Performance Evaluation of Cloud Infrastructure using Complex Workloads

Athanasios Antoniou

Delft University of Technology

Performance Evaluation of Cloud Infrastructure using Complex Workloads

Master's Thesis in Computer Science

Parallel and Distributed Systems group
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology

Athanasios Antoniou

20th January 2012

Author
Athanasios Antoniou

Title
Performance Evaluation of Cloud Infrastructure using Complex Workloads

MSc presentation
February 3rd, 2012

Graduation Committee

Prof.dr.ir. D.H.J. Epema	Delft University of Technology
Dr. Ir. Alexandru Iosup	Delft University of Technology
Dr. Ir. Andy Zaidman	Delft University of Technology

Abstract

Infrastructure as a Service (IaaS) is a delivery model of cloud computing, which provides the ability to users to acquire and release resources according to their demand and pay according to their usage. Resources are provisioned from the cloud as Virtual Machines (VMs), many of which can be deployed on a single computing node, realizing a multi-tenancy model. While virtualization and multi-tenancy are two sources of workload-execution overhead that have been studied in the past, we still need a thorough, empirical investigation of the joint impact of these overheads, on workload execution.

Additionally, commercial and private IaaS providers offer mechanisms that facilitate the lease and use of single infrastructure resources, but to execute multi-job workloads IaaS users still need to select adequate provisioning and allocation policies to instantiate resources and map computational jobs to them. Even though some studies on the policies employed in cloud environments already exist, current and potential IaaS users need deeper insight on the achieved performance and incurred cost of the used policies, derived through empirical investigation.

In this work, we address these problems with the use of SkyMark, a performance analysis framework for IaaS clouds. SkyMark has three key features: first, it is designed to analyze IaaS deployments through a sequence of automated tests and the subsequent automated analysis of results. Second, it can analyze the impact of individual provisioning and allocation policies to the performance of the workload execution. Lastly, it is able to generate complex workloads, stressing any of the compute, memory and disk components.

With the use of SkyMark, we first study the overheads that the cloud software stack imposes to the workload execution. Subsequently, we analyze the performance and cost of six provisioning and three allocation policies through experimentation in three IaaS environments, including Amazon EC2.

Preface

I developed a passion for video games during my early teen years. Every year or so, a new, more resource-demanding game would come out, so I had to save-up to buy a new, more capable “gaming rig”. When I first heard about cloud computing, I envisioned a world in which no kid would ever have to suffer again with unnervingly low frame rates or pixelated graphics. Since this has yet to happen, I decided that I offer a helping hand to the distributed systems community. For the sake of the kids!

I would first like to thank my supervisor, Dr. Alexandru Iosup, for his continuous support and encouragement. There were moments throughout this work that I would be full of frustration or disappointment, but he was always there to guide me to the right track and provide the motivation I needed. I would also like to thank prof. Epema for the trust that he has shown to me, for giving me the opportunities that he has, and for being understanding and supportive. Both supervisors have been my mentors over the last year, and I hope I absorbed a bit of their knowledge, work methodology, and wisdom. I would additionally like to express my gratitude to David Villegas, he has been an excellent co-worker and a good friend. I would additionally like to thank Kees Verstoep, Paulo Anita, and Munire van der Kruyk for their immediate help when DAS4 was too much to handle. My thanks also goes to the whole PDS group, for all the help that I have received.

I would also like to thank my friends, Dimitris, Vitto, Momchil, Stanislava and Siem for their help and advice, and for putting up with me during the tough moments. Last but not least, I would like to thank my parents, Savvas and Yiannoulla for their endless support, without it I would have managed nothing.

Athanasios Antoniou

Delft, The Netherlands
20th January 2012

Contents

Chapter 1

Introduction

Current trends in the computing field envision its transformation from a traditional in-house power generation model, into a model that consists of services, provided in a manner similar to utilities such as electricity, gas, and water. A basic characteristic of this provisioning model is that the users consume resources and are billed according to their personal demand. The shift to this new computing paradigm has been accelerated by recent advances [27, 29] in the high-speed, yet low-cost interconnection of off-the-shelf computational and storage resources, which made the construction of massive data-centers possible.

Cloud computing attempts to realize the vision of utility computing, through the provisioning of virtualized hardware, software platforms and software applications as services over the Internet. More specifically, Infrastructure-as-a-Service (IaaS) clouds offer the ability to acquire resources on-demand, usually in the form of virtual machines (VMs), i.e., software implementations of machines with a pre-agreed computing power, memory and disk size, operating system, libraries and applications. Platform-as-a-service (PaaS) clouds offer platform services such as application development, testing and run-time environments. Lastly, Software-as-a-Service (SaaS) clouds deliver specialized software as web-based services.

IaaS clouds such as Amazon Web Services (AWS) [7], GoGrid [2], and ElasticHosts [1], have recently achieved commercial traction. There is, therefore, a need for a deeper understanding of the performance characteristics of real IaaS clouds. The goal of this thesis is to provide insight regarding the performance of such environments.

1.1 Problem Statement

Clouds essentially time-share resources between users. In IaaS environments, virtualized resources can be deployed by multiple users on the same physical machine. The VMs that reside on the same physical host time-share the available physical resources. Early work on the performance of resource time-sharing, which shows performance degradation effects, already exists [9, 10]. Moreover, virtualization

is an additional layer between user and hardware, and consequently it imposes an overhead, which has been studied separately for various virtualized components [11, 17, 28, 56].

Even though several research projects that study the *performance of virtual resources* already exist, we still need a more in-depth understanding of the factors that influence the efficiency of the IaaS paradigm, which has adopted the multi-tenancy and virtualization concepts. An indication that there is still a lot of room for improvement, is that public IaaS providers do not currently offer any performance quality-of-service (QoS) guarantees, despite the desirability of such guarantees [8].

In IaaS clouds, users provision, i.e., acquire and release resources according to their current needs. They can subsequently employ their own allocation scheme, to schedule work on the leased resources, based on the requirements of their workloads. The selected provisioning and allocation policies have been shown to have a considerable impact on the traditional performance metrics [19]. Choosing policies that are incompatible with the workload could lead to wasted resource time and excessive charges [23]. Therefore, IaaS users need also a better understanding of the *performance and incurred cost of the selected resource provisioning and allocation policies*. This problem has been approached with simulations by several studies [19, 23, 46], however, there is a need for empirical evaluation since recent results [38, 40, 45] in cloud performance evaluation show that cloud performance is lower and more variable than considered by simulation models.

The problems at hand give rise to a number of research questions that need to be addressed. The goal of this thesis is to provide an answer to the following two main questions:

- **RQ1**: What are the performance overheads of executing workloads with the IaaS cloud delivery model?
- **RQ2**: What is the impact of the selected allocation and provisioning policies on the performance and cost of IaaS services?

1.2 Approach

Our approach to addressing the research questions posed in Section 1.1 is *SkyMark* and the design of a set of *complex workloads*. In a nutshell, SkyMark is an extensible and portable framework that provides the ability to generate and submit real or synthetic workloads to IaaS cloud systems, collect performance-related results, and subsequently perform analysis on multiple result data-sets. SkyMark can currently facilitate experimentation with several clouds, provisioning, and allocation policies, but it can be easily extended with new clouds and policies.

SkyMark is based on preexisting work of the Parallel and Distributed Systems group (PDS), namely, C-meter [91] and GrenchMark [34]. Grenchmark is a workload generation and submission framework for grid environments. C-Meter was

later developed to port core Grenchmark capabilities to the cloud. In this work, we extend the functionality of the previous frameworks with several features, mainly with the ability to generate our set of complex workloads, experiment with several clouds using different provisioning and allocation policies, and perform analysis on multiple result data-sets using our collection of metrics and visualization techniques.

We chose to address the posed research questions using *measurements* over simulation and analytical modeling, because the other methods require simplifying and, currently, unverifiable assumptions to be made. An IaaS cloud is a form of a distributed system with a complex software stack. Thus, many system and environment parameters need to be taken into consideration. This makes simplifying assumptions, and consequently, simulation and modeling less credible. However, the environment parameters need to be identified even in the case of empirical studies, otherwise the acquired results will be incomparable.

To identify the performance characteristics of a system, we would ideally make use of realistic workloads that represent the activity observed on that specific system or other systems that belong in the same class. In the case of IaaS clouds, it is not yet possible to define realistic workloads, due to the insufficient amount of public workload traces and common practice reports for IaaS environments. To address this matter, we form several complex, synthetic workloads, which take into consideration the trends in parallel and grid computing workloads [37], a well studied area.

Our complex workloads comprise large amounts of micro-benchmarks of different durations, which stress one or more components of the examined IaaS cloud. They further exhibit several arrival patterns. Unlike using micro-benchmarks or applications individually, the use of complex workloads will allow us to observe cloud performance variations caused by interactions between unrelated jobs, executing on the same or different virtual machines.

Our approach for RQ1 requires performing experimentation on both virtualized and non-virtualized resources. Since we want to examine the overheads imposed by the cloud software stack, performing the black-box evaluation that is characteristic to public, commercial IaaS clouds is not adequate. For this reason, the experiments designed to address RQ1 are performed on a private IaaS cloud that we can fully control.

We approach RQ2 by firstly identifying a set of provisioning and allocation policies. Additionally, we create a performance-optimization heuristic that is applicable to the set of provisioning policies. We evaluate the impact of the policies using only a subset of the formed complex workloads, and additionally by varying either the allocation or the provisioning policy, in two separate sets of experiments. The interactions between allocation and provisioning policies is not within the scope of this work, but we have conducted elsewhere a preliminary study of this effect [81].

1.3 Thesis overview

Here, we present an overview of this thesis: first, an overview of the thesis contributions, and second, the outline of the thesis structure.

1.3.1 Thesis Contributions

We identify the following contributions of this thesis:

1. We *design SkyMark*, a framework for performance analysis of IaaS systems (Chapter 4).
2. We *identify a set of provisioning and allocation policies* for IaaS systems, and a *heuristic* that is applicable to the set of provisioning policies (Chapter 4).
3. We *assess the performance of an IaaS cloud*, by conducting an empirical study with the use of SkyMark and a set of complex workloads (Chapter 6).
4. We *evaluate empirically the impact of the selected provisioning and allocation policies* on the IaaS *performance and cost*, using three IaaS clouds (Chapter 6).

1.3.2 Thesis Structure

Firstly, we describe the basic concepts of system performance analysis and cloud computing in Chapter 2. Chapter 3 presents related work in the field of cloud systems performance evaluation. Chapter 4 describes SkyMark, our approach to answering the posed research questions. In Chapter 5, we formalize the experimental setup. We then present the results collected from the experimentation in Chapter 6. Finally, Chapter 7 summarizes the work performed for this thesis, presents our conclusions, and proposes a direction for future work.

Chapter 2

Background

This chapter introduces the basic concepts that relate to performance evaluation and cloud computing. Section 2.1 presents some terminology used in the disciplined systems performance evaluation. Virtualization, an important concept for cloud computing, is discussed in Section 2.2. Lastly, a taxonomy of cloud computing services and some real-world cloud computing implementations are introduced in Section 2.3.

2.1 Systems Performance Evaluation

In this section, we present the techniques used for systems performance evaluation and some essential terminology used in this field. Performance evaluation of computer systems is a well-defined, structured process, with the goal of understanding and comparing the performance characteristics of the systems that participate in the analysis [44]. The necessity of performance analysis stems from the need to find the solution that best meets non-functional requirements, such as performance, reliability, and cost.

2.1.1 Techniques

Three different techniques can be used for performance evaluation of systems [44], namely, *measurement* over real systems, *simulation* and *analytical modelling*. The appropriateness of these techniques depends on applicability, available resources, required accuracy and the amount of time that can be spent on the evaluation.

Measurement results are more widely acknowledged and convincing than modeling or simulation results, since no simplifications are applied to the evaluation procedure. Nevertheless, the accuracy of the results can vary greatly, since a lot of parameters must be recognized and controlled, otherwise environment changes will interfere with the results. Moreover, measurement is not always possible, since the evaluated system might not already exist. It also requires equipment that might be expensive and the process of measuring may take a considerable amount of time.

Simulation is less costly but can be just as time-consuming as measurement and much more error-prone. The accuracy depends on the simplifications and assumptions that are used. Simulation results are not acknowledged as much as measurement results.

Analytical modeling is usually the fastest technique to use, and may provide the best insight regarding the effects of the experiment parameters and their interactions. On the other hand, analytical modeling requires sufficient modeling skills and a lot of simplifications and assumptions that reduce the accuracy of the model. Most importantly, modeling results are acknowledged less than the other two techniques, so confirmation through simulation or measurement usually follows [44].

2.1.2 Terminology

Workload

The requests made by the users of a system and are processed by the system form a workload. A *test workload* refers to any workload used in performance analysis studies.

Test workloads can be *real* or *synthetic*. A workload that is observed during the normal operation of a system is a real workload. Real workloads can be used in evaluation studies using *traces*, i.e., representations of workloads that describe the requested and utilized resources, the timestamps of all major scheduling events (e.g., submission time), the used job credentials, and the job execution environment. Synthetic workloads are used in studies and are generated so as to represent real workloads. A workload model is derived from a real workload, by observing its key characteristics, a process called *workload characterization*. The workload model can then be used to generate synthetic workloads.

Benchmark

Benchmarking refers to the process of performance comparison of several systems by applying the measurement technique. The workloads used in these measurements are called *benchmarks*, and a grouping of related benchmarks is called a *benchmark suite*. *Micro-benchmarks* are small programs that exercise one specific component of a system [50].

Many benchmarking suites for testing various systems are made available for experimenters, an example of which is the Standard Performance Evaluation Corporation (SPEC) benchmarks [76]. SPEC is a non-profit organization that develops a standardized set of performance benchmark suites, such as SPEC CPU2006 (CPU,memory and compiler benchmark), SPECweb2009 (web server benchmark) and SPECvirt_sc2010 (performance evaluation of datacenter servers used in virtualized server consolidation).

System Under Test (SUT)

The system that is evaluated is often referred to as System Under Test (SUT). Occasionally, the evaluation examines the impact that alternative solutions for a specific component have, on the performance of the SUT. The component is named Component Under Study (CUS).

In our case, the SUT and CUS coincide for the IaaS evaluation part (an IaaS cloud). For the policy evaluation part, the SUT is still an IaaS cloud, but the CUS are the used provisioning and allocation policies.

Response variable

The response variable represents the outcome of an experiment, which typically is the measured system performance.

Parameter

The experiment parameters refers to the system, environment and workload characteristics that affect the SUT performance. All the parameters should be determined during the design of the experiments, otherwise unidentified parameters could make the results useless.

Factor

A *factor* is a parameter whose adjustment influences the response variable, and has several alternatives. The values that a factor can assume are called its *levels*. The *primary factors* of a performance evaluation study, are the factors whose impact needs to be appraised. Likewise, factors whose impact is not studied are called *secondary factors*.

Performance Metric

A *performance metric* is a value that describes the performance of the SUT [50]. This value is derived from the values that were measured during the SUT perfomance analysis. Lilja [50] identifies several characteristics that form a 'good' performance metric:

1. **Linearity**: A change in the value of the metric should designate a proportional change in the actual performance of the system.

2. **Reliability**: When a system A scores better than a system B with respect to a reliable metric, then we can deduce that system A outperforms system B.

3. **Repeatability**: A repeatable metric provides the same value each time the same experiment takes place.

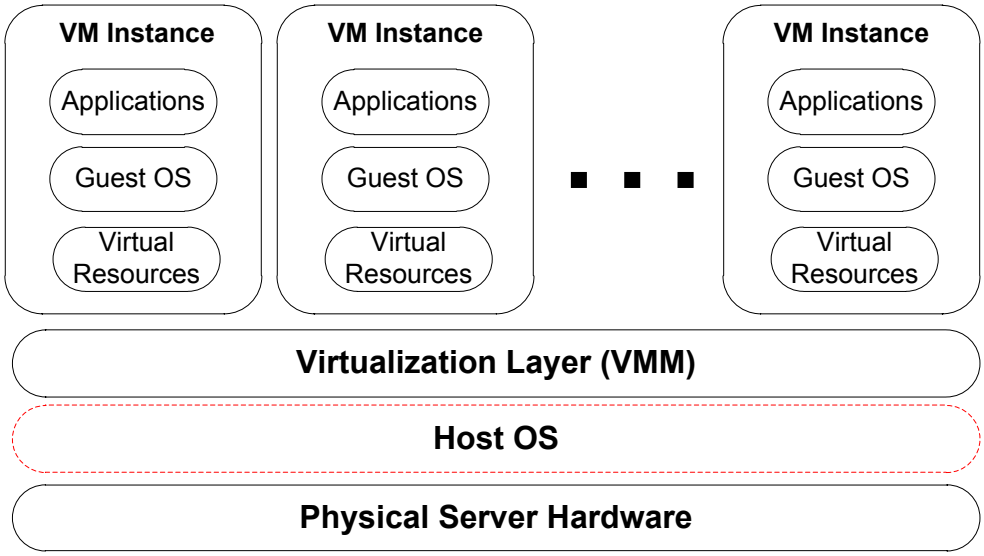

Figure 2.1: Multiple Virtual Machines hosted on a single server. The presented virtualization technologies, require a Host OS layer, except from the bare-metal virtualization technique.

4. **Easy to measure**: Metrics that are difficult to measure are frequently measured incorrectly.

5. **Consistency**: The definition of the metric is independent from the system it is measured on.

Application Profiling

The execution performance characteristics of an application are provided by its *profile*. An application profile might specify, for example, the amount of time spent in each of the application phases or states [50]. The profile can be used to identify the most time consuming parts of an application.

2.2 Hardware Virtualization

Hardware virtualization enables a single physical platform to run multiple operating systems and software stacks [84]. Virtualization creates an abstraction layer between user and physical resource, but at the same time it provides the user the illusion of direct interaction with the physical resource [47].

The virtulization model is depicted in Figure 2.1. The Virtual Machine Monitor (VMM), also known as the hypervisor, establishes the abstraction layer that encapsulates and isolates each Virtual Machine (VM). The VMM runs on the actual ma-

chine and maps physical resources (processing power, memory, storage, network) to VMs. The Operating System (OS) running inside a VM can, therefore, only make use of the virtual resources mapped to the confining VM. Consequently, the physical resources of a machine can be partitioned between multiple VMs, without the VMs being aware of the multi-tenancy.

The virtualization paradigm is a good fit to cloud computing, since it can improve resource utilization [21]. Multiple cloud users can share the resources of a single host, since their leased virtual resources are multiplexed on the same physical machine. Additionally, virtualization provides isolation, because each cloud user receives a separate, confined environment to run his workloads in.

2.2.1 Virtualization techniques

A number of different hardware virtualization techniques are used by the available VMMs:

- **Full virtualization**: The VMM simulates the functionality of the physical hardware, allowing for an unmodified OS to be hosted [74]. The guest OS must support the same instruction set as the hardware.

- **Paravirtualization**: The VMM provides a special API to the VMs, which consequently requires guest OS modification, i.e., OS distributions that are not paravirtualization-aware cannot run on a paravirtualizing VMM. As a result of the API use, the execution of critical instructions that are more difficult to run in a virtual environment, takes place on the host OS instead of on the guest OS [11, 88].

- **Hardware-assisted virtualization**: An extension of the previous two virtualization techniques. It is also referred as Hardware Virtual Machine (HVM). The VMM receives support from the hardware (primarily the host processors), which enables it to efficiently deploy fully-virtualized or paravirtualized machines. With the hardware-assisted virtualization technique, certain operations are performed by the hardware instead of the VMM software, which minimizes the imposed overhead [80].

- **Bare-metal virtualization**: In this case, there is no host OS [82]. Instead, the VMM is installed directly on the physical server. The elimination of the host OS layer from the virtualization stack results in a performance improvement over the hosted VMM techniques.

2.2.2 Virtual Machine Managers (VMMs)

Several Virtual Machine Managers (VMMs) have been developed, and are currently used in cloud computing environments. Here, we give a short description for the most prominent ones, namely KVM, Xen and VMWare.

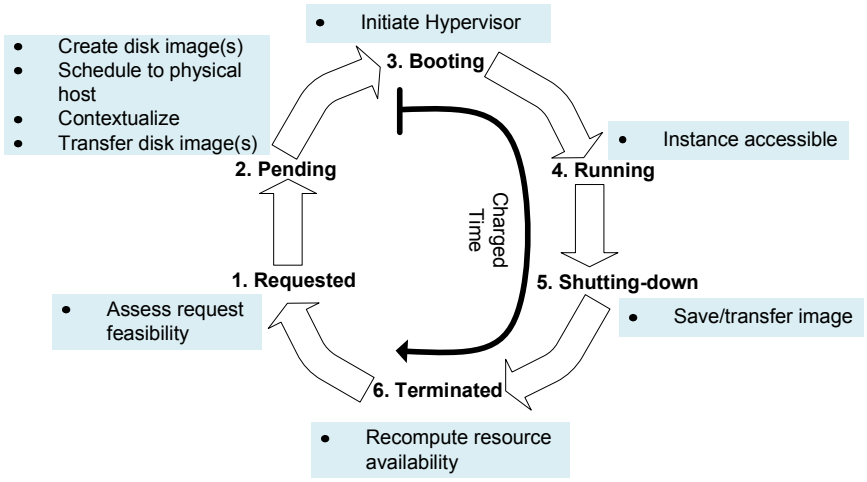

Figure 2.2: The life-cycle of a VM.

