1scheduling+2imagetransfer/BLUE1+BLUE-INJ_perlease.rda \
      notransfer
```

To obtain the data for Table 5.1, run the following:

```
./bin/1scheduling/table_all-best-effort.R \
      data_processed/1scheduling+2imagetransfer/BLUE1+BLUE-INJ_perrun.rda
./bin/1scheduling/table_avg_wait.R \
      data_processed/1scheduling+2imagetransfer/BLUE1+BLUE-INJ_perlease.rda
```

To generate the graph for Figure 5.12, run the following:

```
./bin/1scheduling/graph_utilization_maximum.R graphs/1scheduling/deadline_util.pdf  \
                                              data_processed/4pricing/blue_util.rda \
                                              data_processed/4pricing/ds_util.rda
```

To obtain the data for Table 5.2, run the following:

```
./bin/1scheduling/table_deadline.R data_processed/4pricing/BLUE2+BLUE2-ANNOT_perlease.rda
./bin/1scheduling/table_deadline.R data_processed/4pricing/DS+DS-ANNOT_perlease.rda
```

A.7.2 Chapter 6

To generate the graph for Figure 6.2, run the following:

```
./bin/1scheduling/graph_all-best-effort.R \
      graphs/2imagetransfer/all-best-effort.pdf \
      data_processed/1scheduling+2imagetransfer/BLUE1+BLUE-INJ_perrun.rda \
      transfer
```

To generate the graphs for Figures 6.3 through 6.11, run the following:

```
./bin/1scheduling/graph_wait+slowdown.R \
      graphs/2imagetransfer \
      data_processed/1scheduling+2imagetransfer/BLUE1+BLUE-INJ_perlease.rda \
      transfer
```

A.7.3 Chapter 7

To generate the graphs for Figure 7.4, run the following:

```
./bin/3suspendresume/graph_all-best-effort.R \
     graphs/3suspendresume/all-best-effort.pdf \
     data_processed/3suspendresume/BLUE1+BLUE-INJ_perrun.rda
```

To generate the graphs for Figure 7.5, run the following:

```
./bin/3suspendresume/graph_avg.R graphs/3suspendresume/avg.pdf \
                                 data_processed/3suspendresume/BLUE1+BLUE-INJ_perlease.rda
```

To obtain the data for Table 7.1, run the following:

```
./bin/3suspendresume/table_bw_shutdown.R \
     data_processed/3suspendresume/BLUE1+BLUE-INJ_perrun.rda
```

A.7.4 Chapter 8

To generate the graphs for Figures 8.1 and 8.2, run the following:

```
./bin/4pricing/graph_revenue.R graphs/4pricing/pricing_blue.pdf \
                               data_processed/4pricing/BLUE2+BLUE2-ANNOT_perrun.rda
./bin/4pricing/graph_revenue.R graphs/4pricing/pricing_ds.pdf \
                               data_processed/4pricing/DS+DS-ANNOT_perrun.rda
```

To generate the graphs for Figures 8.3 and 8.4, run the following:

```
./bin/4pricing/graph_rate_revenue.R graphs/4pricing/adaptive_cheapusers_blue.pdf \
                            data_processed/4pricing/blue_ratesweep_cheapusers.csv 0.71 0.85
./bin/4pricing/graph_rate_revenue.R graphs/4pricing/adaptive_uniformusers_blue.pdf \
                            data_processed/4pricing/blue_ratesweep_uniformusers.csv 4.75 4.94
./bin/4pricing/graph_rate_revenue.R graphs/4pricing/adaptive_richusers_blue.pdf \
                            data_processed/4pricing/blue_ratesweep_richusers.csv 9.68 8.99
./bin/4pricing/graph_rate_revenue.R graphs/4pricing/adaptive_cheapusers_ds.pdf \
                            data_processed/4pricing/ds_ratesweep_cheapusers.csv 0.70 0.75
./bin/4pricing/graph_rate_revenue.R graphs/4pricing/adaptive_uniformusers_ds.pdf \
                            data_processed/4pricing/ds_ratesweep_uniformusers.csv 1.90 4.84
./bin/4pricing/graph_rate_revenue.R graphs/4pricing/adaptive_richusers_ds.pdf \
                            data_processed/4pricing/ds_ratesweep_richusers.csv 7.75 9.19
```

The numbers provided to the R script are the median rate and estimated rate obtained when running the `rate_sweep.py` script earlier.

To generate the graphs for Figure 8.5, run the following:

```
./bin/4pricing/graph_revenue_utilization.R graphs/4pricing/revenue_utilization_blue.pdf \
                                    data_processed/4pricing/BLUE2+BLUE2-ANNOT_perrun.rda \
                                    data_processed/4pricing/blue_util.rda
./bin/4pricing/graph_revenue_utilization.R graphs/4pricing/revenue_utilization_ds.pdf \
                                    data_processed/4pricing/DS+DS-ANNOT_perrun.rda \
                                    data_processed/4pricing/ds_util.rda
```

These scripts will also print the minimum and maximum utilization values quoted in the text.

The following script determines what cases produce the best and worst revenue (when dealing with Uniform Users and Rich Users), also quoted in the text:

```
./bin/4pricing/misc_stats.R data_processed/4pricing/BLUE2+BLUE2-ANNOT_perrun.rda
./bin/4pricing/misc_stats.R data_processed/4pricing/DS+DS-ANNOT_perrun.rda
```

A.8 Non-simulated results

Whereas the previous sections should allow you to replicate this dissertation's simulated results exactly, reproducing the non-simulated results presented in Chapter 7 is not straightforward and may result in different results, since the experiments are dependent on the hardware and software configuration of the cluster where the experiments were run. Nonetheless, I am providing the scripts I used to run these experiments, with some minor modifications to remove references to our cluster's directory structure, hostnames, etc. Although they may not work out-of-the-box in other clusters, they may be of use to understand the experiments in more detail, or to set up similar experiments. Note that these scripts also assume that the experiments will run on five dual-processor quad-core machines, with OpenNebula 1.0 installed on one of them, and the rest running Xen 3.2. Other setups will likely involve non-trivial modifications to the scripts.

The required scripts can be downloaded at the following URL: http://people.cs.uchicago.edu/~borja/dissertation/dissertation-exp-nonsim.tgz. This file also contains the raw data used to produce the results in Chapter 7.

First, you will need to create 64 VM disk images: 32 for the AR lease and 32 for the best-effort lease. Networking must be configured in the images themselves, with IPs in the same /24 subnet. For example, assuming the images are created in `/images/ar/` and `/images/besteffort/`, the contents of these directories should look like this:

```
/images/ar/ar-01/disk.img
/images/ar/ar-01/swap.img
/images/ar/ar-02/disk.img
/images/ar/ar-02/swap.img
...
/images/ar/ar-32/disk.img
/images/ar/ar-32/swap.img
/images/besteffort/besteffort-01/disk.img
/images/besteffort/besteffort-01/swap.img
/images/besteffort/besteffort-02/disk.img
/images/besteffort/besteffort-02/swap.img
...
/images/besteffort/besteffort-32/disk.img
/images/besteffort/besteffort-32/swap.img
```

Next, we will use the provided `gen_configs.py` script to generate the OpenNebula templates used to request the VMs that comprise each lease, and the Haizea configuration file. The first lines of the script contain a number of variables that you will have to modify to match your site's configuration. Once you have done this, run the following commands:

```
./gen_configs.py -m 512 -d configs -z etc/haizea-template.conf
./gen_configs.py -m 768 -d configs -z etc/haizea-template.conf
./gen_configs.py -m 1024 -d configs -z etc/haizea-template.conf
```

The `-m` specified the amount of memory each VM will request, the `-d` parameter specifies where the OpenNebula templates will be created, and the `-z` parameter specifies the template

for the Haizea configuration file. The provided template should work with the same version of Haizea used for the simulation results. The exact SVN revision used for the original experiments is 541 (although the provided Haizea template won't actually work with that old version; the actual file used in the experiments is provided as etc/haizea-template-orig.conf).

Next, we run the experiments themselves. First, edit file `etc/hosts` to contain the names of hosts managed by OpenNebula. The user account you use to run the experiments must have passwordless SSH access to those nodes (that same user account must exist on all the worker nodes; ideally, the user must have a shared home directory across all the nodes). Additionally, you must copy the provided `clean_one` script to the same path on all hosts. Finally, the provided `run.py` script has several variables in the top of the file that you must edit to match the configuration of your site (e.g., the location of the `clean_one` script)

To run the experiments, run the following command:

```
./run.py -f etc/run.local
```

The experiments will take several hours to run. If an experiment fails, an e-mail will be sent to the address specified in the `run.py` scripts. If there is a failure that requires stopping all subsequent runs (e.g., a host goes down), you can resume the experiments (without repeating those that finished successfully) like this:

```
./run.py -f etc/run.local --resume
```

The above commands only run the experiments with suspensions and resumptions taking place to the local filesystem. To run the experiments on a global filesystem:

```
./run.py -f etc/run.global
```

The raw data collected during the experiment will be stored in the `data` directory. A copy of the data I obtained in the experiments is included in the `dissertation-exp-nonsim.tgz`

file. This is the data you will use to produce the graphs included in the dissertation. Before doing this, run the following commands to extract certain information (times to suspend, etc.) from the data:

```
./summarize_results.py -d data/
```

The summarized data is placed in the `data_processed` directory.

To generate Figure 7.2, run the following:

```
./graph-timeaccuracy.R
```

To generate Figure 7.3, run the following:

```
./graph-individualtimes.R
```

Graphs are saved in `graphs` directory.

REFERENCES

[1] S. Adabala, V. Chadha, P. Chawla, R. Figueiredo, J. Fortes, I. Krsul, A. Matsunaga, M. Tsugawa, J. Zhang, M. Zhao, L. Zhu, and X. Zhu. From virtualized resources to virtual computing grids: The In-VIGO system. *Future Gener. Comput. Syst.*, 21(6):896–909, June 2005.

[2] A. Andrieux, K. Czajkowski, A. Dan, K. Keahey, H. Ludwig, T. Nakata, J. Pruyne, J. Rofrano, S. Tuecke, and M. Xu. Web services agreement specification (WS-Agreement).

[3] N. Bobroff, A. Kochut, and K. Beaty. Dynamic placement of virtual machines for managing SLA violations. *Integrated Network Management, 2007. IM '07. 10th IFIP/IEEE International Symposium on Integrated Management*, pages 119–128, 2007.

[4] T. E. Carroll and D. Grosu. An incentive-compatible mechanism for scheduling non-malleable parallel jobs with individual deadlines. In *ICPP '08: Proceedings of the 2008 37th International Conference on Parallel Processing*, pages 107–114, Washington, DC, USA, 2008. IEEE Computer Society.

[5] C. Castillo, G. N. Rouskas, and K. Harfoush. Efficient implementation of best-fit scheduling for advance reservations and qos in grids. In *Proceedings of the First IEEE/IFIP International Workshop on End-to-end Virtualization and Grid Management (EVGM 2007)*, 2007.

[6] C. Castillo, G. N. Rouskas, and K. Harfoush. On the design of online scheduling algorithms for advance reservations and qos in grids. In *IPDPS*, pages 1–10. IEEE, 2007.

[7] C. Castillo, G. N. Rouskas, and K. Harfoush. Resource co-allocation for large-scale distributed environments. In *HPDC '09: Proceedings of the 18th ACM international symposium on High performance distributed computing*, pages 131–140, New York, NY, USA, 2009. ACM.

[8] W. S. Cleveland. Lowess: A program for smoothing scatterplots by robust locally weighted regression. *The American Statistician*, 35(54), 1981.

[9] K. Czajkowski, I. Foster, and C. Kesselman. Resource co-allocation in computational grids. In *HPDC '99: Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing*, page 37. IEEE Computer Society, 1999.

[10] B. DasGupta and M. A. Palis. Online real-time preemptive scheduling of jobs with deadlines. In *APPROX '00: Proceedings of the Third International Workshop on Approximation Algorithms for Combinatorial Optimization*, pages 96–107, London, UK, 2000. Springer-Verlag.

[11] W. Emeneker and D. Stanzione. Efficient Virtual Machine Caching in Dynamic Virtual Clusters. In *SRMPDS Workshop, ICAPDS 2007 Conference*, December 2007.

[12] N. Fallenbeck, H.-J. Picht, M. Smith, and B. Freisleben. Xen and the art of cluster scheduling. In *VTDC '06: Proceedings of the 1st International Workshop on Virtualization Technology in Distributed Computing*. IEEE Computer Society, 2006.

[13] U. Farooq, S. Majumdar, and E. W. Parsons. Efficiently scheduling advance reservations in grids. Technical Report SCE-05-14, Department of Systems and Computer Engineering, Carleton University, 2005.

[14] U. Farooq, S. Majumdar, and E. W. Parsons. Impact of laxity on scheduling with advance reservations in grids. *Modeling, Analysis, and Simulation of Computer Systems, International Symposium on*, 0, 2005.

[15] D. Feitelson. Analyzing the root causes of performance evaluation results. Technical report, School of Computer Science and Engineering, Hebrew University, 2002.

[16] D. Feitelson, L. Rudolph, and U. Schwiegelshohn. Parallel job scheduling – a status report. *10th Workshop on Job Scheduling Strategies for Parallel Processing, New-York, NY.*, 2004.

[17] D. G. Feitelson and L. Rudolph. Metrics and benchmarking for parallel job scheduling. *Lecture Notes in Computer Science*, 1459:1+, 1998.

[18] I. Foster, C. Kesselman, C. Lee, R. Lindell, K. Nahrstedt, and A. Roy. A distributed resource management architecture that supports advance reservations and co-allocation. In *Proceedings of the International Workshop on Quality of Service*, 1999.

[19] I. T. Foster, T. Freeman, K. Keahey, D. Scheftner, B. Sotomayor, and X. Zhang. Virtual clusters for grid communities. In *CCGRID*, pages 513–520. IEEE Computer Society, 2006.

[20] T. Freeman and K. Keahey. Flying low: Simple leases with workspace pilot. In E. Luque, T. Margalef, and D. Benitez, editors, *Euro-Par*, volume 5168 of *Lecture Notes in Computer Science*, pages 499–509. Springer, 2008.