The *Kernel-based Virtual Machine (KVM)* [48] is part of the Linux kernel, and implements the hardware-assisted full-virtualization technique. KVM makes use of modules embedded in the Linux kernel for operations such as memory management and scheduling. The user-space counterpart is therefore small and simple [84].

Xen [11] implements paravirtualization as well as hardware-assisted virtualization. It is licenced under the GNU General Public License (GPLv2). However, it is used by other commercial VMMs as their core, such as Citrix XenServer [33].

VMware [83] is a company with several virtualization solutions and a pioneer in the field. It offers both hosted and bare-metal VMMs. VMWare Workstation and Fusion are examples of hosted VMMs (paravirtualized and full) while VMware ESX and ESXi are bare-metal hypervisors.

2.2.3 The VM life-cycle

Figure 2.2 shows the stages that a VM goes through, when it is deployed in a cloud environment. Initially, the VM is requested (State 1 in Figure 2.2) from the cloud and is then placed in a *pending* state (State 2). In the start of the pending state, the VM awaits until it is scheduled to a physical resource. The scheduling decision relies on the cloud environment.

After the VM has been assigned to a host, the cloud manager allocates the root disk image that includes the operating system of the VM. Additional disk images can be requested for a VM instance. The cloud subsequently transfers the VM images to the selected host. A process called *contextualization* might take place before or after the transfer of the disk images. During contextualization, the disk images are modified so that they work in a specific environment, e.g., the VM

host-name and network are set up [51].

As soon as all the required files are located on the selected host, the hypervisor boots the VM, i.e., the VM enters the *booting* state (State 3). When the operating system is up, the VM proceeds to the *running* state (State 4). A shutdown request forces the VM to progress to a corresponding *shutting-down* state (State 5). Depending on the cloud policy and user preference, the disk images might, or might not be saved for future use. The saved data could be stored locally, or transferred to a storage server. Lastly, the VM reaches the *terminated* state (State 6).

According to the Amazon EC2 billing scheme [4], the charged VM time begins when the VM enters the booting state, and stops when it enters the terminated state.

2.3 Cloud Computing

The "cloud computing" term encapsulates several layers of computing provisioning. It includes the hardware resources located at the data-centers of cloud providers, the operating system software on top of that hardware, and lastly the applications that are delivered as services over the Internet. Additionally, cloud computing provides these services as a utility, with the customers being billed based on usage, similar to the billing scheme of traditional public services such as water, electricity and telephony.

However, there is no absolute consensus on the meaning of the "cloud computing" term. NIST[1] describes cloud computing [55] as a *"A pay-per-use model for enabling available convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.*"

Driven by the economies of scale, cloud computing enables the use of inexpensive resources. Cloud providers purchase hardware in large quantities, which is significantly more economic [26]. They can then amortize the cost of owning and operating a large capacity infrastructure by time-multiplexing their resources between many clients [8]. Moreover, large scale systems require deep automation, which results in cost reductions due to need for smaller operational staff. In a well-run enterprise, a typical ratio of administrators to servers is 1:100, while in a cloud data-center the ratio is at least 1:1000 [26]. Operating at this scale allows cloud providers to offer services to clients with a lower cost than what an in-house computing facility would achieve.

2.3.1 Cloud Features

Although cloud computing incorporates several models of computing provisioning, several primary features can be identified across the complete domain [55, 84]:

[1] National Institute of Standards and Technology.

- **Scalability/Elasticity**: Clouds provide the illusion of infinite capacity. Users are able to quickly request, acquire and later release resources from the cloud on-demand. When the load of a hosted service increases, the cloud should be able to offer as many computing power as necessary. Similarly, these resources can be released back to the cloud upon load decrease.

- **Pay-per-use metering and billing**: In clouds, the user resource consumption is monitored, and categorized based on the resource type. Users are billed according to their measured consumption. Therefore, users do not need to make any up-front commitments that require the prediction of their application's resource requirements beforehand.

- **Self-service based usage**: The user is able to administer the offered computing capabilities of the cloud services, such as requesting or releasing network storage and paying for used resources, without interacting with human operators from the provider.

- **Quality of Service (QoS)**: Cloud service providers offer QoS guarantees with Service Level Agreements (SLAs), which are legally binding contracts. Usually, cloud providers may make guarantees in terms of system uptime, but do not provide any guarantees regarding the application level performance [60].

- **Resource pooling**: Multiple customers are served in a multi-tenant model, i.e., software and hardware resources are shared between users. Computing power is offered from a pooled set of resources, which might reside in multiple data-centers. The exact location of the resources is abstracted from the users. However, the user might be able to choose the location at a higher abstraction level, e.g select the preferred country or data-center.

2.3.2 Service Models

Figure 2.3 presents the cloud delivery models and their provided services. Clouds provide services at three different levels of abstraction: At the highest abstraction level, *Software as a Service (SaaS)* delivers specialized software to the consumers over the Internet. SaaS typically involves a usage-based pricing scheme, in which the cost increases in relation to the number of users and the used application features. Salesforce [73] SaaS cloud delivers Customer Relationship Management (CRM) software services.

Platform as a Service (PaaS) is located at a lower abstraction level. PaaS pertains to the provisioning of an integrated environment which can be used for the development, testing and deployment of applications. The PaaS users are not occupied with deploying and managing the underlying hardware and software. An example of a PaaS provider is Google AppEngine [25], which provides Java and Python run-time environments with automatic load-based scaling.

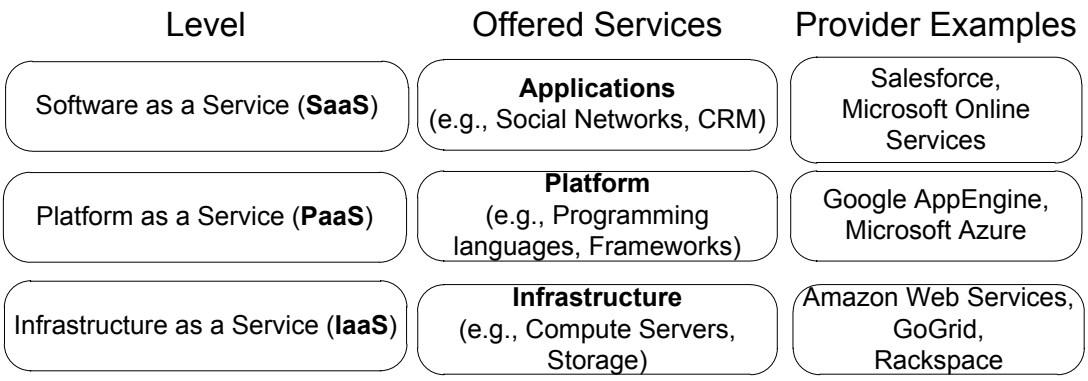

Figure 2.3: The Cloud Computing Stack.

At the other end of the abstraction level spectrum is *Infrastructure as a Service (IaaS)*. IaaS refers to the on-demand provisioning of virtualized resources, i.e., computation, storage and networking. Users can lease VM instances, which encapsulate a provider-specified amount of resources (providers typically offer several types of VM instances), and can run a user-specified operating system enriched with required applications and libraries. Clients fully configure and control their instances as root via ssh. Amazon Web Services [7] is a typical IaaS cloud, whose services are outlined later on.

2.3.3 Deployment models

A secondary classification scheme for clouds distinguishes cloud deployments based on how and to whom they are distributed. Using this scheme, clouds can be classified as *public*, *private*, *community*, and *hybrid* [84].

Public clouds are made available to the general public by third-party providers. The clients are billed for the offered service in a pay-as-you-go manner. In contrast, private clouds are deployed within a business or an organization, on top of the organization's data center. Private clouds are purposed for internal use and the users might or might not be charged for their resource consumption.

Cloud deployments that are shared by several organizations with the goal of supporting a specific community are called community clouds. Lastly, hybrid clouds provide the ability to combine resources from public and private clouds. Resources from public clouds are usually leased when the private cloud capacity is not sufficient to handle the current load.

2.3.4 A typical IaaS: The Amazon Web Services (AWS)

Amazon Web Services (AWS) [7], offered by Amazon.com is one of the most prominent cloud providers, and the first service to employ the IaaS model, in 2006.

Instance Type	Category	Capacity (ECUs)	Cost (US$/hour)
t1.micro	Micro	Up to 2 ECUs for short bursts (1 virtual core)	0.025
m1.small	Standard	1 (1 virtual core with 1 ECU)	0.095
m1.large	Standard	4 (2 virtual cores with 2 ECU)	0.38
m1.xlarge	Standard	8 (4 virtual cores with 2 ECU)	0.76
c1.medium	High-CPU	5 (2 virtual cores with 2.5 ECU)	0.19
c1.xlarge	High-CPU	20 (8 virtual cores with 2.5 ECU)	0.76
m2.xlarge	High-Memory	6.5 (2 virtual cores with 2.75 ECU)	0.57

Table 2.1: Description of the basic EC2 instance types. t1.micro is the latest VM instance type introduced by Amazon, in June 2011.

There are currently seven data-centers distributed over the world, that provide computing capacity to AWS users. AWS offers several cloud services, the most important of which are the Elastic Compute Cloud (EC2) and the Simple Storage Service (S3).

With EC2, computing capacity is provided in the form of virtual machine instances that are based on the XEN hypervisor. EC2 offers a fixed set of instance types, each with different capabilities, operating systems, architectures and price. The CPU capacity of the offered instance types is characterized in Elastic Compute Units (ECUs). One ECU unit is approximately equivalent to a 1.2 GHz 2007 Opteron or 2007 Xeon processor [3]. VM instance usage is charged on an hourly-basis. Some of the instance types, their computing capacity and their hourly cost are presented in Table 2.1. The Amazon Machine Image (AMI) format is used in EC2, which allows the users to deploy customized operating systems containing software and libraries that are fit for purpose.

Amazon S3 offers storage capabilities to AWS users. Data is organized into buckets, with each bucket being able to store an unlimited amount of data. Each file can be up to 5GB in size. Users can create, modify and read objects in buckets.

AWS additionally offers services such as CloudWatch (resource and application monitoring), Simple DB (SDB, structured datastore), Relational Database Service (RDS) and Elastic Block Store (EBS, persistent disk service). Particularly interesting for this work, are the Elastic Load Balancing and Auto Scaling services. The Elastic Load Balancing service is responsible for scheduling incoming application traffic across multiple EC2 instances. Auto Scaling allows the number of instances to automatically scale up or down, according to a set of customizable rules.

2.3.5 Virtual Infrastructure Manager (VIM)

A cloud is in need of software that can manage physical and virtual resources and present a complete view of the current status of the cloud. It should be also able to supervise the full life cycle of the VMs that are deployed on top of the

physical resources. The software that is used for these purposes is called a Virtual Infrastructure Manager (VIM) [75]. In contrast to the VMM or hypervisor (see Section 2.2), which is responsible for the virtual machines deployed on a single node, a VIM controls the state of the whole cloud. A VIM therefore cooperates with one or more VMM types, installed in all the nodes that form the IaaS cloud.

A differentiation between "cloud toolkits" and VIMs is proposed by Sotomayor et al. [75]. Their argument is that solutions that belong in the toolkit category should expose a remote "cloud-like" interface for creating, controlling and monitoring virtual resources, and should employ user administration and a user permission management mechanism. On the other hand, VIMs should provide advanced features such as automatic load balancing, server consolidation, and dynamic infrastructure resizing and partitioning. Since the current IaaS software exhibits significant overlap between these two categories, we address both as VIMs.

OpenNebula

The OpenNebula [63] VIM begun as a project in the Complutense University of Madrid in 2005, but evolved into an open-source project with the first release taking place in 2008.

The OpenNebula platform is modular and is composed by three main components, namely the *core*, the *scheduler* and the *drivers*. The core is responsible for coordinating the physical servers and the hypervisors running on top of them, providing virtual networks for the VMs, and preparing disk images for VMs. The driver component provides an extensible set of drivers to the core, that are used to perform specific network, storage or virtualization operations. The drivers interact with APIs of hypervisors, storage and network technologies, and public clouds [84]. Lastly, the scheduler makes decisions on the placement of virtual resources to physical resources, based on information on the current state of the cloud. It implements VM placement policies for workload balancing, with a simple matchmaking policy set as default.

OpenNebula can be used through a Command Line Interface (CLI) or a web-based GUI (Sunstone). Additionally, OpenNebula exposes its functionality through several APIs, namely, the Open Cloud Computing Interface (OCCI) [62] of Open Grid Forum and a subset of the Amazon's EC2 Query interface. Additionally, it provides an XML-RPC [64], a Java, and a Ruby interface.

With OpenNebula, two different disk image allocation methods can be used, namely, *eager* and *lazy* allocation. With eager allocation, the VM image is allocated on the physical disk when the VM is provisioned. Eager allocation is implemented with the raw image format, which is a plain binary image. With lazy allocation, the physical areas are allocated as needed, when the guest operating system tries to write data. Lazy allocation is implemented with the Qemu Copy on write 2 (qcow2) optimization strategy [89].

OpenNebula also supports two image transfer methods. The *Network File Sys-*

tem (NFS) method uses the underlying shared file system to implicitly transfer the images. The *LocalDisk* method performs an explicit copy to the local disk of the node that is hosting a VM, using the Secure Copy Protocol (SCP). A variant of localDisk with *caching* is also supported, where images are cached to the host nodes and can be reused, thus not requiring a network transfer.

Through its virtualization subsystem, OpenNebula can interact with the VMMs on the host nodes. The current version of OpenNebula can be used with the KVM, Xen and some VMware hypervisors.

Eucalyptus

Similarly to OpenNebula, Eucalyptus is an open-source product coming from the academia. There is also an enterprise version of Eucalyptus that offers some additional features.

One of the identifying features of Eucalyptus is that it is built with the concern of providing similar functionality as Amazon Web Services, through the same APIs. Thus, Eucalyptus implements Amazons EC2, S3 and EBS interfaces for provisioning compute, storage and block-level storage services.

Eucalyptus has a modular and hierarchical design. It is comprised by five high-level components. The *Cloud Controller (CLC)* is the user-visible entry point and global decision-making component of the installation [61]. It is responsible for coordinating the provisioning of virtual resources to the users and monitoring the system's components and virtual resources. The *Cluster Controller (CC)* manages a collection of servers that actually provide the virtual resources. The *Node Controller (NC)* module is deployed on each one of these servers. NC is responsible for executing, inspecting, terminating and cleaning-up VM instances on the host machine where it is installed. *Walrus* is the data storage service of Eucalyptus, and is interface-compatible with Amazon's S3. Lastly, the *Block Storage Service* provides block level storage volumes that can be used by the VM instances.

The currently supported hypervisors for Eucalyptus are XEN and KVM. The enterprise version of Eucalyptus also supports VMware hypervisors. In the current version, Eucalyptus only supports eager image allocation and localDisk image transfer with caching on the host nodes.

Chapter 3

Related work

This chapter presents the related work. Since the work is divided into two research components, namely, the study of the performance characteristics of IaaS environments and the evaluation of several provisioning and allocation policies, we provide one related work section for each component. Section 3.1 presents work studying the performance characteristics of IaaS clouds. Correspondingly, Section 3.2 discusses previous work on provisioning and allocation on IaaS environments.

3.1 Cloud performance analysis

There is already extensive research on the performance analysis of clouds and virtualized systems. The predecessor of this work, C-Meter [91], is used to examine the overhead of acquiring and releasing resources from and to the Amazon EC2 cloud. This work is the continuation of the work performed by [91]. Here, we extend the C-Meter framework to study the performance of several clouds, in addition to Amazon's EC2 compute cloud, to use varying workloads, and to support experiments with provisioning and allocation policies.

Many more *performance studies* have taken place on Amazon Web Services (AWS). Ostermann et al. [65] analyze the performance of EC2 using micro-benchmarks and kernels and conclude that the performance of virtualized resources acquired from public clouds have a much lower performance when compared to the theoretical performance peak, especially for computation and network intensive applications. In contrast, we compare the observed virtualized performance of a private cloud with the non-virtualized performance of the underlying hardware. Additionally, we make use of complex workloads instead of micro-benchmarks or applications used as individual benchmarks, which can capture the interactions between seemingly unrelated jobs submitted by different cloud users.

Palankar et al. [67] examine the performance of Amazon S3, including an evaluation of file transfers between EC2 and S3. Amazon EC2 is also studied using the NPB benchmark suite [86], designed to evaluate the performance of HPC systems, and with an x-ray spectroscopy and electronic structure application [70]. Deel-

man et al. [20] study the performance and cost of executing Montage, a scientific workflow, on EC2. Jackson et al. [42] port the SNfactory pipeline, an astronomy application comprised of pipelined serial processes, to AWS. The execution of several HPC applications on EC2 is compared to the execution on supercomputers and clusters [43].

Iosup et al. [40], study the performance variability of production cloud services, using year-long traces of Amazon AWS and Google AppEngine. These cloud services exhibit periods of stable performance as well as yearly and daily patterns. The performance stability and homogeneity of small Amazon EC2 instances is studied by [45]. Li et al. [49] carry out a performance and cost comparison between four major public clouds. The clouds are compared on the common functionality set, which includes elastic computing, persistent storage, intra-cloud network and Wide-area network.

Ueda et al. [79] study the performance of OpenNebula and Eucalyptus with the use of a workload based on Wikipedia software and data. They examine the impact of Lazy/Eager image allocation methods, and the NFS/localDisk image transfer methods. Their main finding is that these two configurations have a great impact on the performance of the clouds, regarding provisioning and processing times. They also address a multi-tenancy scenario that shows a significant impact of provisioning on the achieved throughput. An early comparative study of Eucalyptus with Amazon EC2 is presented in [61].

Performance studies on *virtualization* cost have shown that the virtualization performance overhead for the XEN hypervisor is at most 5% for computation [11, 17] and 17% for networking [11, 56], with the use of general purpose benchmarks. Grund et al. [28] find that the virtualization cost for memory is around 7%, but can increase up to 60% when running multiple VMs at the same time and with the use of CPU architectures that have only one global memory controller. Yu et al. [93] find that the virtualization cost for parallel I/O is below 30%. Two other studies [16, 57], examine the virtualization cost for web server I/O, showing that it is below 10%. Paravirtualization for compute-intensive HPC kernels is found to pose no statistically significant overhead [92]. Weng et al. [87] research the performance degradation caused by virtualization to parallel program execution. Our work extends on these previous findings, by examining the overheads posed by the IaaS software stack, using several workloads and workload arrival patterns.

Regarding virtualization *performance isolation*, Barham et al. report a small interference between co-located VMs, when testing the Xen hypervisor with web server workloads. Nathuji [60] et al. develop Q-Clouds, a framework that tunes resource allocations to deal with the performance degradation caused by VM interference. Pu et al. [68] study the performance interference among VMs running network I/O workloads.

Work has also been performed in developing *cloud benchmarks* and *benchmarking methodologies*. Binnig et al. [13] argue that traditional benchmarks are not adequate for cloud performance analysis, because they don't take the main cloud

characteristics (see Subsection 2.3.1) into consideration.

There is also related work performed in the industry. CloudHarmony [18] is a startup that has developed a set of benchmarks in order to provide performance comparisons between public clouds. Their Cloud SpeedTest service benchmarks the network latency and throughput between a public cloud and the consumer. They additionally perform cloud-to-cloud network benchmarking. Lastly, they measure the service availability for a great number of public clouds, and keep historical up-time data which are accessible through their website.

3.2 Provisioning and Allocation Policies

The study of policies for dynamically provisioned computing environments is a new research field, but a lot of effort has been put recently. A substantial part of the related work examines policy alternatives under the assumption of extending the capacity of a private cluster with on-demand resources drawn from a cloud environment [12, 15, 19, 46, 54, 66, 72]. In contrast, this work is the first empirical comprehensive investigation of provisioning and allocation policies conducted using *real* IaaS cloud environments.

Closest to this work, Genaud et al. [23] perform *simulations* in order to evaluate several provisioning policies, using workloads constructed from real production grid traces. Their policies assume that the job durations are known, which is the case for only one of our proposed provisioning policies. Furthermore, their model neglects VM boot-up and shut-down times, which makes it unclear whether their simulation results can be regarded as realistic. Lastly, the evaluation considers the job wait time and the cost according to Amazon's billing scheme, but does not use workload-oriented metrics such as workload makespan and speed-up.

Mao et al. [53] present an automatic cloud provisioning algorithm, which considers VM startup times and can also select among multiple instance types. The jobs have known deadlines and the number of jobs that finished within the deadline is used to measure the mechanism's performance. The provisioning policy is evaluated using simulations and a real scientific application on Microsoft Azure.