[21] J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke. Condor-G: A computation management agent for multi-institutional grids. *Cluster Computing*, 5(3):237–246, 2002.

[22] L. Grit, D. Irwin, V. Marupadi, P. Shivam, A. Yumerefendi, J. Chase, and J. Albrecht. Harnessing virtual machine resource control for job management. In *Proceedings of the First Workshop on System-level Virtualization for High Performance Computing (HPCVirt)*, 2007.

[23] L. E. Grit. *Extensible resource management for networked virtual computing*. PhD thesis, Durham, NC, USA, 2007. Adviser-Jeffrey S. Chase.

[24] L. E. Grit and J. S. Chase. Weighted fair sharing for dynamic virtual clusters. In *SIGMETRICS '08: Proceedings of the 2008 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 461–462, New York, NY, USA, 2008. ACM.

[25] P. H. Hargrove and J. C. Duell. Berkeley lab checkpoint/restart (blcr) for linux clusters. *Journal of Physics: Conference Series*, 46:494–499, 2006.

[26] F. Heine, M. Hovestadt, O. Kao, and A. Streit. On the impact of reservations from the grid on planning-based resource management. In *Proceedings of the 5th International Conference on Computational Science (ICCS 2005), volume 3516 of Lecture Notes in Computer Science (LNCS*, pages 155–162. Springer, 2005.

[27] M. Hovestadt, O. Kao, A. Keller, and A. Streit. Scheduling in hpc resource management systems: Queuing vs. planning. In D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn, editors, *JSSPP*, volume 2862 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2003.

[28] C. Hu, J. Huai, and T. Wo. Flexible resource reservation using slack time for service grid. In *ICPADS '06: Proceedings of the 12th International Conference on Parallel and Distributed Systems*, pages 327–334, Washington, DC, USA, 2006. IEEE Computer Society.

[29] I.Raicu, Y.Zhao, C.Dumitrescu, I.Foster, and M.Wilde. Falkon: a fast and light-weight task execution framework. In *IEEE/ACM International Conference for High Performance Computing, Networking, Storage, and Analysis (SC07)*, 2007.

[30] D. Irwin, J. Chase, L. Grit, A. Yumerefendi, D. Becker, and K. G. Yocum. Sharing networked resources with brokered leases. In *USENIX Technical Conference*, June 2006.

[31] D. B. Jackson, Q. Snell, and M. J. Clement. Core algorithms of the maui scheduler. In *JSSPP '01: Revised Papers from the 7th International Workshop on Job Scheduling Strategies for Parallel Processing*, pages 87–102, London, UK, 2001. Springer-Verlag.

[32] K. Keahey, R. Figueiredo, J. Fortes, T. Freeman, and M. Tsugawa. Science clouds: Early experiences in cloud computing for scientific applications. In *Cloud Computing and Applications 2008 (CCA08)*, 2008.

[33] K. Keahey, I. Foster, T. Freeman, and X. Zhang. Virtual workspaces: Achieving quality of service and quality of life on the grid. *Scientific Programming*, 13(4):265–276, 2005.

[34] K. Keahey and T. Freeman. Contextualization: Providing one-click virtual clusters. In *eScience 2008*, 2008.

[35] N. Kiyanclar, G. A. Koenig, and W. Yurcik. Maestro-VC: A paravirtualized execution environment for secure on-demand cluster computing. In *CCGRID '06: Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06)*, page 28. IEEE Computer Society, 2006.

[36] I. Krsul, A. Ganguly, J. Zhang, J. A. B. Fortes, and R. J. Figueiredo. Vmplants: Providing and managing virtual machine execution environments for grid computing. In *SC '04: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, page 7. IEEE Computer Society, 2004.

[37] D. A. Lifka. The ANL/IBM SP scheduling system. In *IPPS '95: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 295–303, London, UK, 1995. Springer-Verlag.

[38] M. W. Margo, K. Yoshimoto, P. Kovatch, and P. Andrews. Impact of reservations on production job scheduling. In *13th Workshop on Job Scheduling Strategies for Parallel Processing*, 2007.

[39] S. Mehta and A. Neogi. Recon: A tool to recommend dynamic server consolidation in multi-cluster data centers. *Network Operations and Management Symposium, 2008. IEEE*, April 2008.

[40] A. W. Mu'alem and D. G. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. *IEEE Trans. Parallel Distrib. Syst.*, 12(6):529–543, 2001.

[41] H. Nishimura, N. Maruyama, and S. Matsuoka. Virtual clusters on the fly — fast, scalable, and flexible installation. In *CCGRID '07: Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*. IEEE Computer Society, 2007.