Assuncao et al. [19] consider the same scenario, i.e., extending a privately-owned cluster with virtual resources from an IaaS provider. They evaluate by simulation with real traces, three allocation policies and several redirection strategies, that specify which job requests are redirected to the cloud. Marshall et al. [54] propose a model to elastically extend a static-resource site by integrating remote cloud resources on-demand. They additionally propose three provisioning policies that take decisions based on the job queue status, but their policy comparison and evaluation is not in-depth, since the focus is on the proposed elastic framework. In yet another piece of work that considers extending the resources of a local cluster with resources from a cloud, Kijsipongse et al. [46] examine two provisioning policies. The policies are assumed to know the job resource requirements and try to provision the proper types of instances for the jobs in the queue. The evaluation

however is substantially small-scale and cannot be regarded as conclusive.

In [58], the authors address the provisioning problem on a virtual cluster level, each one leased by a different Virtual Organization (VO). The implementation is based on the Condor scheduler [77]. Lu et al. [52], identify the problem of load imbalance while executing DNA profiling workloads on Microsoft's Azure cloud. Candeia et al. [15], propose a greedy allocation policy to manage bursts of bags-of-tasks on a hybrid infrastructure (use of local and cloud resources). Ostermann et al. [66] extend a Grid workflow application environment to harness resources from IaaS clouds, when necessary, with a simple provisioning policy. Deelman et al. [20] study through simulation the performance-cost trade-off of various static provisioning plans, for a real-life astronomy application. Ben-Yehuda et al. [12] develop ExPERT, a Bag-of-Tasks (BoTs) scheduling framework that selects the pareto-efficient strategies, i.e., the strategies that deliver the best results for makespan and cost, for running BoTs, with replicated tasks, on a mixture of environments with varying reliability, cost, and speed. Amini et al. [72] propose two market-oriented scheduling policies that consider resource cost, user budget and application deadlines by supplementing local resources with resources from an IaaS provider.

The approach in [14, 30, 69, 78] considers the estimation of application service times and *workload pattern prediction*. Quiroz et al. [69] address the VM provisioning problem from the provider's perspective, in order to improve server utilization. They propose an online clustering model, to detect patterns in the stream of requests. Incoming jobs are analyzed using a model, to provide application service-time estimations. These estimations are used to form a set of VM classes, that describe the VM resource configuration and the required instance quantity.

Cost is another important parameter when provisioning: Henzinger et al. [31] describe a model where a cloud presents different schedules and costs. Other related work [71, 90] uses market approaches to determine when to provision new resources. Our study complements these approaches with a more realistic investigation focusing on simpler policies.

Public IaaS providers also have introduced several related offerings. Amazon Web Services offer the Auto Scaling [5] and Elastic Load Balancing [6] services to the EC2 compute cloud users. Auto Scaling provides the capability to the users to develop their own provisioning policies, by making use of user-defined alarms, i.e., objects that monitor metrics. A policy can be defined through a set of command-line tools supplied by Amazon. The created policies can only be based on the observed VM load, and not on other kind of knowledge, such as the job queue state. The Elastic Load Balancing service distributes incoming application traffic across multiple EC2 instances. However, the user has no control over the scheduling policy that is used to distribute incoming requests. GoGrid offers the f5 load balancer [24], which can use two simple scheduling policies, namely, round-robin or least-connect, which selects the instance with the smallest current load.

Chapter 4

The SkyMark Performance Evaluation Framework

Our approach to analyzing the performance of IaaS environments is SkyMark, a configurable, extensible and portable framework that enables the generation and submission of complex workloads to IaaS cloud environments, using a provisioning and allocation policy specified by the user, prior to the initiation of the experiment. Through the accumulation of statistical information regarding the workload execution, the framework is able to carry out a performance analysis of the underlying IaaS systems.

This chapter presents the features and architecture of SkyMark, and its associated workloads, allocation and provisioning policies. We start with an overview of the framework in Section 4.1. Section 4.2 presents the steps of the experimentation process that lead to analysed results. We then proceed to defining the workloads, in Section 4.3. Section 4.4 presents six provisioning policies and three allocation policies that are evaluated using the manufactured workloads and SkyMark.

4.1 SkyMark Overview

The SkyMark framework is based on two major pre-existing components, Grenchmark and C-Meter, written in Python. These components have been modified and extended, in order to achieve the goals of this work. The modifications/extensions that form SkyMark are also presented in this section.

The components of the SkyMark framework are presented in Figure 4.1. The modules that were subjected to modifications are shown in light gray, while all the newly added modules are drawn in dark grey.

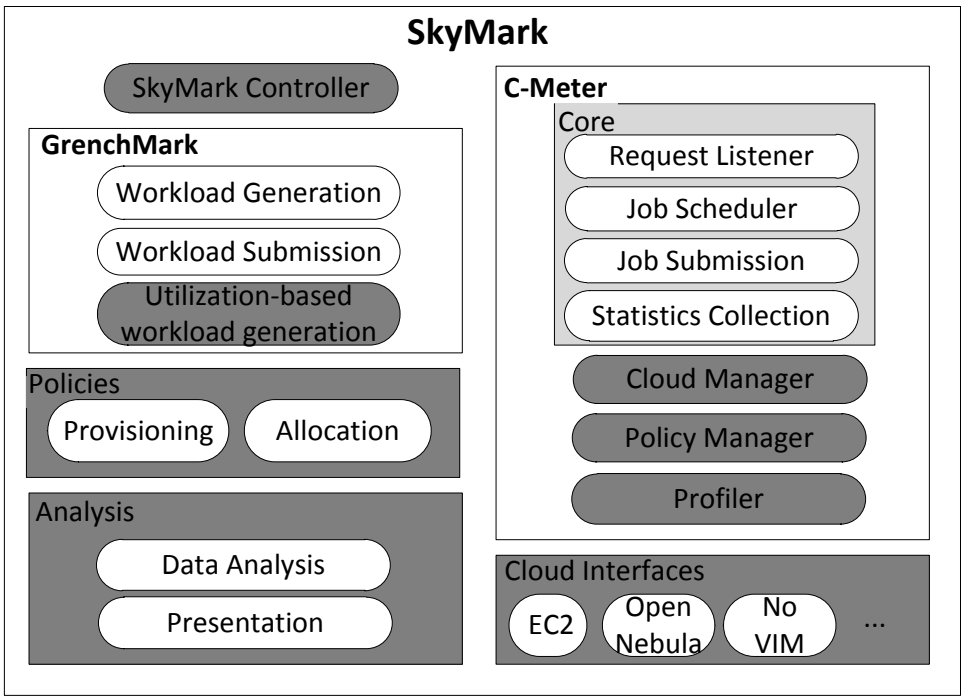

Figure 4.1: The SkyMark architecure. Modified modules are drawn in light grey, and newly-added modules in dark grey.

4.1.1 Grenchmark

Grenchmark [34] is a framework that is able to generate and submit real or synthetic workloads to grid computing environments. Its functionality can therefore be split into two main modules: the *workload generator* and the *workload submitter*.

The workload generator uses pluggable *unit generators* to produce a workload according to the user requirements, specified in a workload description file which is given as input. The unit generators can subsequently use a Job Description File (JDF) *printer* to output the workload. Additional printers can also be plugged-in, thus providing output in different formats.

When designing the workload, the user can specify the job inter-arrival times based on well-known statistical distributions. It is also possible to mix several workloads together, producing a mixture-of-workloads. Lastly, the applications can be both unitary or composite, deployed on a single or multiple sites [34]. An example workload description file is depicted in Figure 4.2. This description specifies a workload mixture with three components, comprising 200 unitary CPU-intensive, memory-intensive and I/O-intensive jobs respectively. The jobs arrive with a poisson distribution with 10 seconds inter-arrival time.

The workload submitter takes as input the generated workload description. It can then deploy the workload units, one-by-one, at the designated job arrival times.

```
#File-type: text/wl-spec
# ID   GeneratorType   Times   UnitType    SiteType   SiteInfo   ArrivalTimeDistr   OtherInfo
0      unitary         200     sser_cpu    single     *:?        Poisson(10s)       StartAt=0
1      unitary         200     sser_mem    single     *:?        Poisson(10s)       -
2      unitary         200     sser_io     single     *:?        Poisson(10s)       -
```

Figure 4.2: An example of a workload description file, given as input to Grenchmark.

```
#File-type: text/wl-spec
# ID    Times    UnitType    OtherInfo
0       200      sser_cpu    StartAt=0
c   Util=0.7, ErrorMargin=0.1, ComputeUnits=4, MaxDuration=20m
1       200      sser_cpu    -
c   Util=0.7, ErrorMargin=0.1, ComputeUnits=8, MaxDuration=20m
2       200      sser_cpu    -
c   Util=0.7, ErrorMargin=0.1, ComputeUnits=16, MaxDuration=20m
```

Figure 4.3: Workload description with specified CPU-utilization configuration. Each workload unit is accompanied with a utilization description line.

The core functionality of Grenchmark did not need modifying for the purposes of this work. However, Grenchmark has been extended with the ability to generate workloads that satisfy a user-specified VM utilization configuration. The utilization-based workload generation was needed for the production of workloads that correspond to the provisioned capacity. Using this module, it is possible to specify the load to be exercised on the acquired virtual resources.

To produce a load-based workload, the profiles of all the job types in the workload must be generated first (see subsection 4.1.2). After the job profile production, the Grenchmark generator will construct a workload, based on the provided description file, an example of which is given in Figure 4.3. The load configuration must be included in this workload description file. For each workload unit, the user must specify the preferred load and the number of virtual machines that this specific load should be applied on. An error margin for the preferred load should also be supplied; the generated load will not deviate more than the error margin from

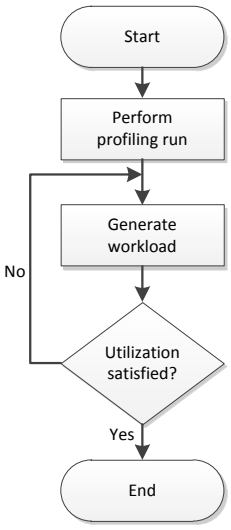

Figure 4.4: The procedure followed for utilization-based workload generation.

the desired load. In the example, we require a workload with an increasing load, starting with 70% for 4 compute units and escalating to 70% for 16 units.

During the workload generation phase, the job profiles are used to estimate the arrival distribution parameters that will produce a workload satisfying the required utilization level. Currently, only the Poisson arrival distribution is used. Subsequently, the module examines whether the generated workload satisfies the required load, for each individual workload unit. The procedure is repeated until a workload with a verified utilization is produced. A flowchart of the used utilization-based workload generation is presented in Figure 4.4.

4.1.2 C-Meter

C-Meter [91] was designed as an extension to Grenchmark, in order to port its capabilities to the cloud. As was originally designed and implemented, C-Meter receives the jobs that comprise the workload, submitted from Grenchmark, and subsequently forwards them to a static pool of resources that have been leased from an IaaS cloud prior to the experiment start.

The original C-Meter design is not adequate to perform the required experiments within the scope of this work. C-Meter was implemented to work with Amazon EC2 and was lacking a refined resource management module, since it was designed to work with a static pool of VM instances. As a result, a large portion of the tool had to be re-factored. In addition, the framework was extended with some additional functionality, namely, the cloud manager, the policy manager, and the job profiler, all of which are described below.

The C-Meter modifications targeted towards a configurable and extensible framework that is able to perform fundamental cloud resource management, while not being dependent on a specific underlying cloud implementation. Since the clouds we were working with exhibited frequent failures, fault-tolerance was an added concern that had to be considered in the resource management component. Moreover, the need of evaluating allocation and provisioning policies has driven us towards providing basic policy management capabilities. By performing simple configuration, the SkyMark user should be able to easily plug-in different policies, in order to perform an experiment. Lastly, the planned analysis methodology as well as the Grenchmark utilization-based workload generator, required the implementation of a profiler module.

In C-Meter, the core functionality is performed by four components, as can be seen in Figure 4.1. These components were already present in the initial version of C-Meter, but they have been modified considerably to fit the purposes of this work:

The *request listener* is responsible for waiting for incoming job execution requests from Grenchmark, as well as for job execution reports from the virtual resource they have been deployed on. Job execution requests received by C-Meter are placed in a queue. The *job scheduler* periodically checks the queue. If there are available resources, then the job at the head of the queue can be assigned to a free

VM instance, according to a scheduling policy that is designated by the *allocation policy manager*. Given a selected idle VM instance, the *job submission* module is responsible for making all the preparations for the job execution on the remote resource, such as copying all the required files to a web server and deploying a job execution agent on the selected VM. The agent downloads the files from the web server, executes the job and upon conclusion, reports back to C-Meter with a collection of gathered statistics. The *statistics collection* unit is responsible for keeping the job execution statistics, as well as statistics regarding the virtual resources, such as the time an instance has been requested and the time it became accessible.

In every scheduling period, the job scheduler makes calls to the *provisioning policy manager*. Depending on the active provisioning policy, the current queue size and the number of idle VM instances, new resources might be requested from the cloud or released back to it. The provisioning and release requests are made to the *cloud manager*. For each experiment session, a cloud-specific interface is dynamically plugged-in by the cloud manager. The preferred interface and all the interface parameters should be provided in the C-Meter configuration file.

The *profiler* component is responsible for identifying all the unique job types in the workload, and subsequently submitting them to the cloud in a dedicated execution mode. Each job has a signature, which is a hash value constructed by using the job's name, its input parameters and input files. The unique jobs of a workload are the jobs that have different signatures. During the dedicated execution mode, each unique job is submitted multiple times. The dedicated execution profiles are created after the data are subjected to basic statistical analysis (outlier detection and averaging). By the end of a profiling session, all the job types observed in a workload should be accompanied by a profile.

4.1.3 Additional Extensions

In addition to the modifications and extensions to Grenchmark and C-Meter, described in the previous subsections, some further work was performed towards implementing the SkyMark framework:

1. **SkyMark Controller**: The main SkyMark module. Experiments were previously performed manually, by firstly generating a Grenchmark workload, setting-up C-Meter and submitting the workload through Grenchmark. This process has now been automated with the addition of this module. Multiple experiments can be defined and will be executed in sequence.

2. **Post-analysis module**: The analysis module was initially part of the C-Meter framework, and could only perform analysis on one dataset at a time. Since the evaluation task of this work requires to make comparisons between different experiment runs, the performance analysis module has been

redesigned. The re-factored module can carry out multi-set analysis and deliver the required graphical output and performance results.

3. **Allocation and provisioning policies**: As was previously described, C-Meter has been modified so that new provisioning and allocation policies can be dynamically plugged-in and used. Based on experience, it is relatively simple and non-time consuming to add new provisioning and allocation policies. For example, adding the most complex policy investigated in this work, `ExecKN`, was a matter of one day's work.

   For the policy evaluation purposes of this work, we have implemented six provisioning and three allocation policies, described extensively in Section 4.4.

4. **Complex workloads**: Several workloads have been formed, that exhibit various arrival patterns and consist of three types of synthetic applications. These workloads are a fundamental part of this work and are described in Section 4.3.

5. **Cloud interfaces**: C-Meter was designed specifically to work with Amazon EC2. The need to experiment on privately-owned infrastructure that is accessible at the physical resource level, has driven us to develop a module that can dynamically load and plug interfaces in to different clouds. Additionally, we have implemented an interface to Eucalyptus, a fairly easy task, since Eucalyptus implements the Amazon EC2 interface. We have also implemented an interface to OpenNebula, using the XML-RPC API [64]. Adding more interfaces is really easy with SkyMark; To be able to instantiate and terminate instances, the corresponding methods for a specific cloud interface have to be implemented.

   Lastly, in order to be able to capture virtualization overheads, we provided Skymark with an interface that is able to submit jobs to non-virtual resources. We make use of this interface to measure the non-virtualized dedicated job run-times. By comparing them to the virtualized job run-times, we will be able to exhibit any overhead introduced by the virtualization layer.

4.2 Experimentation Process

The steps that take place during a typical experiment using Skymark are listed below and depicted in Figure 4.5:

1. The Skymark user prepares two input files, namely, a workload description document (see subsection 4.1.1) and a general configuration file. The latter should specify all the cloud-related configuration, such as the cloud front-end URL, the user credentials to the cloud management platform, and a description of the type and quantity of the required resources.

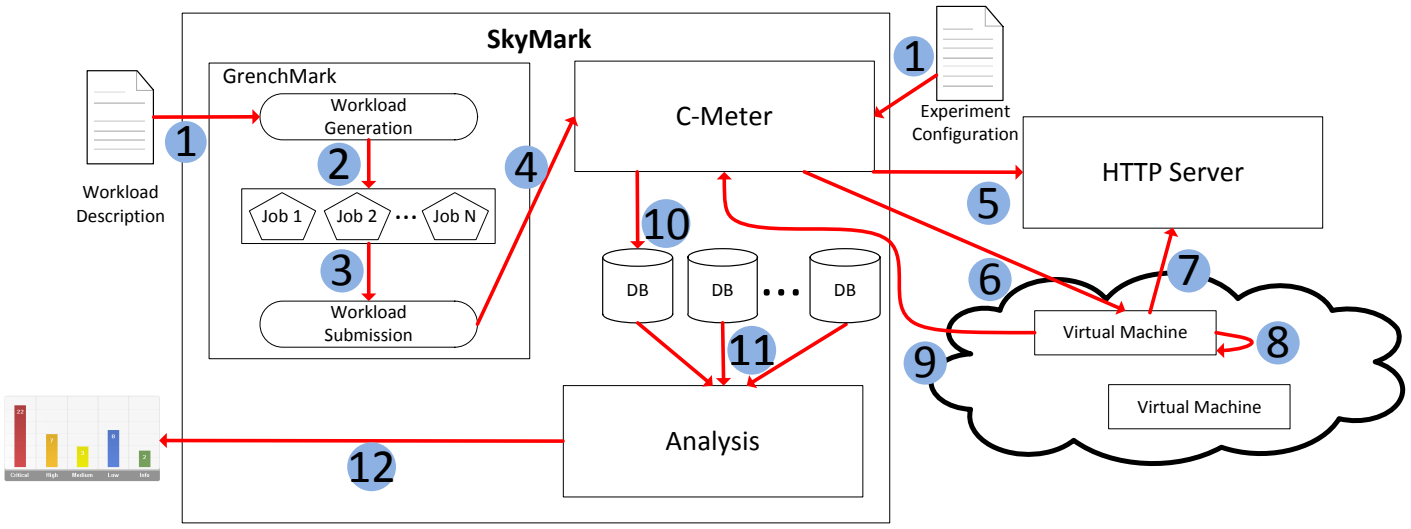

Figure 4.5: The Skymark Experimentation flow.

2. The workload description file is used by the Grenchmark workload generation module to produce a workload described in the Job Submission Description Language (JSDL) [22] format. If the workload requires to have a specified load, then the utilization-based workload generation process shown in Figure 4.4 is followed.

3. Skymark submits the generated workload by using the Grenchmark workload submission unit.

4. C-Meter, which has been already initialized, awaits for new incoming job submissions, with the listener module. A submitted job is received by C-Meter, which subsequently parses its corresponding description file and places the job in a work queue.

5. Next, C-Meter copies the job's binary and all other essential stage-in files to an HTTP server, running on the same host as SkyMark.

6. Skymark decides where to schedule each job that is taken out of the work queue, based on the active *allocation policy*. If there is a lack of available resources, Skymark notifies the plugged-in *provisioning policy*, which will make a decision on whether it should lease more resources from the cloud. As soon as a VM is allocated to the job, C-Meter deploys an execution agent on the provided instance.

7. On the allocated VM, the execution agent downloads the binaries and stage-in files from the HTTP server.

8. The job is executed on the virtual resource.
9. A report containing a collection of statistics regarding the job execution performance is sent back to SkyMark.
10. After all the workload units comprising the workload have been executed, Skymark stores all the collected statistics to an SQLite database.
11. Skymark performs analysis on the data that have been collected throughout one or more experiments.
12. Lastly, the analysis module generates reports and visualizations from the processed data.

4.3 Workload Generation

Defining an appropriate set of workloads is a principal part for this performance evaluation project. Inappropriate workload selection could lead to unfathomable results and eventually to deceptive conclusions. It is therefore important to lay down a set of requirements that the designed workloads should conform to.

4.3.1 Design Requirements

Revisiting our initial hypothesis (Section 1.2), by performing our set of proposed experiments, we would ideally like to observe non-trivial effects on the performance of IaaS systems. Towards achieving this goal, we should make use of *complex* workloads to trigger this behavior. By using such a workload, we allow the jobs that comprise it and the underlying software stack to compete with each other for resources, causing performance phenomena that would not be captured with micro-benchmarks or applications used as individual benchmarks.

Furthermore, the designed workloads should be targeted towards answering a particular research question. Different types of workloads should serve in exhibiting, confirming or contradicting a certain anticipated behavior. It is, for this reason, significant use two sets of workloads, one for each of our initial hypothesis.

The workloads should additionally be *realistic*, representing actual usage scenarios of cloud computing systems. On the other hand, there exists no universally acceptable realistic workload definition for clouds, because of the scarcity of public workload traces or common practice reports. There is a large collection of workloads from parallel and grid environments [37]; however, it is uncertain whether the user of such systems will migrate to IaaS clouds [38]. In this work, we have selected a number of synthetic applications as workload building blocks. Even though the realism of these applications can be challenged, we believe that the construction of workloads using these units will enable us to see resource-specific effects taking place, that would have been more difficult by using only real production traces from non-cloud environments.

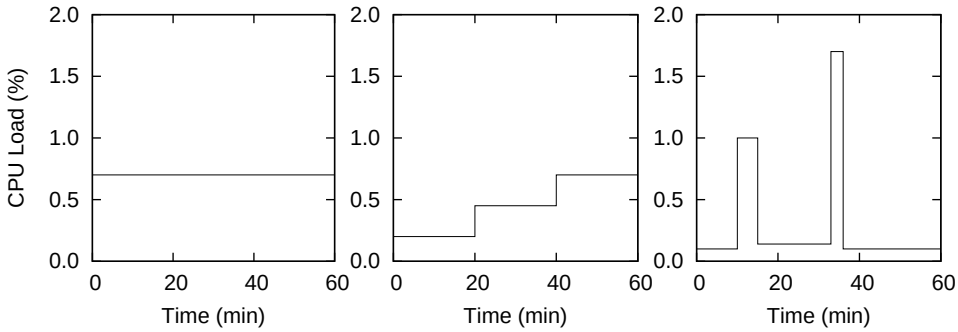

Figure 4.6: The `Uniform` (Left), `Increasing` (Center) and `Bursty` (Right) workload patterns.

Lastly, the workloads should be *diverse*, so that they can drive all of the underlying resources into saturation. At the same time, if we were to stress all resource types simultaneously, the results would be difficult to interpret after analysis. For this reason, it is required that we have an adequate set of workloads, each stressing one or a few resource types individually.

4.3.2 Workload Characterization

The workloads were designed in such a way, so that they put stress on one or two types of resources:

- WL1: **CPU-Intensive** workload;
- WL2: **Memory-Intensive** workload;
- WL3: **I/O-Intensive** workload;
- WL4: **Mixture of Memory and I/O-Intensive** workload.

The composition of the workloads listed above are presented in Subsection 4.3.4.

4.3.3 Workload Patterns

We make use of three different workload patterns:

- WP1: **Uniform**: This pattern maintains a steady stream of jobs throughout the experiment. It uses a Poisson arrival distribution with an average system load of 70%.
- WP2: **Increasing**: This pattern provides a stepwise increase of workload intensity, in three steps. The average system load is around 50%.

Workload Unit	CPU	Memory	I/O	Appears in
WU1	X			WL1
WU2		X		WL2,WL4
WU3			X	WL3,WL4

Table 4.1: Workload unit characteristics and occurrence in workloads

- WP3: **Bursty**: Features short spikes of intense activity amid long periods of mild or moderate activity. For a few minutes the load reaches up to 170%, however, the average system load is around 15%.

These workload patterns are depicted in Figure 4.6.

4.3.4 Workload Units

The workload components are in our case a number of synthetic and real application benchmarks, of variable durations.

Unit Types

The workload unit type can be one of the following:

- WU1: **CPU-Intensive** synthetic application;
- WU2: **Memory-Intensive** synthetic application;
- WU3: **I/O-Intensive** synthetic application.

The characteristics of the workload units are presented in Table 4.1, along with the test workloads they appear in.

Job Durations

For the formation of the job types that will compose the workloads, we consider two current trends for workloads in grid environments. Firstly, grid workloads include many independent single machine jobs, grouped into Bags-Of-Tasks (BoTs) [35]. In contrast, tightly-coupled parallel jobs are less frequent. Additionally, the job runtimes have decreased over the last two decades to the order of seconds to minutes [35], especially for typical data-mining processing [32, 94].

As an example, Facebook uses the MapReduce programming framework to perform operations such as business intelligence, spam detection and ad-optimization [32]. These workloads consist of short jobs, with a median of 84s. Each job is composed of fine-grainded map and reduce tasks, each of which has a runtime in the order of tens of seconds (median 23s).

Job Type	Average Job Duration (s)	Standard Deviation	25th Percentile	75th Percentile
CPU-intensive	47.0	41.0	19.5	57.9
Memory-intensive	46.7	52.7	13.5	65.1
IO-intensive	43.5	45.0	15.1	54.4

Table 4.2: Job Durations for the three job types.

Although SkyMark supports any job lengths, to satisfy the 'realistic' requirement, we have manufactured our synthetic jobs to exhibit durations in the order of tens of seconds. The duration characteristics of the jobs that compose our workloads are presented in Table 4.2.

4.4 Policies

We describe here the provisioning and allocation policies that have been implemented for the analysis purposes of this work. In total, we have implemented 6 provisioning and 3 allocation policies.

4.4.1 Provisioning

A provisioning policy is responsible for acquiring and releasing resources from and to the cloud. For the purposes of this work, we have defined six provisioning policies. These policies can be classified as static or dynamic. Static policies provide a fixed amount of resources, prior to the start of the experiment, while dynamic policies can variate the amount of VMs that are currently leased, according to the demand and to a policy-specific strategy.

Two key points differentiate the policies from each other:

1. **The provision/release trigger**: The policies employ different triggering mechanisms that provision or release resources. For example, the provision trigger could be the start of the workload execution, or the moment when the queue size exceeds a predefined threshold. Similarly, the release trigger could be the completion of the workload execution, or when the queue becomes empty.

2. **The increase/decrease factor**: This factor represents the amount of instances to provide/release when the corresponding event is triggered. The simplest case could be to lease/release one virtual resource.

Table 4.3 provides an overview of the implemented provisioning policies with their characteristics.

Policy	Class	Trigger	Adaptive
Startup	Static	–	–
OnDemand	Dynamic	QueueSize	No
ExecTime	Dynamic	Exec.Time	Yes
ExecAvg	Dynamic	Exec.Time	Yes
ExecKN	Dynamic	Exec.Time	Yes
QueueWait	Dynamic	Wait Time	Yes

Table 4.3: Overview of the provisioning policies.

PP1. Startup

The `Startup` policy leases the specified number of VMs and makes sure that all the requested resources are accessible before initiating the experiments. The resources are released only when the workload has finished executing. An obvious disadvantage of this policy is its inflexibility. The resources remain idle when the load is low, and under heavy load it is not possible to lease new resources. On the other hand, the Startup provisioning policy does not impose any VM instantiation overhead to the workload execution. `Startup` is the only static policy that was implemented.

PP2. OnDemand

The `OnDemand` policy provisions one new instance whenever a new job execution request arrives, and there are currently no available resources that can take up the request. Similarly, it shuts-down a VM whenever it remains idle for a specified amount of time. If the idle-time is set to 0, then each VM is released as soon as its assigned job is finished, if the queue is empty. In general, `OnDemand` can lead to *thrashing*, i.e., frequent leasing and releasing of VM instances.

PP3. ExecTime

This policy assumes known job execution times. `ExecTime` makes use of this information to calculate the aggregate execution time of all the jobs that are currently in the queue, awaiting to be scheduled on a VM instance. When the aggregate execution time exceeds a threshold, `ExecTime` proceeds in leasing a number of instances to meet the demand. Likewise, it releases resources whenever the aggregate execution time drops below a second threshold.

The provision/release thresholds are adaptive to the cloud, i.e., the execution time of the queued jobs must exceed the average time needed to provision and boot a VM instance by a specified factor. The average boot-up time is calculated by using the observed boot-up times of previously leased VMs.

PP4. ExecAvg

This policy is similar to `ExecTime`, but instead of using actual job run-times, it employs a statistical prediction algorithm to predict them. The job run-time is estimated as the average execution time of all the jobs that have already finished. An initial prediction of the average job run-time must be provided by the user.

PP5. ExecKN

Like `ExecAvg`, `ExecKN` also estimates the job execution times. The run-time prediction is based on [41]. For each job in the queue, `ExecKN` acquires its k-nearest neighbors (from the already completed jobs), based on the job input parameter size. The estimated execution time for a job is the average over this set of k neighbors.

PP6. QueueWait

`QueueWait` is another threshold-based dynamic policy, that considers the total queue wait-time of all the jobs currently in the queue. If the total wait-time exceeds or drops below a specified threshold, then the VM provision/release event is triggered respectively. Like the `Exec` family of policies, `QueueWait` also makes use of adaptive thresholds, in order to adjust to the cloud-specific VM boot-up times.

Adaptive Threshold Heuristic (`ATH`)

This simple heuristic allows the policies that use it to adjust to the performance characteristics of the cloud in use. The threshold is modified according to the boot-up times of the already provisioned instances. The weighted average of these instances is calculated and is given as input to the adaptive mechanism. The weights are constructed in such a way, so that the more recently provided instances have more influence in the average boot-up time.

With `ATH`, the policies can adjust to the current cloud behavior. Longer boot-up times will cause the thresholds to rise, so fewer instances will be leased. Shorter boot-up times will have an adverse effect.

Increase Factor (`IF`)

The increase factor determines how many resources will be provided/released, when a resource acquisition/release event is triggered. Three different schemes have been implemented, namely the `Single`, `Multiple`, and `Geometric` `IF` schemes.

The `Single` scheme provides/releases only one VM instance, when the corresponding event takes place. The `Multiple` scheme provides/releases as many VMs as the number of times that the threshold has been exceeded, in order to meet

Policy	Queue-based	Known job durations
FCFS	Yes	No
FCFS-NW	No	No
SJF	Yes	Yes

Table 4.4: Overview of allocation policies.

the demand. Lastly, the `Geometric` scheme starts with acquiring one VM instance, and later increases the number of resources to be acquired by a specified factor (in this work, 2). When the load drops, then `Geometric` starts by releasing one VM, and then increasing in the same fashion the amount of resources to be released.

4.4.2 Allocation

Three basic allocation policies are considered in this work. Table 4.4 summarizes their characteristics.

AP1. First-Come-First-Served (`FCFS`)

A traditional scheduling policy, `FCFS` assigns the job at the head of the work queue to one of the available VMs. The job remains in the queue until a VM becomes available (either a job has finished executing on a VM, or a new VM has been provisioned). `FCFS` is easy to implement, however, since it does not consider the job durations, it can cause low throughput at certain cases.

AP2. FCFS-No Wait (`FCFS-NW`)

This policy does not use a waiting queue. Instead it assigns incoming job execution requests to provisioned VMs that might be currently busy, in a round-robin manner. This policy eliminates waiting in the queue, but introduces resource contention between jobs allocated to the same VM instance.

AP3. Shortest Job First (`SJF`)

`SJF` is another traditional scheduling policy that requires knowing or having an estimation of the job durations. `SJF` gives priority to shorter jobs, by allocating the shortest job in the waiting queue to an idle resource. Although it improves upon the `FCFS` disadvantage, it can lead to long running task starvation, when there are multiple short tasks.

Chapter 5

Experimental Setup

Our performance evaluation is based on measurements over real systems that provide IaaS services. Here, we describe the setup of the experiments. The experiments are divided into two groups. The first set of experiments study the performance of IaaS systems under various types of workloads. The second set of experiments compares six provisioning and three allocation policies.

The chapter is structured as follows: Section 5.1 investigates the complexity of this performance evaluation problem by listing the parameters that can have influence on the results of the experiments. Section 5.2 gives a description of the design of the experiments. Lastly, Section 5.3 presents the chosen performance metrics.

5.1 Parameter identification

Setting up a performance evaluation experiment is usually non-trivial. Our case involves a great number of parameters which make the problem complex. These parameters need to be identified during the experimental design phase; otherwise, the interactions between two or more unidentified parameters might cause undesirable or unexpected effects to the experimental results.

In this section we make a complete list of system, workload and environment characteristics that can have an effect on the performance results, and thus should be considered during the experimentation procedure.

Workload parameters:

1. Composition: The selected test workloads (see Section 4.3) have different characteristics, with regard to the system component that they primarily stress. A workload might be CPU-, memory-, or I/O-intensive. A workload can also be a mixture that stresses two or more system components.

2. Pattern: The workloads might exhibit different arrival patterns of job execution requests. The used arrival patterns were described in Section 4.3.3.

3. Density: The workload density might be defined by manually configuring the arrival distribution parameters, or by indicating a preferred VM CPU utilization factor.

System parameters:

4. System: A distributed system together with a VIM, capable of providing IaaS services. In this work we explore three different systems with the same VIM, and Amazon's EC2 compute cloud, in which the distributed system and the VIM are integrated.

   The selected VIM, OpenNebula, exposes two main configurable parameters (see Subsection 2.3.5):

   (a) Image transfer method: How the VM images are transferred from the image repository node to the VM hosts. Three possible methods exist, NFS-based, localDisk and localDisk with caching.

   (b) Image allocation method: The way that disk images are allocated for VMs. Two methods can be used, namely, eager or lazy allocation.

5. Hypervisor and virtualization type: The underlying hardware virtualization technology. Different systems might be using different hypervisors (e.g., KVM, XEN, VMware) as well as different virtualization types (e.g., full virtualization, paravirtualization, hardware-assisted).

6. Instance type: The defined VM types on different systems might have different capabilities, i.e., there is no universal definition of VM types between systems or cloud providers.

7. Provisioning policy: The strategy for leasing resources from the cloud. The provisioning policy controls the number of virtual resources that are allocated for the execution of a workload. Reducing the number of instances is a desirable scheduling goal.

8. Allocation policy: The policy for scheduling jobs to virtual resources. It is used to manage the already provisioned resources.

9. VM operating system, e.g, Linux distribution, Windows, 32/64-bit architecture.

Environment parameters:

10. System background load: Whether the physical resources of the system are dedicated to the execution of the workload, or are shared with other users that deploy workloads of varying characteristics.

11. SkyMark performance: We show in the experiments (Section 6.1) that SkyMark imposes a small performance overhead.

Factor	Levels
Workload composition	WL1-WL4 (Section 4.3.2)
Workload pattern	WP1-WP3 (Subsection 4.3.3)
Provisioning Policy	PP1-PP6 (Subsection 4.4.1)
Allocation Policy	AP1-AP3 (Subsection 4.4.2)
Systems	Sys1-Sys4 (Subsection 5.2.3)
Instance number	1 up to system capacity (Subsection 5.2.3)
Instance type	Small (Subsection 5.2.4)

Table 5.1: Factors and their levels.

5.1.1 Interactions between parameters

Some of the factors of this experimental setup interfere with each other. The identified parameter interactions are listed below:

1. System - Instance Type: The capabilities of the VM types depend on the system they are defined for.

2. System - Hypervisor: Each system is equipped with one hypervisor. Depending on the hypervisor, different systems employ different virtualization types. Lastly, some of the systems use hardware-assisted virtualization and some do not.

3. System - VIM: Each system is equipped with a different VIM.

Comparisons between different systems and VIMs cannot take place without considering these interactions, because any observations made on the performance variations between the participating systems might be caused by them.

5.1.2 Selecting factors

In this work, emphasis is given to the impact of Workloads and Provisioning/ Allocation policies on the performance of clouds. These are therefore the *primary factors* in the experimentation process.

The systems under test, the number of instances, the VIM, and the instance type are *secondary factors*, meaning that we will not be quantifying their impact. Table 5.1 presents all the factors of the experiments along with their levels of operation.

5.2 Experimental Design

We now design the experiments for this thesis, using the parameter investigation performed in the previous section.

5.2.1 Experiment Specification

Here, we describe the two sets of experiments that will be performed for this work.

IaaS evaluation

The intent of the first part of the experiments is to study the cloud performance under various types of workloads, that were described in Section 4.3.2. We characterize the cloud performance by breaking down the job slowdown, into its main overhead causes. We have identified the following overheads:

1. **Virtualization**.The virtualization overhead, refers to the cost added by executing jobs inside of VM instances, imposed by the virtualization layer.

2. **VM Contention**.VM contention is the overhead originating from the resource time-sharing between VM instances on a single physical host.

3. **Complex Workloads**. The Complex Workloads overhead originates from the execution of multiple jobs on the same VM instance, thus creating virtual-resource contention between jobs.

4. **Other**. The remaining overhead is attributed to both SkyMark and network delays. The SkyMark cost originates from the job submission preparation and scheduling phases that each job execution request must go through.

We are able to isolate the four overheads by performing the following experimental process:
Each workload is executed twice, with two allocation policies, `FCFS` and `FCFS-NW`, while keeping the provisioning policy fixed at `Startup`. At the same time, for each unique job in the workloads, we perform a profiling run, inside of a VM instance and additionally on the physical host. Therefore, we collect two types of profiles for each job, one with *virtualized* and another one with *non-virtualized* execution.

With the use of `FCFS-NW`, we expect to capture all four overheads. On the other hand, the Complex Workload overhead will not be present in the workload execution with `FCFS`, since each VM instance can have one job at most allocated to it. Subtracting the `FCFS` slowdown from the `FCFS-NW` slowdown, while comparing to the virtualized profiles, is expected to provide us with the Complex-Workload overhead.

Policy	Adaptive	Lease Threshold	Release Threshold	Increase Factor
Startup	–	–	–	–
OnDemand	No	–	–	Single
ExecTime	Yes	$5\times$ boot−up	$0.3\times$ boot−up	Multiple
ExecAvg	Yes	$5\times$ boot−up	$0.3\times$ boot−up	Multiple
ExecKN	Yes, No	$5\times$ boot−up	$0.3\times$ boot−up	Single, Multiple, Geometric
QueueWait	Yes	$5\times$ boot−up	$0.3\times$ boot−up	Multiple

Table 5.2: The provisioning policy parameter configuration. `ExecKN` is used for the evaluation of `ATH` and `IF`, therefore these two factors also variate with the use of this policy.

To isolate the virtualization overhead, we calculate the job slowdown for the workload execution with `FCFS`, against the virtualized and non-virtualized profiles. The performance difference between them should give us the overhead imposed by the virtualization layer.

The VM contention overhead can be found by calculating the job slowdown with `FCFS`, against the virtualized job profiles. This performance deviation between workload execution and dedicated job execution (inside a VM) should be caused by VM contention, because of resource time sharing. In a dedicated execution, the VM contention overhead does not exist. Finally, the difference between the total overhead and the sum of the three overheads mentioned above, can be attributed to the "other" overhead.

Policy Evaluation

The policy analysis compares the performance of the six provisioning policies and three allocation policies, described in section 4.4. For the provisioning policy evaluation, we keep the allocation policy fixed to `FCFS` and subsequently compare the policies on several metrics, presented in the next section. For the evaluation of adaptive threshold heuristic and the increase factor alternatives, we use only the `ExecKN` policy. Table 5.2 describes all the used policy configurations. In a similar manner, we use the `Startup` provisioning policy to test the allocation policies.

5.2.2 Workload Specification

Workloads for IaaS evaluation

For this evaluation section we make use of the full set of workloads that were defined in Section 4.3.

Workloads for Policy evaluation

As mentioned earlier, the goal of the second part of this work is to observe how the performance of IaaS is affected under the execution of complex workloads, when different scheduling (allocation and provisioning) policies are used. The workloads should therefore be designed to observe the differences in behavior and performance of the policies.

For the purpose of this work, we require only a subset of the workloads that are defined for the IaaS evaluation section. More specifically, a workload that drives only one type of resource to saturation is required for this experimentation session. We have selected WL1, a CPU-Intensive workload.

5.2.3 System Specification

For our performance evaluation purposes, a system is a distributed system equipped with a certain Virtual Infrastructure Manager (VIM). We have made use of several such systems. Table 5.3 lists the systems that were used in our tests.

The systems specification demonstrates a variety of capabilities across the participating systems. Some are of particularly small capacity, such as Sys3. Additionally, the hardware in Sys3 is quite outdated, so it does not provide hardware assisted virtualization. DAS4 [85] (Sys1 and Sys2) has better hardware and larger capacity, but only a small fraction of the total resources could be allocated to this work. The physical resource specification of Amazon EC2 is not made publicly known. It is certain, however that different EC2 instance types use different physical hardware [3]. There is, therefore, no single hardware specification for Amazon EC2. The capacity of EC2 is very large, but we limit our experiments to 20 VMs to put a limit on the cost.

Systems Sys1-Sys3 make use of the OpenNebula VIM, but they differ on the versions of these software platforms. Amazon EC2 makes use of its own VIM about which information is publicly available.

We performed the first set of experiments on Sys2. Sys1 was also used for experimentation, however, because of VM failures, we only report on the observed problematic behavior. For the policy experiments, systems Sys2-Sys4 were used.