[42] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The eucalyptus open-source cloud-computing system. In *Cloud Computing and Applications 2008 (CCA08)*, 2008.

[43] D. C. Nurmi, R. Wolski, and J. Brevik. Varq: virtual advance reservations for queues. In *HPDC '08: Proceedings of the 17th international symposium on High performance distributed computing*, pages 75–86, New York, NY, USA, 2008. ACM.

[44] P.Beckman, S.Nadella, N.Trebon, and I.Beschastnikh. SPRUCE: A system for supporting urgent high-performance computing. *IFIP International Federation for Information Processing, Grid-Based Problem Solving Environments*, 239:295–311, 2007.

[45] L. Ramakrishnan, D. Irwin, L. Grit, A. Yumerefendi, A. Iamnitchi, and J. Chase. Toward a doctrine of containment: grid hosting with adaptive resource control. In *SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, page 101, New York, NY, USA, 2006. ACM.

[46] T. Röblitz, F. Schintke, and A. Reinefeld. Resource reservations with fuzzy requests. *Concurr. Comput. : Pract. Exper.*, 18(13):1681–1703, 2006.

[47] B. Rochwerger, D. Breitgand, E. Levy, A. Galis, K. Nagin, I. Llorente, R. Montero, Y. Wolfsthal, E. Elmroth, J. Caceres, M. Ben-Yehuda, W. Emmerich, and F. Galán. The reservoir model and architecture for open federated cloud computing. *IBM Systems Journal*, October 2008.

[48] P. Ruth, P. McGachey, and D. Xu. VioCluster: Virtualization for dynamic computational domains. *Proceedings of the IEEE International Conference on Cluster Computing (Cluster'05)*, 2005.

[49] P. Ruth, J. Rhee, D. Xu, R. Kennell, and S. Goasguen. Autonomic live adaptation of virtual computational environments in a multi-domain infrastructure. *IEEE International Conference on Autonomic Computing*, 2006.

[50] M. Siddiqui, A. Villazón, and T. Fahringer. Grid capacity planning with negotiation-based advance reservation for optimized qos. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 103, New York, NY, USA, 2006. ACM.

[51] G. Singh, C. Kesselman, and E. Deelman. Performance impact of resource provisioning on workflows. Technical Report 05-850, Department of Computer Science, University of South California, 2005.

[52] G. Singh, C. Kesselman, and E. Deelman. Adaptive pricing for resource reservations in shared environments. In *GRID '07: Proceedings of the 8th IEEE/ACM International Conference on Grid Computing*, pages 74–80, Washington, DC, USA, 2007. IEEE Computer Society.

[53] W. Smith, I. Foster, and V. Taylor. Scheduling with advanced reservations. In *IPDPS '00: Proceedings of the 14th International Symposium on Parallel and Distributed Processing*, page 127. IEEE Computer Society, 2000.

[54] Q. Snell, M. J. Clement, D. B. Jackson, and C. Gregory. The performance impact of advance reservation meta-scheduling. In *IPDPS '00/JSSPP '00: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 137–153, London, UK, 2000. Springer-Verlag.

[55] B. Sotomayor, R. S. Montero, I. M. Llorente, and I. Foster. Virtual infrastructure management in private and hybrid clouds. *IEEE Internet Computing*, 13:14–22, 2009.

[56] E. Walker, J. Gardner, V. Litvin, and E. Turner. Creating personal adaptive clusters for managing scientific tasks in a distributed computing environment. In *Challenges of Large Applications in Distributed Environments*, 2006.

[57] J. P. Walters, B. Bantwal, and V. Chaudhary. Enabling interactive jobs in virtualized data centers. In *Cloud Computing and Applications 2008 (CCA08)*, 2008.

[58] S. Yamasaki, N. Maruyama, and S. Matsuoka. Model-based resource selection for efficient virtual cluster deployment. In *VTDC '07: Proceedings of the 2nd International Workshop on Virtualization Technology in Distributed Computing*, 2007.

[59] H. Zhao and R. Sakellariou. Advance reservation policies for workflows. In *12th Workshop on Job Scheduling Strategies for Parallel Processing*, 2006.

[60] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. von Laszewski, V. Nefedova, I. Raicu, T. Stef-Praun, and M. Wilde. Swift: Fast, reliable, loosely coupled parallel computation. *IEEE International Workshop on Scientific Workflows*, 2007.

[61] Final report. teragrid co-scheduling/metascheduling requirements analysis team. http://www.teragridforum.org/mediawiki/images/b/b4/MetaschedRatReport.pdf.

[62] Parallel workloads archive. http://www.cs.huji.ac.il/labs/parallel/workload/.