5.2.4 Instance Specification

We use in our work one VM type, which is specifically configured for each system as summarized in Table 5.4, but overall, the instance types on the used systems are

Code	Description	Hardware Spec	VIM	Hypervisor	Max VMs
Sys1	Distributed ASCI Supercomputer 4 at VU *(DAS4/VU)*	**32** Dual quad-core 2.4 GHz (2 hardware threads) 24 GB RAM 2*1 TB storage	OpenNebula 3.0 LocalDisk-caching Lazy allocation	KVM (Full, HVM)	512
Sys2	Distributed ASCI Supercomputer 4 at Delft *(DAS4/Delft)*	**20** Dual quad-core 2.4 GHz (2 hardware threads) 24 GB RAM 2*1 TB storage	OpenNebula 3.0 LocalDisk-caching Lazy allocation	KVM (Full, HVM)	256
Sys3	Florida International University Cluster (*FIU*)	**7** Pentium 4 3.0 GHz 5 GB Memory 340 GB storage	OpenNebula 2.2 NFS Lazy allocation	XEN Paravirtualization No HVM	7
Sys4	Amazon EC2 Public Cloud, eu-west-1 region	unknown/varying	Amazon EC2	XEN (paravirtualization, no HVM)	20

Table 5.3: Description of systems under test

very similar to each other.

We primarily make use of lazy allocation (qcow2 image) throughout the experiments (see Subsection 2.3.5). The staging time for lazy allocation is much shorter, since the image size is smaller. During the staging phase in cloud environments, the image has to be transferred to the VM instance host machine. Therefore, the total boot-up time is expected to be much shorter than for eager allocation (RAW image). There is a price to pay, however, for the lazy allocation. The disk write access overhead is much higher than for eager allocation, since any disk write call has to make a storage allocation request to the host OS. To avoid including this cost in the virtualization overhead, *we use eager allocation* for all experiments that involve I/O-intensive workloads, and *lazy allocation* for all other experiments.

5.3 Performance Metrics

Several traditional [36] metrics have been chosen, in order to assess the performance of clouds:

System	OS	Instance Specification			
		Virt. cores	Mem.	Disk	Platform
Sys1	CentOS 5.4 x86-64	1	1GB	5GB	64-bit
Sys2	CentOS 5.4 x86-64	1	1GB	5GB	64-bit
Sys3	CentOS 5.3 i386	1	512MB	3GB	32-bit
Sys4	RHEL 6.1 i386 ID:ami-9289bae6	1 (*m1.small*)	1.7GB	160GB	32-bit

Table 5.4: Specification of VM Instances across environments.

Job Wait Time (WT)

The time each job waits in the queue before it is dispatched for execution on an available virtual resource.

Job Response Time (ReT)

The time between the job arrival at SkyMark, and the receipt of a report from the virtual resource it was executed on.

Workload Makespan (MS)

The workload makespan is defined as the interval between the time that the first job in the workload arrives at SkyMark, and the time that the execution results of the last job in the workload have been received by Skymark:

$$MS(W) = t_{lc} - t_{fa} \tag{5.1}$$

where t_{fa} is the arrival time at SkyMark of the first job in the workload, and t_{lc} is the time SkyMark receives the completion report for the last job in the workload.

Job Slowdown (`JSD`)

`JSD` for each job in the workload, is the ratio of the actual runtime in the cloud and the runtime in a dedicated environment.

Workload Speedup One (`SU`$_1$)

The workload speedup is the ratio between its makespan and the sum of its job runtimes in a dedicated environment.

$$SU_1(W) = \frac{MS(W)}{\sum_{i \in W} t_R(i)} \tag{5.2}$$

where $\mathtt{SU}_1$(W) is the speedup for workload W,$MS(W)$ is the makespan for workload W, and $\mathtt{t}_R(i)$ is the dedicated runtime for job i, that belongs in W. Intuitively, this metric exhibits the performance gained against the execution on a single-processor machine. SU_1 has values above 1 and is, theoretically, not bound.

Workload slowdown infinite ($\mathtt{SD}_\infty$)

$\mathtt{SD}_\infty$ represents the slowdown against an infinitely large system. SD_∞ of workload W is the ratio between its makespan and the maximum of the job runtimes in a dedicated environment:

$$SD_\infty(W) = \frac{MS(W)}{\max_{i \in W} t_R(i)} \tag{5.3}$$

where $\mathtt{SD}_\infty$(W) is the infinite slowdown for workload W, `MS(W)` is the makespan for workload W, and $\mathtt{t}_R(i)$ is the dedicated runtime for job i. SD∞ has values above 1.

Actual Cost (C_a)

The actual workload execution cost is the aggregated amount of time that each instance participating in the workload execution has been running for. This time is measured from the moment an instance is requested, until the time it is shutdown.

$$C_a(W) = \sum_{i \in leased\ VMs} t_{stop}(i) - t_{start}(i) \tag{5.4}$$

where $\mathtt{t}_{start}(i)$ is the time instance i has been leased from the cloud, and $\mathtt{t}_{stop}(i)$ is the time SkyMark made the request for instance i to be shut-down.

Charged cost (C_c)

The charged cost follows Amazon's pricing policy for EC2. Amazon charges per hour of use of each leased instance.

$$C_c(W) = \sum_{i \in leased\ VMs} \lceil t_{stop}(i) - t_{start}(i) \rceil \tag{5.5}$$

where $\mathtt{t}_{start}(i)$ is the time instance i has been leased from the cloud, $\mathtt{t}_{stop}(i)$ is the time SkyMark made the request for instance i to be shut-down, and $\lceil\rceil$ is the ceiling to the nearest hour. This metric does not reflect special pricing policies, such as premium charges for long-term users, discounts for starting users, etc.

Cost Efficiency ($\mathtt{C}_{eff}$)

Cost efficiency is defined as the ratio of the charged and actual cost:

$$C_{eff}(W) = \frac{C_c(W)}{C_a(W)} \tag{5.6}$$

Utility (U)

Utility is a compound metric that rewards low performance overheads and low cost:

$$U(W) = \frac{SU_1(W)}{C_c(W)} \tag{5.7}$$

All the selected metrics can be accepted as good metrics. They all exhibit the desired characteristics that were described in Subsection 2.1.2.

Chapter 6

Experimental Results

In this chapter, we present our findings, from the experimentation performed with SkyMark. The chapter is divided into two sections: Section 6.1 studies the overheads imposed when executing different types of workloads in cloud environments. Section 6.2 evaluates the performance of several provisioning and allocation policies, when executing CPU-intensive workloads on different systems.

6.1 IaaS evaluation

As was explained in Subsection 5.2.4, we use eager allocation for the experiments that involve I/O-intensive workloads, and lazy allocation for the rest. To justify this decision, we exhibit the instance boot-up time/disk access time trade-off in Figures 6.1 and 6.2, where we request 64 VM instances to be provided at the start of the experiment. The observed boot-up time with lazy allocation is roughly three times shorter than with eager allocation. However, the captured virtualization cost with the use of an I/O-intensive workload is twice as much. Therefore, we had to switch to eager allocation, just for these workloads, so that there are no interferences of the lazy allocation strategy with the virtualization performance.

6.1.1 Uniform Workloads

Figures 6.3, 6.4, 6.5 and 6.6, show the breakdown of the JSD for `Uniform` workloads, with CPU-intensive, Memory-intensive, I/O-intensive, and Memory+I/O-intensive jobs, respectively.

From these graphs, we make several observations. Firstly, the JSD caused by the virtualization layer remains stable throughout the workload execution, for all job types. However, the slowdown suffered with Memory-Intensive jobs ($\approx$1.06) is slightly larger than with CPU-intensive jobs ($\approx$1.03), and considerably larger with I/O-intensive-jobs ($\approx$1.30). The virtualization overhead with the Memory-I/O mixture is less ($\approx$1.20) than with I/O jobs alone.

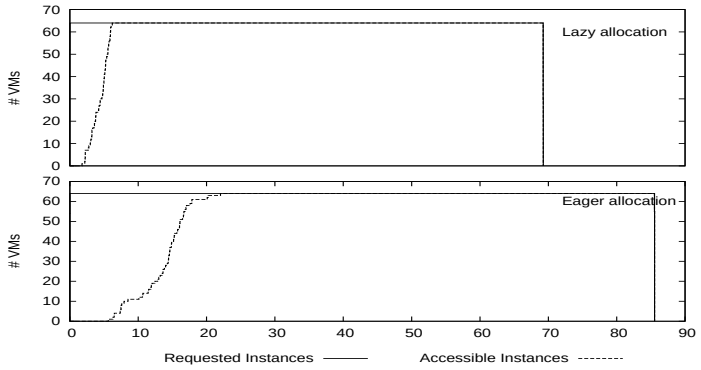

Figure 6.1: Comparison of boot-up times between lazy and eager allocation, for experiments with 64 VMs, on DAS4/Delft.

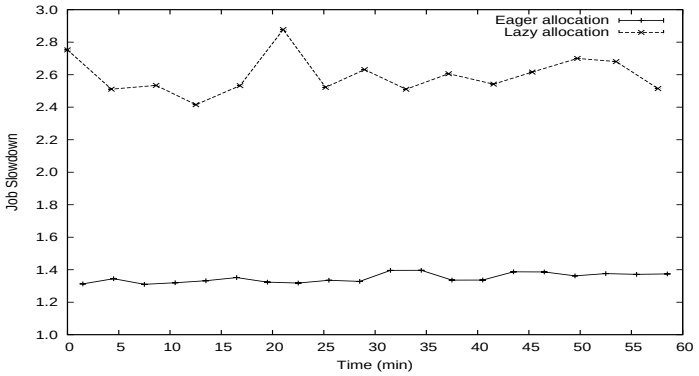

Figure 6.2: Comparison of virtualization cost for `I/O-intensive, Uniform` workload, between lazy and eager allocation, on DAS4/Delft.

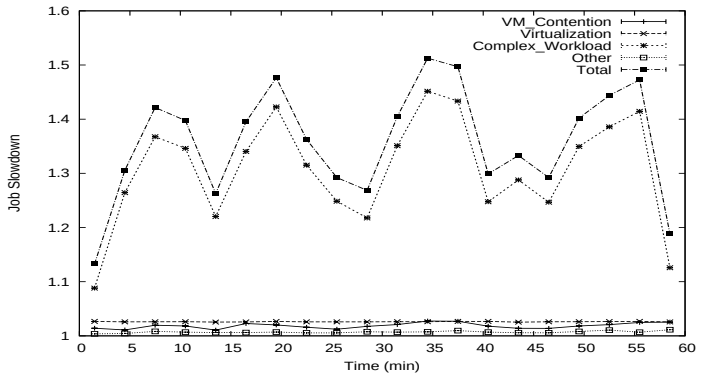

Figure 6.3: Job Slowdown Breakdown for `CPU-Intensive, Uniform` workload, on DAS4/Delft.

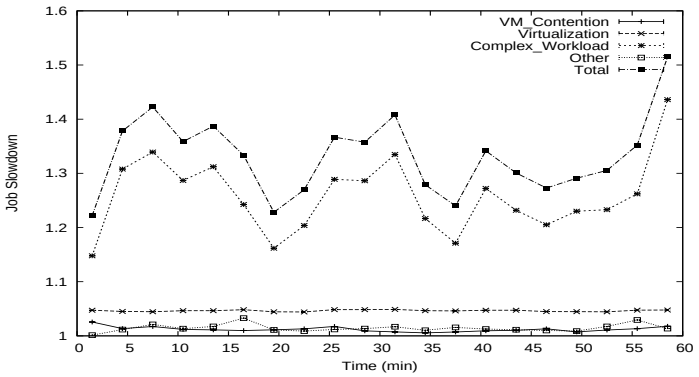

Figure 6.4: Job Slowdown Breakdown for `Mem-Intensive, Uniform` workload, on DAS4/Delft.

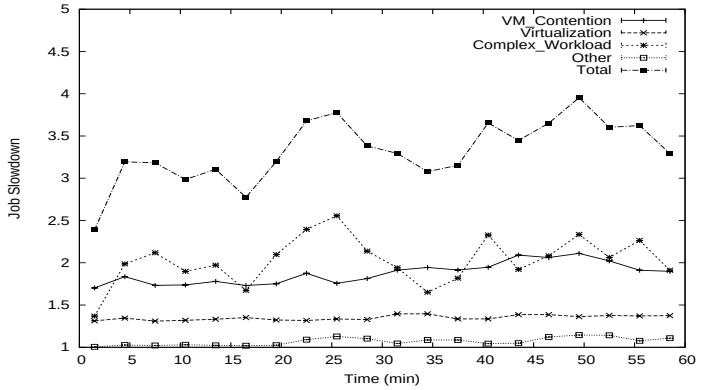

Figure 6.5: Job Slowdown Breakdown for `I/O-Intensive, Uniform` workload, with eager allocation, on DAS4/Delft.

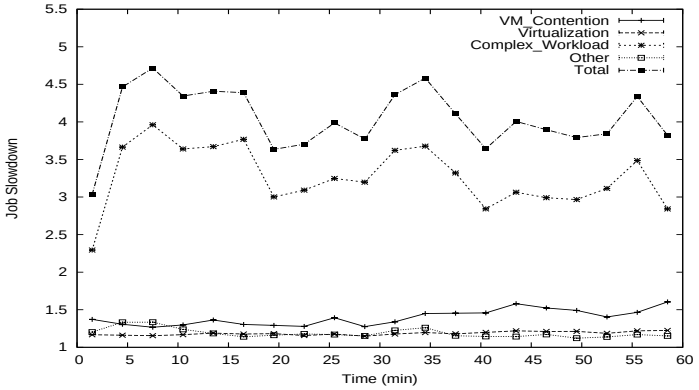

Figure 6.6: Job Slowdown Breakdown for `Mem+ I/O-Intensive, Uniform` workload, with eager allocation, on DAS4/Delft.

Second, there is a large JSD variation caused by the complex workload overhead. This overhead is about the same for CPU-intensive and Memory-intensive workloads (≈1.3), but grows significantly for I/O-intensive (≈2.0) and Mem+I/O-intensive workloads (≈3.0).

Furthermore, the VM contention has small variations over time, especially for the CPU and Memory-Intensive workloads. These variations are larger in the workload mixture and the I/O-intensive workload. What is most important, however, is that the VM contention overhead imposed with the I/O and Memory+I/0-intensive workloads are considerable (≈2.0 and 1.4 respectively), while insignificant for the CPU and Memory-intensive workloads (≈1.03 for both).

Lastly, the overhead imposed by SkyMark and additional network delays (other) is very small and is relatively stable. Overall, the total slowdown is the greatest for the workload mixture (≈4.0), followed by the I/O-intensive workload (≈3.5). The largest component of the total slowdown is the complex workload overhead.

6.1.2 Increasing Workload

The same job slowdown breakdown is presented for increasing workloads, in Figures 6.7, 6.8, 6.9 and 6.10 for the four types of jobs.

For the increasing workloads, we observe that the Complex Workload overhead increases as the load increases. This stands for all types of jobs. As with the uniform workloads, the complex workload overhead is the most significant source of job slowdown, and is more substantial for the workloads that include I/O-intensive jobs. Likewise, the VM contention overhead also increases with the load. This is more apparent for the I/O and Memory+I/O-intentensive workloads.

The virtualization overhead remains stable over time for the CPU and Memory-intensive workloads, but a slight increase is observed for the I/O-intensive workload and the and mixture of workloads. This virtualization cost increase can be attributed to the non-fixed disk read/write access time. As the I/O load increases, with multiple files being read and written to and from the disk, the cost of a single block read or write increases. This is because the average seek and rotation times increase. Thus, what we capture as virtualization cost growth for I/O-intensive workloads, *originates in reality from the other two overheads*: the contention between VMs and the complex workloads. The reason why we capture it as virtualization overhead, is because it is an indirect of time-sharing.

6.1.3 Bursty Workload

We finally show the job slowdown breakdown for Bursty workloads and the four job types, in Figures 6.11, 6.12, 6.13 and 6.14.

Here, the VM Contention increases during the two short load bursts. The increase differs from job type to job type. It is almost negligible for the CPU-intensive jobs (at most 1.03) and Memory-intensive jobs (at most 1.05), but is an

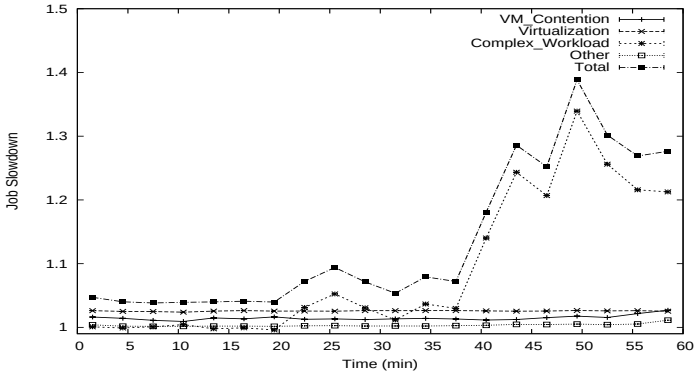

Figure 6.7: Job Slowdown Breakdown for `CPU-Intensive, Increasing` workload, on DAS4/Delft.

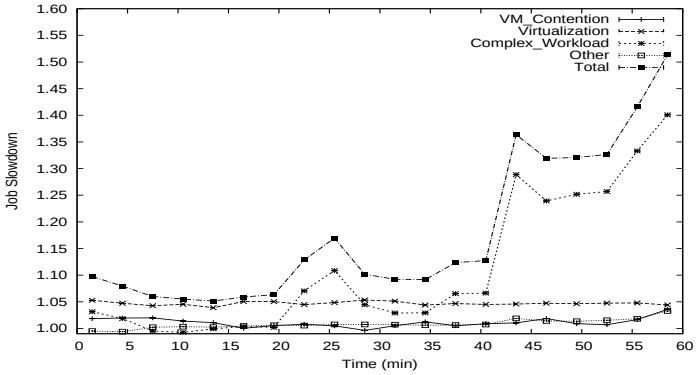

Figure 6.8: Job Slowdown Breakdown for `Mem-Intensive, Increasing` workload, on DAS4/Delft.

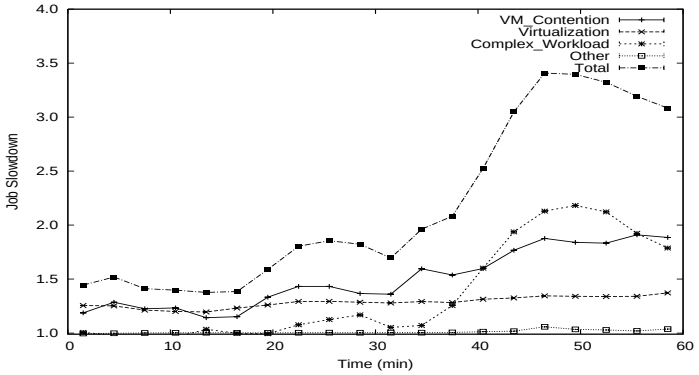

Figure 6.9: Job Slowdown Breakdown for `I/O-Intensive, Increasing` workload, with eager allocation, on DAS4/Delft.

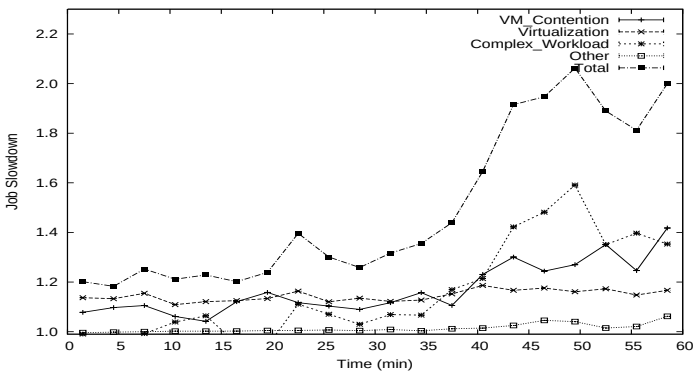

Figure 6.10: Job Slowdown Breakdown for `Mem+I/O-Intensive, Increasing` workload, with eager allocation, on DAS4/Delft.

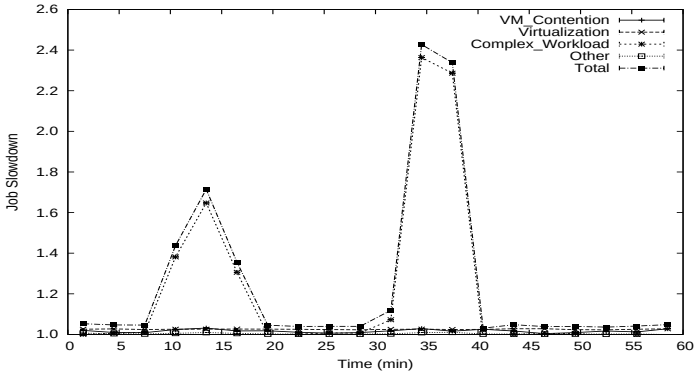

Figure 6.11: Job Slowdown Breakdown for `CPU-Intensive, Bursty` workload, on DAS4/Delft.

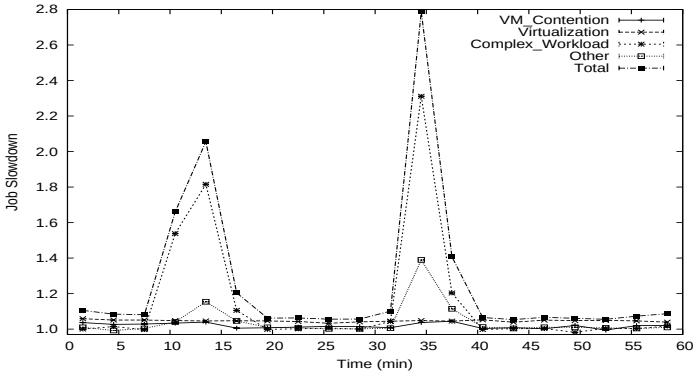

Figure 6.12: Job Slowdown Breakdown for `Mem-Intensive, Bursty` workload, on DAS4/Delft.

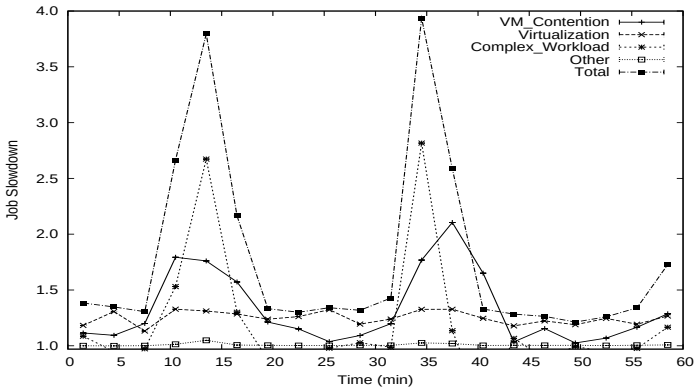

Figure 6.13: Job Slowdown Breakdown for `I/O-Intensive, Bursty` workload, with eager allocation, on DAS4/Delft.

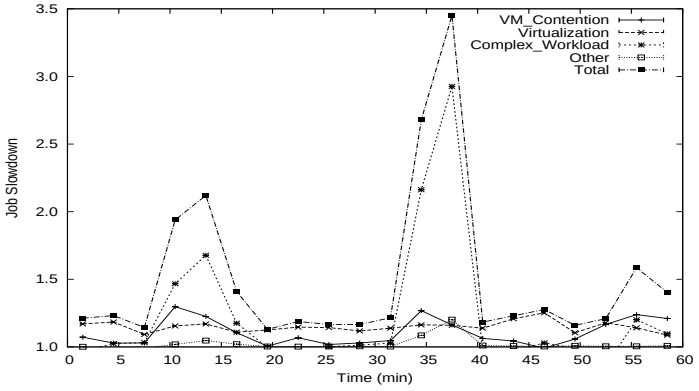

Figure 6.14: Job Slowdown Breakdown for `Mem+I/O-Intensive, Bursty` workload, with eager allocation, on DAS4/Delft.

important contributor to the total job slowdown for the workloads that include I/O-intensive jobs (at most 2.0 and 1.3 respectively).

As with the increasing workload, the virtualization cost is stable for CPU and Memory-intensive jobs, but variates slightly for the other two workload types, with peaks at the moments of intense load.

The overhead imposed by SkyMark and additional network delays (other) has a considerable increase at the moments of intense load of the Memory-intensive and Memory+I/O-intensive workloads. One possible explanation for this, is that the experimentation node was placed inside the tested IaaS cloud. Consequently, the workload caused some interference with SkyMark, which uses memory more than the other types of resources. Placing SkyMark outside DAS4/Delft was not an option, since VM instances could not be made accessible from outside the local network.

6.2 Policy evaluation

In this section we present our findings on the impact of different provisioning and allocation policies, on the performance of the workload execution. We want to determine which policies perform better, and which ones offer the best performance-cost trade-off.

6.2.1 Provisioning

We first explore the effect of the provisioning policies. To this end, we use the same allocation policy, `FCFS`, coupled in turn with each one of the provisioning policies. We show the results in Figures 6.15-6.25 and in Tables 6.1-6.3.

Figures 6.15-6.18 present the workload makespan (`WMS`), the job slowdown

(JSD), the workload speedup against a single node (SU_1) and the workload slowdown against an infinetely large system (SD_∞) respectively. From these figures, it is apparent that Startup always achieves the best performance. OD-S has similar performance for the uniform workload, but is not as good for the variable workloads. From the threshold-based policies, QueueWait usually performs better than the rest, because it reacts faster to load variation. ExecTime and its variants have similar performance, with ExecTime usually performing better, since ExecAvg and ExecKN do not have exact job runtime information.

The actual cost (C_a) and charged cost (C_c) are presented in Figures 6.19 and 6.20 respectively. Even though Startup incurs the highest actual cost, since it acquires the full set of resources from the start until the end of the experiments, it is OD-S that costs the most according to Amazon's billing scheme. With OD-S the VMs are started and stopped reactively to individual job arrivals. The group of threshold-based policies and especially the Exec family of policies significantly reduce the cost of workload execution. The cost reduction becomes bigger for the increasing and bursty workloads. QueueWait appears to have similar actual cost to the Exec policies, however the charged cost is higher, especially for the variable-load workloads.

Figures 6.21 and 6.22 show the cost efficiency (C_{eff}) and utility (U) for the set of provisioning policies. The dynamic policies hold on to resources for shorter periods of time than Startup, especially for bursty workloads. This leads to a worse cost efficiency value. However, they do achieve better utility scores, which means that they provide a better performance-cost trade-off. The charged cost(C_c), cost efficiency (C_{eff}) and utility (U) values for the three clouds are presented in Tables 6.1-6.3.

More insight about the provisioning policies can be gained from Figures 6.23-6.25. Here, the number of requested and acquired resources is plotted over time, for the DAS4 cloud, when under the three different types of workloads. OD-S requests and releases resources very often, leading to VM thrashing. Particularly visible under the increasing and bursty workloads is that OD-S often releases resources before they even become accessible. QueueWait reacts faster to load variation, thus leading to higher cost. The Exec group of policies, and especially the two run-time estimating policies, have slower reaction to load variation. The discrepancy between ExecTime and its variants is caused by a bad initial prediction of the job run-time, that is manually configured prior to the start of the experiments. For the bursty workload, the inaccurate prediction of the run-time leads to better behavior for the two estimation-based policies.

Impact of the Adaptive Threshold Heuristic (ATH)

In this section we evaluate the effect of ATH on the performance of the provisioning policies. Using the ExecKN provisioning policy, we toggle ATH on or off. The experiments with the adaptive mechanism turned off involve two scenarios. The

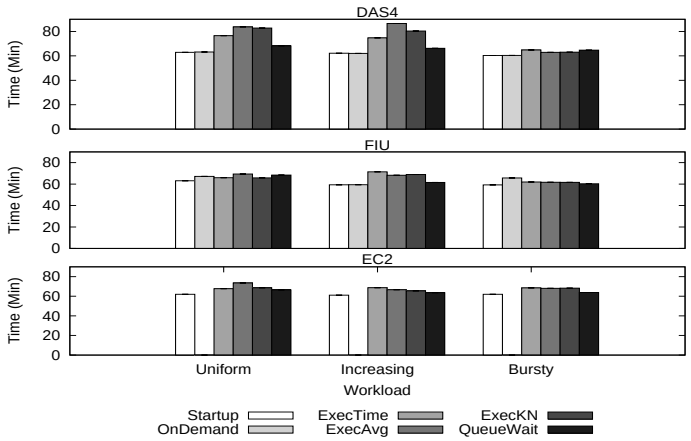

Figure 6.15: Workload Makespan (WMS).

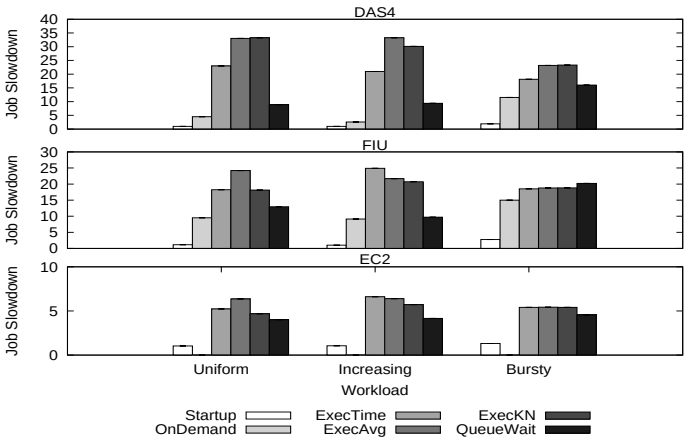

Figure 6.16: Job Slowdown (JSD).

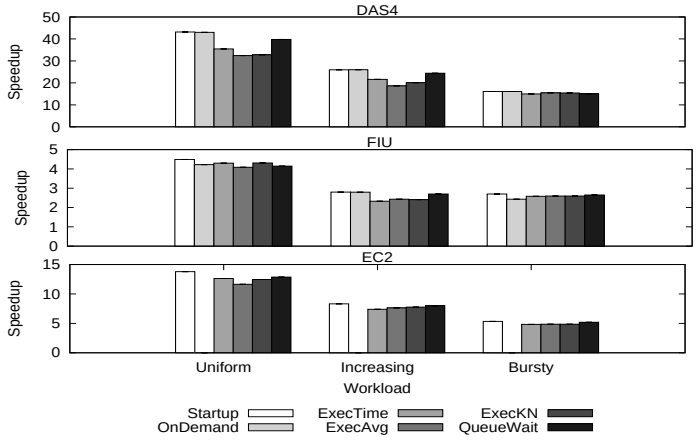

Figure 6.17: Workload speedup (SU_1).

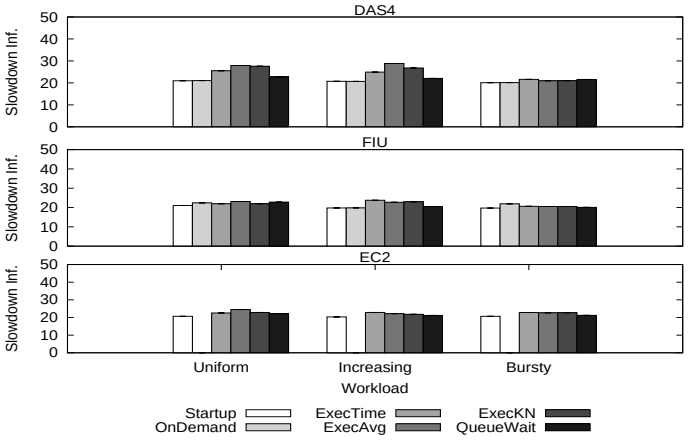

Figure 6.18: Workload Slowdown Inf. (SD_∞).

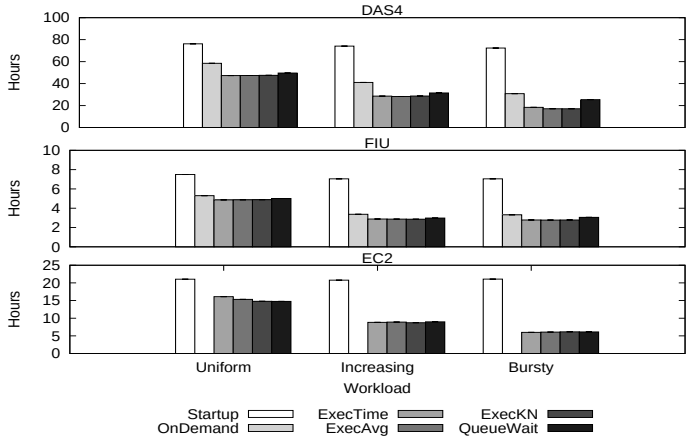

Figure 6.19: Actual cost (C_a).

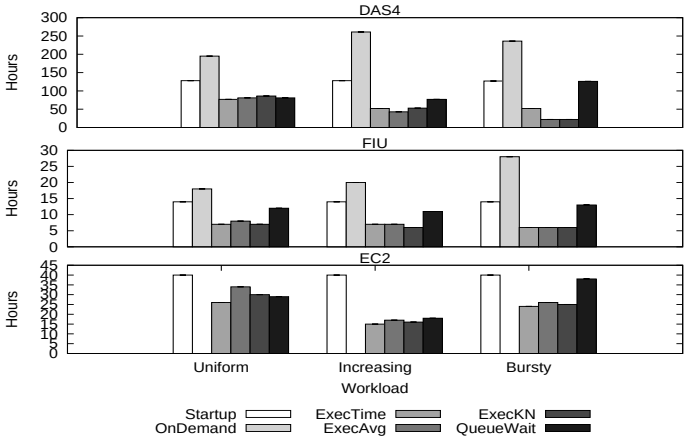

Figure 6.20: Charged cost (C_c).

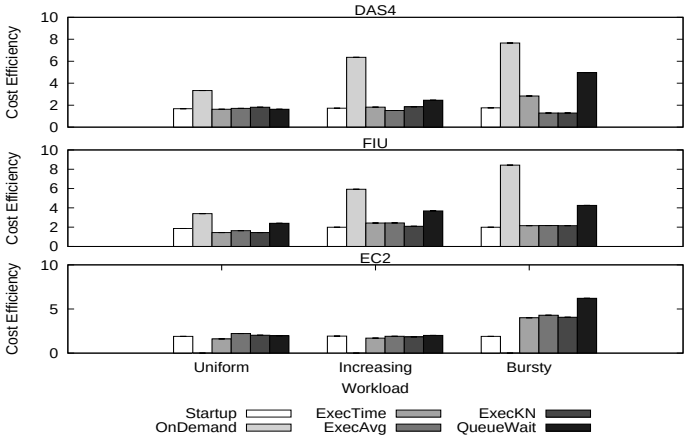

Figure 6.21: Cost efficiency (C_{eff}).

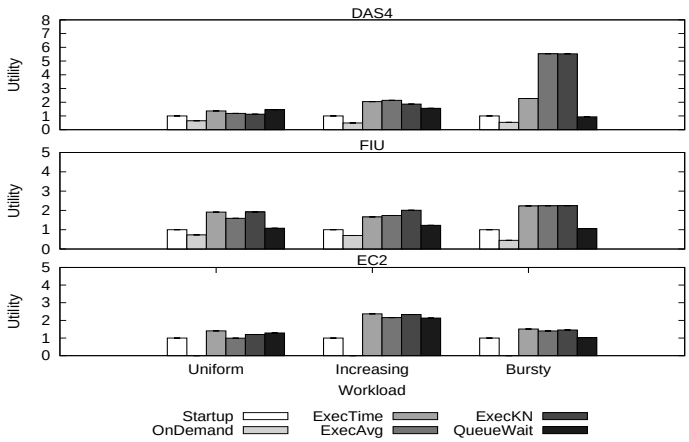

Figure 6.22: Utility (U).

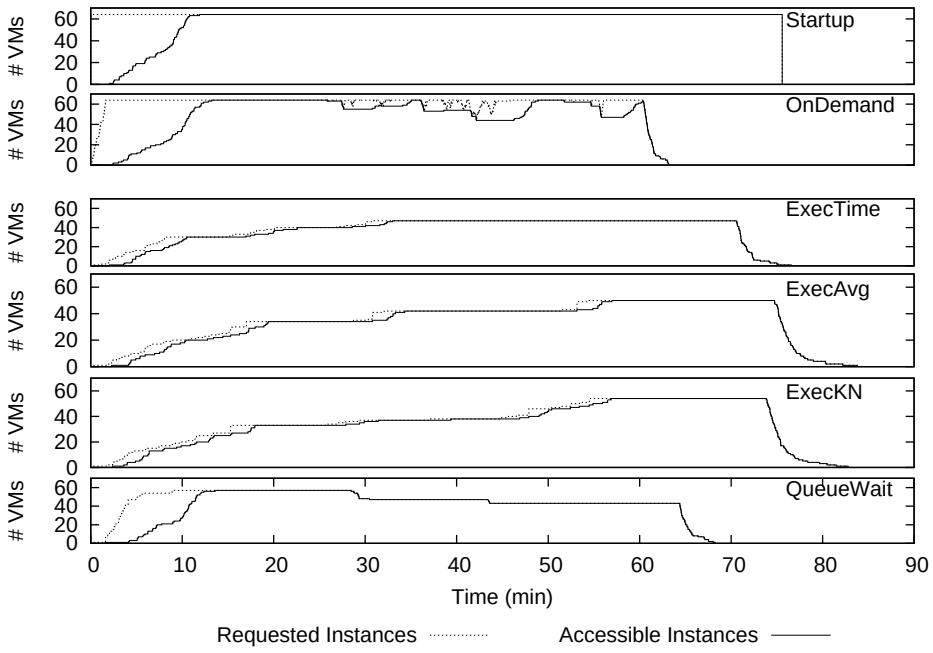

Figure 6.23: Instances over time for the provisioning policies with the *Uniform* workload on DAS4.

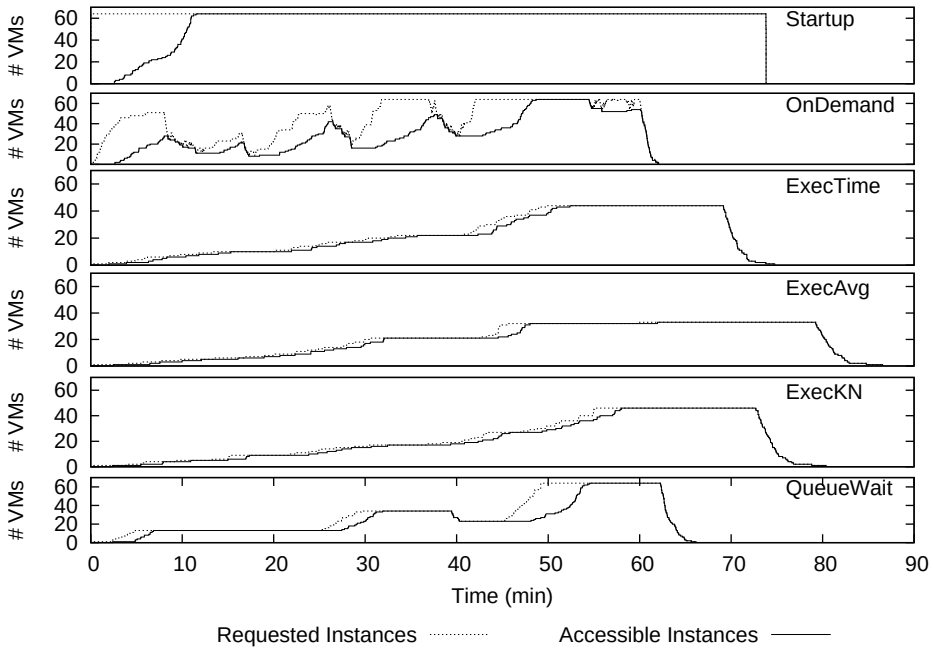

Figure 6.24: Instances over time for the provisioning policies with the *Increasing* workload on DAS4.

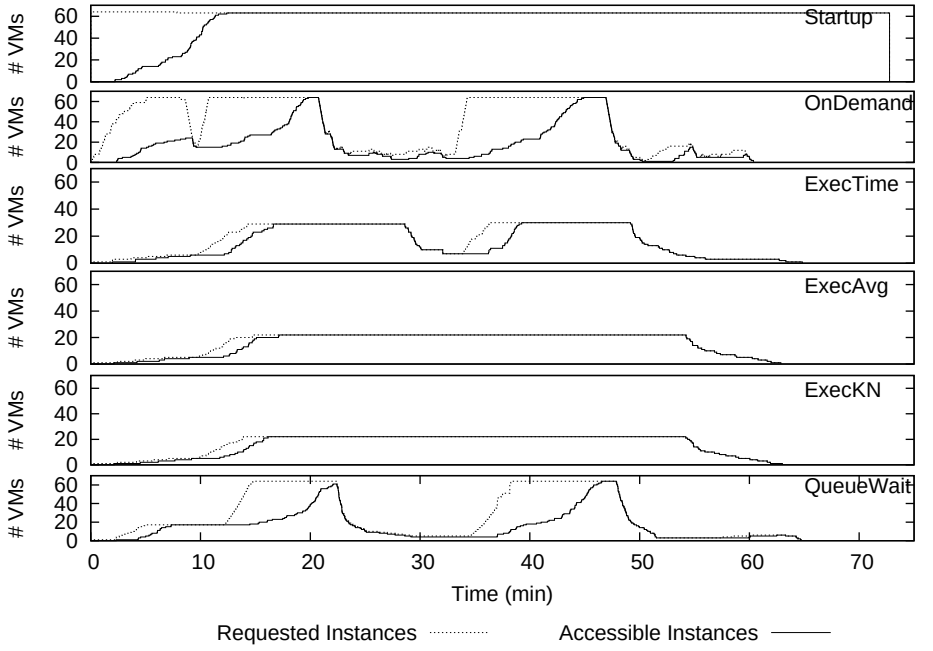

Figure 6.25: Instances over time for the provisioning policies with the *Bursty* workload on DAS4.

	ChargedCost			CostEfficiency			Utility		
	Uniform	Increasing	Bursty	Uniform	Increasing	Bursty	Uniform	Increasing	Bursty
Startup	128	128	127	1.7	1.7	1.8	1.0	1.0	1.0
ExecTime	77 (-40%)	52 (-59%)	52 (-59%)	1.6 (-3%)	1.8 (+5%)	2.8 (+61%)	1.4 (+37%)	2.0 (+105%)	2.3 (+127%)
ExecAvg	81 (-37%)	43 (-66%)	22 (-83%)	1.7 (+2%)	1.5 (-12%)	1.3 (-27%)	1.2 (+19%)	2.1 (+114%)	5.5 (+454%)
ExecKN	86 (-33%)	53 (-59%)	22 (-83%)	1.8 (+8%)	1.8 (+7%)	1.3 (-27%)	1.1 (+13%)	1.9 (+87%)	5.5 (+453%)
QueueWait	81 (-37%)	77 (-40%)	126 (-1%)	1.6 (-3%)	2.4 (+42%)	5.0 (+183%)	1.5 (+46%)	1.6 (+56%)	0.9 (-6%)

Table 6.1: Charged cost, cost efficiency and utility for policies on DAS4/Delft.

	ChargedCost			CostEfficiency			Utility		
	Uniform	Increasing	Bursty	Uniform	Increasing	Bursty	Uniform	Increasing	Bursty
Startup	14	14	14	1.9	2.0	2.0	1.0	1.0	1.0
ExecTime	7 (-50%)	7 (-50%)	6 (-57%)	1.4 (-23%)	2.4 (+22%)	2.2 (+9%)	1.9 (+91%)	1.7 (+66%)	2.2 (+123%)
ExecAvg	8 (-43%)	7 (-50%)	6 (-57%)	1.6 (-12%)	2.4 (+23%)	2.2 (+9%)	1.6 (+59%)	1.7 (+74%)	2.2 (+124%)
ExecKN	7 (-50%)	6 (-57%)	6 (-57%)	1.4 (-23%)	2.1 (+5%)	2.2 (+8%)	1.9 (+92%)	2.0 (+101%)	2.2 (+124%)
QueueWait	12 (-14%)	11 (-21%)	13 (-7%)	2.4 (+28%)	3.7 (+85%)	4.3 (+114%)	1.1 (+8%)	1.2 (+23%)	1.1 (+6%)

Table 6.2: Charged cost, cost efficiency and utility for policies on FIU.

	Charged Cost			Cost Efficiency			Utility		
Workload	Uniform	Increasing	Bursty	Uniform	Increasing	Bursty	Uniform	Increasing	Bursty
Startup	40	40	40	1.9	1.9	1.9	1.0	1.0	1.0
ExecTime	26 (-35%)	15 (-62%)	24 (-40%)	1.6 (-15%)	1.7 (-12%)	4.0 (+111%)	1.4 (+41%)	2.4 (+137%)	1.5 (+51%)
ExecAvg	34 (-15%)	17 (-57%)	26 (-35%)	2.2 (+17%)	1.9 (-1%)	4.3 (+126%)	1.0 (-1%)	2.2 (+116%)	1.4 (+40%)
ExecKN	30 (-25%)	16 (-60%)	25 (-38%)	2.0 (+7%)	1.8 (-5%)	4.1 (+115%)	1.2 (+20%)	2.3 (+133%)	1.5 (+45%)
QueueWait	29 (-28%)	18 (-55%)	38 (-5%)	2.0 (+3%)	2.0 (+4%)	6.2 (+227%)	1.3 (+29%)	2.1 (+113%)	1.0 (+2%)

Table 6.3: Charged cost, cost efficiency and utility for policies on EC2.

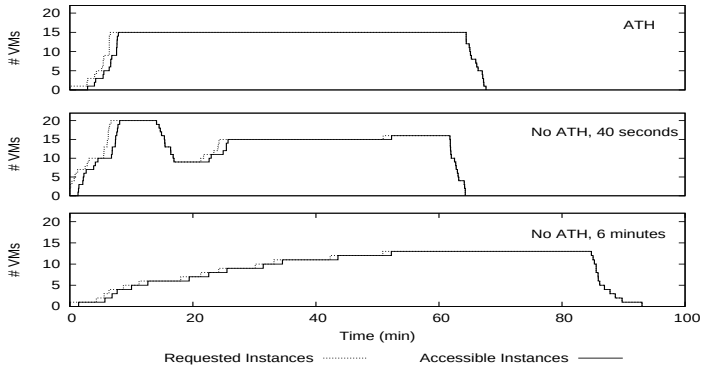

Figure 6.26: Instances over time for the ExecKN policy with and without the use of ATH, using the *Uniform* workload, on EC2.

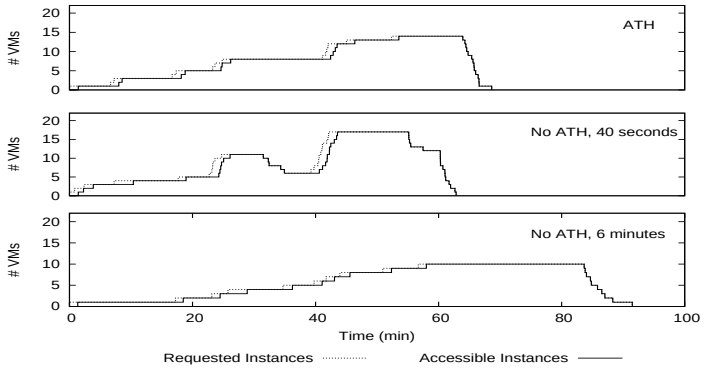

Figure 6.27: Instances over time for the ExecKN policy with and without the use of ATH, using the *Increasing* workload, on EC2 .

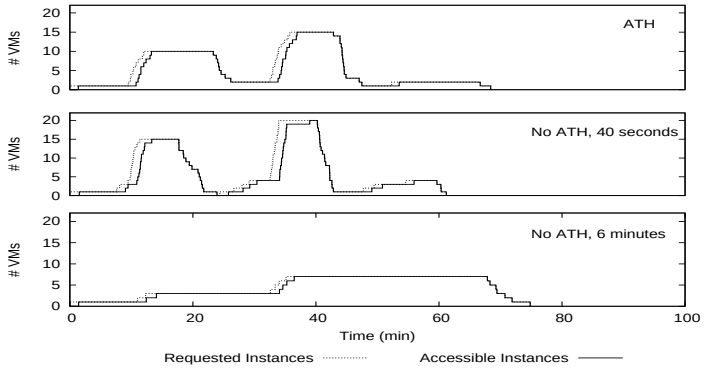

Figure 6.28: Instances over time for the ExecKN policy with and without the use of ATH, using the *Bursty* workload, on EC2 .

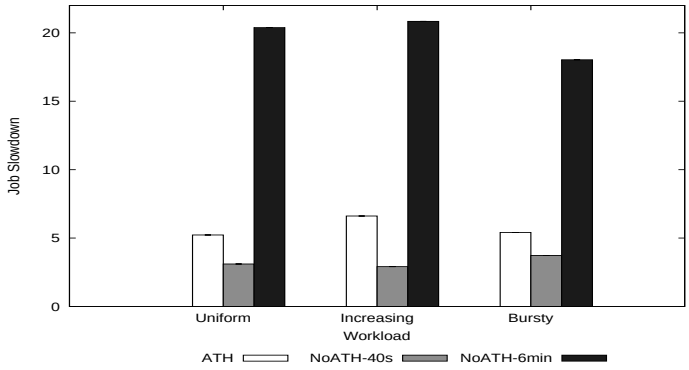

Figure 6.29: Job Slowdown (JSD) achieved with and without ATH, using ExecTime as the provisioning policy.

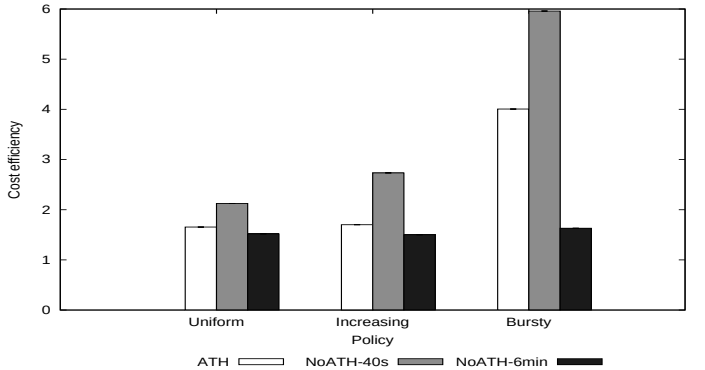

Figure 6.30: Cost efficiency (C_{eff}) achieved with and without ATH, using ExecTime as the provisioning policy.

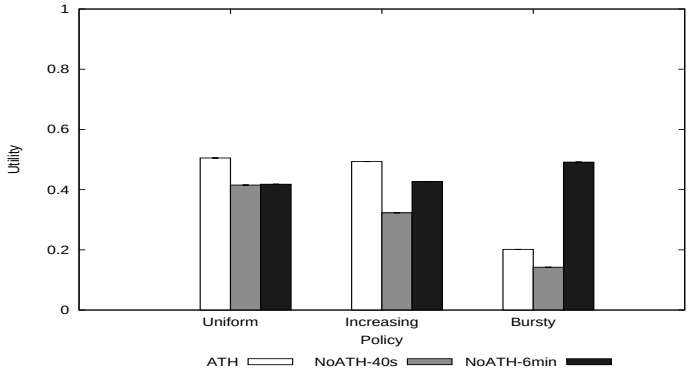

Figure 6.31: Utility (U) achieved with and without ATH, using ExecTime as the provisioning policy.

first scenario uses an overoptimistic prediction of the VM instance boot-up time, as it is configured to 40 seconds, about two times less than the observed resource acquisition times for Amazon EC2 [65]. Likewise, the second scenario uses a pessimistic prediction for the boot-up times, set to 6 minutes.

Figures 6.26-6.31 present our findings on the impact of ATH. On the one end, the 40-second boot-up time configuration is overeager to provide/release resources, while at the other end, the 6-minute configuration is more hesitant. This results in immense JSD (Figure 6.29), but better C_{eff} (Figure 6.30) and U (Figure 6.31) than the 40-second configuration. ATH achieves the best utility, except in the case of the bursty workload. However, if we also consider the achieved JSD, we can see that ATH achieves the best performance-cost trade-off. Using ATH, the provisioning policy is able to adapt to the cloud's performance.

Additionally, ATH is expected to improve utility in clouds with considerable resource acquisition time variability, even though it has not been tested under such circumstances. The policy should be more sparing when the cloud under-performs, and more performance-inclined when resource acquisition times are improved. Overall, for clouds with big resource acquisition time variability, it is expected that ATH will improve utility.

Impact of the Increase Factor (IF)

Here we evaluate the impact of IF, to the performance of the provisioning policies. As before, the provisioning policy is fixed to ExecKN, and IF variates between the Single, Multiple and Geometric schemes.

The acquisition and release of instances over time on EC2, while using the three increase factors, is presented in Figures 6.32-6.34. From these figures, it is observed that Geometric achieves the shortest makespan, followed by Multiple and with Single trailing. The reason behind this is because Geometric and Multiple respond faster to load variation. The job slowdown illustrated in Figure 6.35 shows that Geometric outperforms the other two schemes. However, when using variable-load workloads, Geometric is the least cost-efficient, and achieves the worst utility. Geometric provisions pro-actively, by acquiring a larger amount of resources than what is currently needed, keeping the system under lower load. Faster and more intense reaction to load variation leads to better performance, but also to resource under-utilization and lower performance-cost trade-offs.

6.2.2 Allocation

In this experiment we want to study the performance of different allocation policies and the static provisioning policy, Startup. Resources are acquired at the beginning of the experiment, and then jobs are sent to the system.

We use the FCFS, the SJF, and the FCFS-NW allocation policies in the three testbeds. Figure 6.38 lists the results only for the job slowdown metric, since we

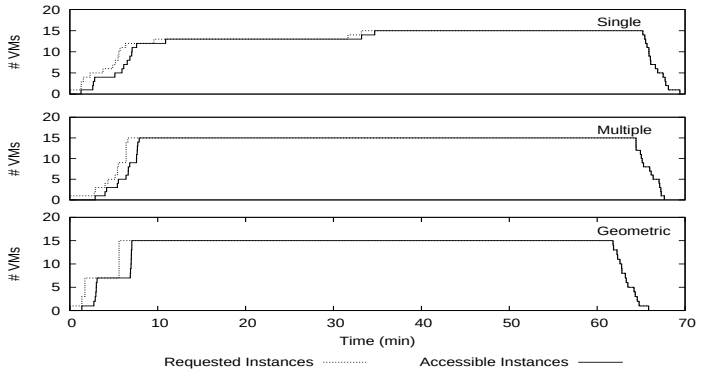

Figure 6.32: Instances over time for the `ExecKN` policy with variable `IF` with *Uniform* workload, on EC2.

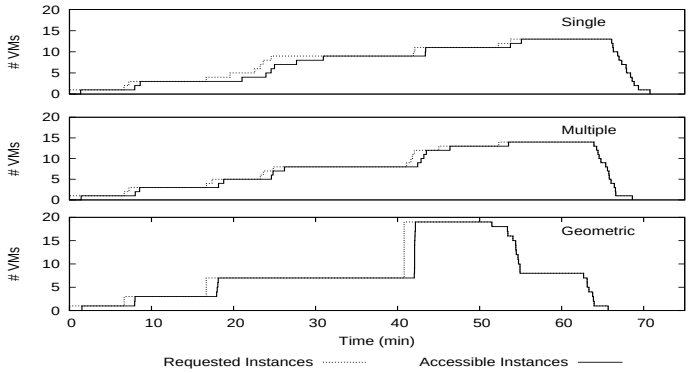

Figure 6.33: Instances over time for the `ExecKN` policy with variable `IF` with *Increasing* workload, on EC2.

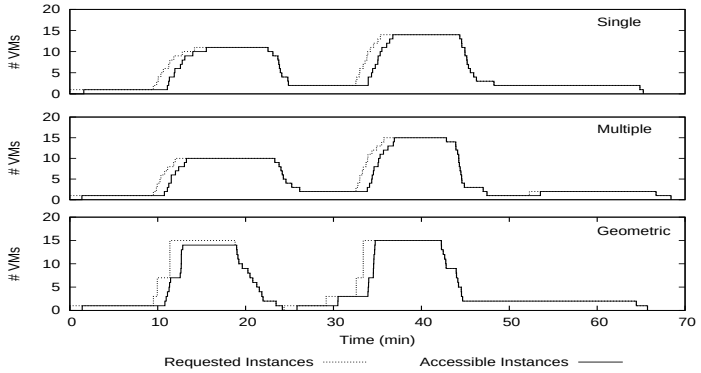

Figure 6.34: Instances over time for the `ExecKN` policy with variable `IF` with *Bursty* workload, on EC2.

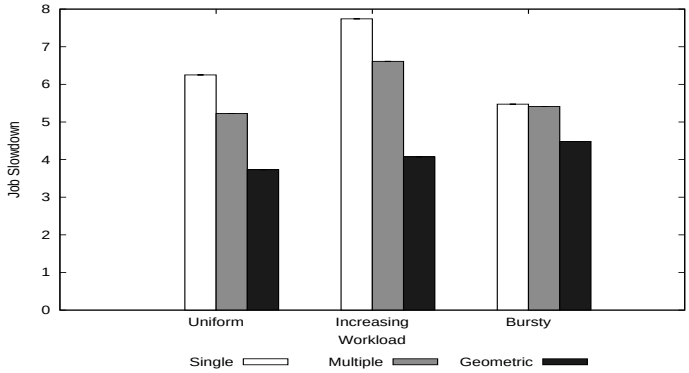

Figure 6.35: Job Slowdown (`JSD`) achieved with the three `IF` factors, using `ExecTime` as the provisioning policy.

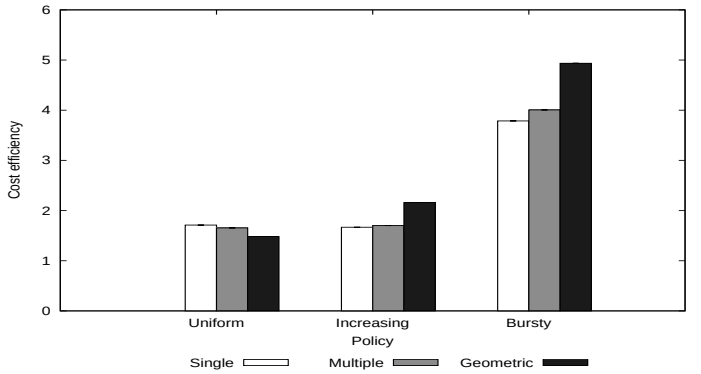

Figure 6.36: Cost efficiency (C_{eff}) achieved with the three `IF` factors, using `ExecTime` as the provisioning policy.

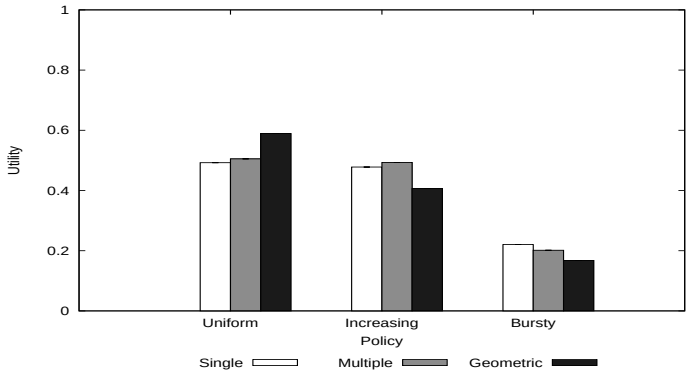

Figure 6.37: Utility (`U`) achieved with the three `IF` factors, using `ExecTime` as the provisioning policy.

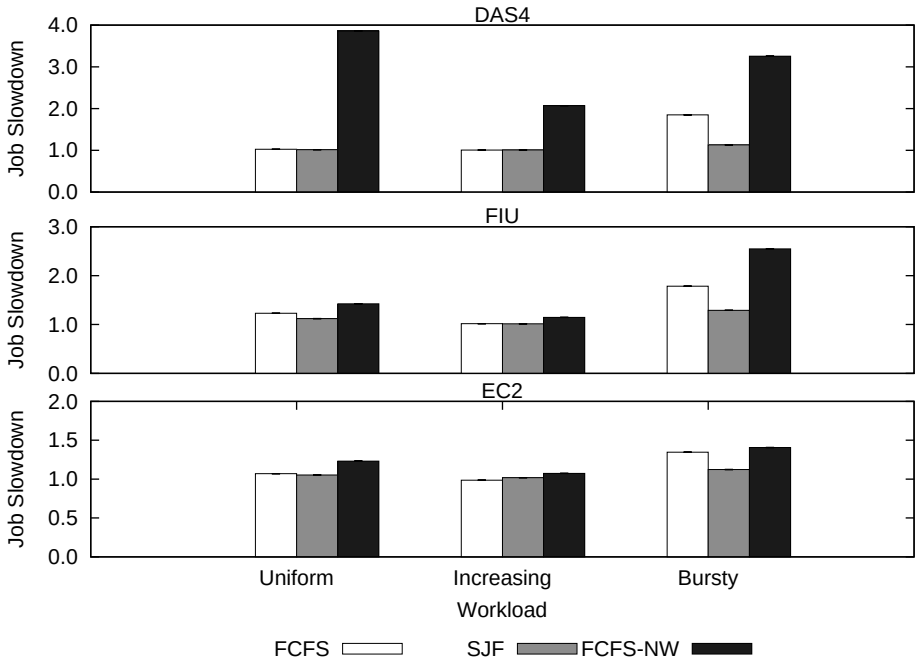

Figure 6.38: Average Job Slowdown for Allocation policies.

did not observe significant differences in cost or makespan. The experiment shows that `SJF` gives a lower slowdown, since shorter jobs are processed first, which means jobs in wait less time in the queue. Overall, `FCFS` performs similarly to SJF for the uniform and increasing workloads, however its performance degrades when under a bursty load. Lastly, the `FCFS-NW` policy, which assigns jobs to VMs with round-robin, creates resource competition, and thus has worse results in all experiments.

6.3 Impact of Workloads on Cloud Reliability

Large-scale experiments were planned and performed for the purposes of this work, on DAS4/VU and DAS4/Delft sites. However, we have experienced massive VM instantiation failures. This situation effectively made the results unusable from a performance-analysis perspective. A large portion of the requested VM instances were unexplainably failing during the pending state.

Further investigation showed that the failures were caused by the CPU-intensive workloads submitted to the cloud. As can be observed in Figures 6.39 and 6.40, when the VM instances were requested prior to the workload submission using the `Startup` provisioning policy, no instantiation failures took place. However, when using a dynamic policy, such as `ExecAvg`, few of the requested instances reached the “running” state. This can be observed from the inability of requested

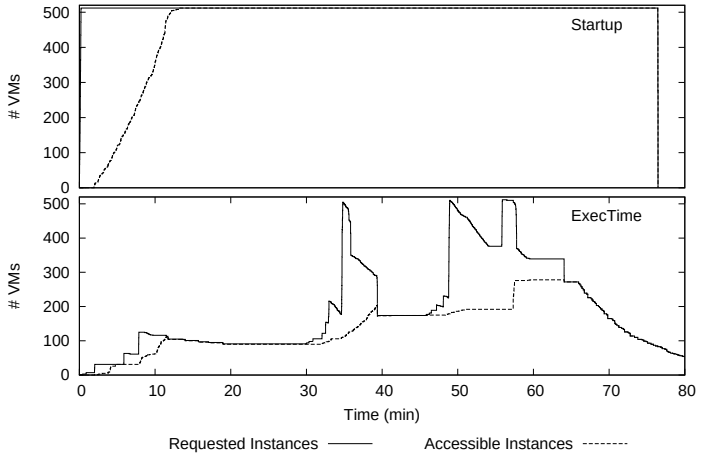

Figure 6.39: Observed failures on DAS4/VU, while using a `CPU-intensive, Increasing` workload. The failures only take place with dynamic policies, such as `ExecTime`.

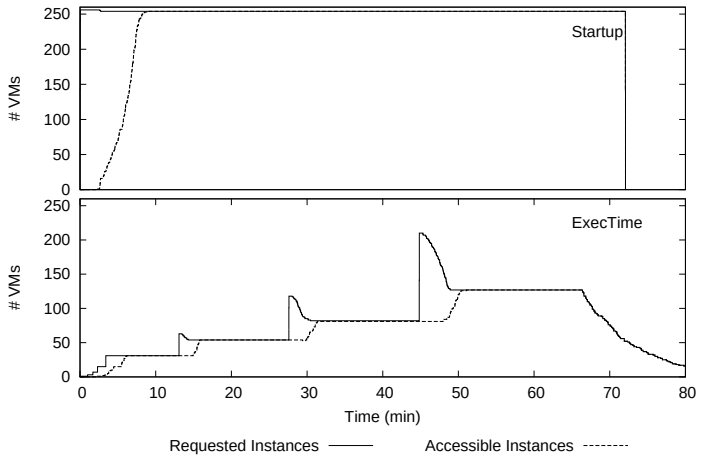

Figure 6.40: Observed failures on DAS4/Delft, while using a `CPU-intensive, Increasing` workload.

instances to become accessible instances. For example, in Figure 6.39, `ExecTime` starts requesting more instances at minute 30. At about minute 35, `ExecTime` has already requested almost 400 new instances. Two minutes later, however, a massive amount of instantiation failures take place, indicated by the sudden drop in requested instances. What we eventually manage to acquire between minutes 30 and 40, is around 100 instances, only 25% of what was actually requested.

The difference between the static and dynamic provisioning scenarios was that for the dynamic case, there were already several VMs running on each physical node, each of which with scheduled CPU-intensive jobs.

While we were able to pinpoint the CPU-intensive workloads as the origin of the failures, we could not identify why they have this influence on cloud reliability. We were only able to trace the failures down to the virtualization layer, where there seems to be some mis-configuration of the current DAS4 hypervisor (KVM).

6.4 Success Stories

Apart from the issue described in the previous section, several other problems regarding the OpenNebula VIM installation on the DAS4 supercomputer were identified with the use of SkyMark and were addressed afterwards. In other words, SkyMark has already proved its usefulness in testing IaaS cloud deployments, a goal for which it was not designed, but for which no other public tools exist.

Firstly, we noticed that OpenNebula required a considerable amount of time to respond to acquire/release requests and VM status updates, especially when the cloud was under heavy load. A migration from an Sqlite backend to a MySQL

backend, helped in alleviating this performance issue.

Another issue we encountered was with the OpenNebula default image transfer configuration, which was based on the use of the underlying Network File System (NFS) (See Subsection 2.3.5). When we tried to deploy large scale experiments on DAS4/Delft, the load exercised on the supercomputer network was so large, that DAS4 collapsed and had to be hardware-rebooted. This issue did not appear in the study of Ueda et al. [79], where they examined the impact of this configuration to the cloud performance. We then had to use the localDisk transfer method with caching, in which the VM images are explicitly transferred to the local file systems of the nodes. All subsequent instantiation requests on a specific node, can be served using the cached image. A shared filesystem is currently needed for performing live migration of instances, but it comes with a great network performance penalty.

An additional performance issue was the considerable staging time for instances using eager allocation, especially when tens of instances are provisioned and are currently in the staging phase. The boot-up time grew significantly with the number of instances requested, something that would have a severe effect on the allocation and provisioning policy results. For this reason, we adopted the use of lazy image allocation, that would reduce the time needed to prepare an instance.

Finally, through the use of SkyMark, we discovered a bug in the XML-RPC interface of OpenNebula. More specifically, the interface did not inform SkyMark on the event of an instance failure. The bug has been reported [59], but we had to switch to an "internal-state" attribute, that successfully informed us of instance failure events.

To conclude, we acknowledge that setting-up a reliable, scalable and elastic cloud on private infrastructure might be more of a challenge than initially regarded. Moreover, we emphasize on the necessity of using performance evaluation frameworks, such as SkyMark, to certify an IaaS deployment.

Chapter 7

Conclusions and Future Work

To bring this thesis to a close, we present our conclusions in Section 7.1 and potential future work in Section 7.2.

7.1 Conclusions

We provide our conclusions for each of the two thesis research components individually: first, the study on the performance of IaaS clouds, and subsequently the study on the provisioning and allocation policies. Lastly, we discuss our findings regarding cloud reliability.

7.1.1 IaaS performance evaluation

The performance characteristics of IaaS clouds need to be understood better. Towards providing insight for this problem, we have developed SkyMark, a performance analysis framework for IaaS environments. We used SkyMark to firstly generate a set of complex, synthetic workloads that stress one or two components of the cloud, and exhibit three different arrival patterns. Using an elaborate, but mostly automated experimental methodology, we were able to isolate the overheads imposed by the software stack and the resource time sharing.

Our main findings for this section are listed below:

1. The virtualization cost is small for CPU-intensive workloads ($\approx$ 3%) and Memory-intensive workloads ($\approx$ 5%), but is significantly more for I/O-intensive workloads ($\approx$ 40%) and Memory+ I/O workload mixtures ($\approx$ 20%). These results are in accordance with the virtualization overheads when they were studied individually in a virtualized, but non-cloud environment [11, 16, 17, 28, 56, 93].

2. The contention between VMs imposes a small overhead for CPU and Memory-intensive workloads (up to $\approx$ 4% and $\approx$ 7% respectively). The performance

isolation across VMs is considerably worse in the case of the I/O and Memory+I/O intensive workloads ($\approx$ 100% and $\approx$ 40% respectively). These findings suggest that it is paramount to focus on providing VM performance isolation for disk access.

3. The performance variation in response to the imposed load is significant, especially for I/O-intensive workloads. This variation in performance could make it especially hard to provide any performance QoS guarantees to cloud users. Currently, public IaaS providers only make availability guarantees.

4. SkyMark imposed negligible overhead to the workload execution. However, the decision to place the experimentation node within the cloud caused some performance interference in the case of Memory-Intensive Bursty workloads. Although we could have used different configurations, they would have been less considerate of other DAS4 users.

5. Even though we conducted the experiments using one cloud configuration, we previously considered several configurations. The performance variates considerably with the current cloud platform configuration, a conclusion also drawn by [79].

7.1.2 Policy evaluation

Current and potential IaaS users need to be provided with deeper insights for the achieved performance and charged cost that their workloads would incur, with a selected set of provisioning and allocation policies. Without a deeper understanding of the performance and cost that the used policies can achieve, potential IaaS users, and the industry in general, would hesitate to migrate to the cloud.

In this work we have conducted a comprehensive, empirical study of six provisioning and three allocation policies, using SkyMark, and a subset of our complex workloads. We performed experimentation on three IaaS clouds, two of which were set up on private infrastructure using open-source cloud manager implementations, and the other was Amazon's EC2 public cloud.

Here, we list our main findings regarding the policy investigation:

1. `OD-ExecTime` and its two variants (especially `OD-ExecKN`), are a good performance-cost trade-off among the investigated provisioning policies; Unlike the `OD-ExecTime` policy, `OD-ExecKN` does not assume known job run-times.

2. `Startup`, the only static policy we used, delivers stable performance, but incurs up to 5 times higher cost. The efficiency of this static policy is lower for workloads

3. Our Adaptive Threshold Heuristic (`ATH`) is able to automatically adjust the provisioning policy's behaviour to the cloud's current provisioning perfor-

mance, thus improving the policy's performance-cost trade-off. An additional benefit of using ATH is that configuring the policy does not require to know the cloud's current acquisition times.

4. Regarding IF, faster reaction to load variation leads to better performance, but also to a worse performance-cost trade-off. Overall, we find that the Multiple increase factor (IF) achieves the best performance-cost trade-off.

5. The SJF allocation policy achieves the best performance among the examined policies, however, it makes the assumption that the job run-times are known.

7.1.3 Cloud reliability

We found that it is difficult to set-up a reliable and well-performing private cloud on our local infrastructure, even though DAS4 is equipped with state-of-the-art commercial hardware. When we configured OpenNebula to use its NFS image transfer method during our large-scale experiments, the supercomputer crashed because of network overload. When using the eager image allocation method, the extremely long provisioning times when tens of VMs were provisioned, were considered to be obtrusive to performing our experiments. An issue whose cause we were not able to identify, was the high VM instantiation failure ratio under the localDisk configuration with caching, when the physical nodes were already hosting VMs with our workloads. SkyMark was vital in discovering these reliability issues of the used IaaS environments, even though it was not originally designed for this purpose.

Virtual Machine instantiation failures were not exactly rare even for the Amazon EC2 cloud, as previously reported in [42]. This was usually the case when we requested tens of VMs simultaneously. Occasionally, one or a couple of them would fail to boot.

7.2 Future Work

The SkyMark framework is currently going through the certification process of the Standards Performance Evaluation Corporation (SPEC) [76]. Apart from the certification process, we plan to extend this work to consider new provisioning and allocation policies that adapt to changing workload, evolving resources, and complex Service Level Agreements. We will work on creating a taxonomy of provisioning and allocation policies, and examine classes of policies that we have not previously explored.

Another research direction is to consider more diverse and realistic workloads, such as synthetic Bag-of-Tasks (BoT) based workloads generated with the model in

[39], mixtures of synthetic parallel workloads and synthetic BoT-based workloads, or real traces from grid workloads [37].

In this work, we have used mainly traditional metrics to evaluate and compare the performance of clouds and policies. In the future, we plan to investigate new cloud-oriented metrics that can quantify some of the major cloud characteristics, such as elasticity and scalability.

Bibliography

[1] ElasticHosts. Online, 2011. http://www.elastichosts.com.

[2] GoGrid. Online, 2011. http://www.gogrid.com/.

[3] Amazon. Amazon EC2 Instance Types. Online, 2011. http://aws.amazon.com/ec2/instance-types.

[4] Amazon. Amazon EC2 Pricing. Online, 2011. http://aws.amazon.com/ec2/pricing/.

[5] Amazon Web Services. Auto Scaling. Online, 2011. http://aws.amazon.com/autoscaling/.

[6] Amazon Web Services. Elastic Load Balancing. Online, 2011. http://aws.amazon.com/elasticloadbalancing/.

[7] Amazon Web Services LLC. Amazon Web Services Scaling. Online, 2011. http://aws.amazon.com.

[8] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, and Matei Zaharia. Above the clouds: A berkeley view of cloud computing. Technical report, UC Berkeley Reliable Adaptive Distributed Systems Laboratory, 2009.

[9] Remzi H. Arpaci-Dusseau, Andrea C. Arpaci-Dusseau, Amin Vahdat, Lok T. Liu, Thomas E. Anderson, and David A. Patterson. The interaction of parallel and sequential workloads on a network of workstations. In *SIGMETRICS*, pages 267–278, 1995.

[10] Immaneni Ashok and John Zahorjan. Scheduling a Mixed Interactive and Batch Workload on a Parallel, Shared Memory Supercomputer. In *SC*, pages 616–625, 1992.

[11] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Timothy L. Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP*, pages 164–177, 2003.

[12] Orna Agmon Ben-Yehuda, Assaf Schuster, Artyom Sharov, Mark Silberstein, and Alexandru Iosup. ExPERT: Pareto-efficient task replication on grids and clouds. Technical Report CS-2011-03, Technion CS Dept., Apr 2011.

[13] Carsten Binnig, Donald Kossmann, Tim Kraska, and Simon Loesing. How is the weather tomorrow?: towards a benchmark for the cloud. In *DBTest*, 2009.

[14] Eun-Kyu Byun, Yang-Suk Kee, Jin-Soo Kim, and Seungryoul Maeng. Cost optimized provisioning of elastic resources for application workflows. *Future Gener. Comput. Syst.*, 27:1011–1026, October 2011.

[15] David Candeia, Ricardo Araujo, Raquel Vigolvino Lopes, and Francisco Vilar Brasileiro. Investigating business-driven cloudburst schedulers for e-science bag-of-tasks applications. In *CloudCom*, pages 343–350, 2010.

[16] Ludmila Cherkasova and Rob Gardner. Measuring cpu overhead for i/o processing in the xen virtual machine monitor. In *USENIX Annual Technical Conference, General Track*, pages 387–390, 2005.

[17] Bryan Clark, Todd Deshane, Eli Dow, Stephen Evanchik, Matthew Finlayson, Jason Herne, and Jeanna Neefe Matthews. Xen and the art of repeated research. In *USENIX Annual Technical Conference, FREENIX Track*, pages 135–144, 2004.

[18] CloudHarmony.com^{TM}. Cloudharmony services. Online, January 2011. http://cloudharmony.com/services/.

[19] Marcos Dias de Assunção, Alexandre di Costanzo, and Rajkumar Buyya. Evaluating the cost-benefit of using cloud computing to extend the capacity of clusters. In *HPDC*, pages 141–150, 2009.

[20] Ewa Deelman, Gurmeet Singh, Miron Livny, G. Bruce Berriman, and John Good. The cost of doing science on the cloud: the montage example. In *SC*, page 50, 2008.

[21] Mohamed El-Refaey. *Virtual Machines Provisioning and Migration Services*, pages 121–156. John Wiley & Sons, Inc., 2011.

[22] Open Grid Forum. Job Submission Description Language (JSDL) specification, Version 1.0. Online, 2005. http://www.gridforum.org/documents/GFD.56.pdf.

[23] Stéphane Genaud and Julien Gossa. Cost-wait trade-offs in client-side resource provisioning with elastic clouds. In *IEEE CLOUD*, pages 1–8, 2011.

[24] GoGrid. F5 Hardware Load Balancers. Online, 2011. http://www.gogrid.com/cloud-hosting/load-balancers.php.

[25] Google. Google AppEngine. Online, 2011. http://code.google.com/appengine/.

[26] Albert Greenberg, James Hamilton, David A. Maltz, and Parveen Patel. The cost of a cloud: research problems in data center networks. *SIGCOMM Comput. Commun. Rev.*, 39:68–73, Dec. 2008.

[27] Albert G. Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. Vl2: a scalable and flexible data center network. In *SIGCOMM*, pages 51–62, 2009.

[28] Martin Grund, Jan Schaffner, Jens Krüger, Jan Brunnert, and Alexander Zeier. The effects of virtualization on main memory systems. In *DaMoN*, pages 41–46, 2010.

[29] Chuanxiong Guo, Haitao Wu, Kun Tan, Lei Shi, Yongguang Zhang, and Songwu Lu. Dcell: a scalable and fault-tolerant network structure for data centers. In *SIGCOMM*, pages 75–86, 2008.

[30] Thomas J. Hacker and Kanak Mahadik. Flexible resource allocation for reliable virtual cluster computing systems. In *SC*, page 48, 2011.

[31] Thomas A. Henzinger, Anmol V. Singh, Vasu Singh, Thomas Wies, and Damien Zufferey. Flexprice: Flexible provisioning of resources in a cloud environment. In *IEEE CLOUD*, pages 83–90, 2010.

[32] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: a platform for fine-grained resource sharing in the data center. In *NSDI*, pages 22–22, Berkeley, CA, USA, 2011. USENIX Association.

[33] Citrix Systems Inc. Citrix XenServer. Online, 2011. http://www.citrix.com/English/ps2/products/product.asp?contentID=683148.

[34] Alexandru Iosup and Dick H. J. Epema. Grenchmark: A framework for analyzing, testing, and comparing grids. In *CCGRID*, 2006.

[35] Alexandru Iosup and Dick H. J. Epema. Grid computing workloads. *IEEE Internet Computing*, 15(2):19–26, 2011.

[36] Alexandru Iosup, Dick H. J. Epema, Carsten Franke, Alexander Papaspyrou, Lars Schley, Baiyi Song, and Ramin Yahyapour. On grid performance evaluation using synthetic workloads. In *JSSPP*, pages 232–255, 2006.

[37] Alexandru Iosup, Hui Li, Mathieu Jan, Shanny Anoep, Catalin Dumitrescu, Lex Wolters, and Dick H. J. Epema. The grid workloads archive. *Future Gener. Comput. Syst.*, 24:672–686, July 2008.

[38] Alexandru Iosup, Simon Ostermann, Nezih Yigitbasi, Radu Prodan, Thomas Fahringer, and Dick H. J. Epema. Performance analysis of cloud computing services for many-tasks scientific computing. *IEEE Trans. Parallel Distrib. Syst.*, 22(6):931–945, 2011.

[39] Alexandru Iosup, Omer Ozan Sonmez, Shanny Anoep, and Dick H. J. Epema. The performance of bags-of-tasks in large-scale distributed systems. In *HPDC*, pages 97–108, 2008.

[40] Alexandru Iosup, Nezih Yigitbasi, and Dick Epema. On the performance variability of production cloud services. In *CCGrid*, pages 104 –113, may 2011.

[41] Michael A. Iverson, Füsun Özgüner, and Gregory J. Follen. Run-time statistical estimation of task execution times for heterogeneous distributed computing. In *HPDC*, pages 263–, 1996.

[42] Keith R. Jackson, Krishna Muriki, Lavanya Ramakrishnan, Karl J. Runge, and Rollin C. Thomas. Performance and cost analysis of the supernova factory on the amazon aws cloud. *Scientific Programming*, 19(2-3):107–119, 2011.

[43] Keith R. Jackson, Lavanya Ramakrishnan, Krishna Muriki, Shane Canon, Shreyas Cholia, John Shalf, Harvey J. Wasserman, and Nicholas J. Wright. Performance analysis of high performance computing applications on the amazon web services cloud. In *CloudCom*, pages 159–168, 2010.

[44] Raj Jain. *The Art of Computer Systems Performance Analysis: techniques for experimental design, measurement, simulation, and modeling*. Wiley, 1991.

[45] Dejun Jiang, Guillaume Pierre, and Chi-Hung Chi. Ec2 performance analysis for resource provisioning of service-oriented applications. In *ICSOC/ServiceWave Workshops*, pages 197–207, 2009.

[46] Ekasit Kijsipongse and Sornthep Vannarat. Autonomic resource provisioning in rocks clusters using eucalyptus cloud computing. In *MEDES*, pages 61–66, 2010.

[47] Tom Killalea. Meet the virts. *Queue*, 6:14–18, January 2008.

[48] Avi Kivity. kvm: the Linux virtual machine monitor. In *OLS '07: The 2007 Ottawa Linux Symposium*, pages 225–230, July 2007.

[49] Ang Li, Xiaowei Yang, Srikanth Kandula, and Ming Zhang. Cloudcmp: comparing public cloud providers. In *Internet Measurement Conference*, pages 1–14, 2010.

[50] David J. Lilja. *Measuring Computer Performance: A Practitioner's Guide*. Cambridge University Press, 2005.

[51] Ignacio M. Llorente, Rubn S. Montero, Borja Sotomayor, David Breitgand, Alessandro Maraschini, Eliezer Levy, and Benny Rochwerger. *On the Management of Virtual Machines for Cloud Infrastructures*, pages 157–191. John Wiley & Sons, Inc., 2011.

[52] Wei Lu, Jared Jackson, Jaliya Ekanayake, Roger S. Barga, and Nelson Araujo. Performing large science experiments on azure: Pitfalls and solutions. In *CloudCom*, pages 209–217, 2010.

[53] Ming Mao, Jie Li, and Marty Humphrey. Cloud auto-scaling with deadline and budget constraints. In *GRID*, pages 41–48, 2010.

[54] Paul Marshall, Kate Keahey, and Timothy Freeman. Elastic site: Using clouds to elastically extend site resources. In *CCGRID*, pages 43–52, 2010.

[55] Peter Mell and Timothy Grance. The NIST Definition of Cloud Computing (Draft). *National Institute of Standards and Technology*, page 7, Jan. 2011.

[56] Aravind Menon, Jose Renato Santos, Yoshio Turner, G. John Janakiraman, and Willy Zwaenepoel. Diagnosing performance overheads in the xen virtual machine environment. In *VEE*, pages 13–23, 2005.

[57] Umar Farooq Minhas, Jitendra Yadav, Ashraf Aboulnaga, and Kenneth Salem. Database systems on virtual machines: How much do you lose? In *ICDE Workshops*, pages 35–41, 2008.

[58] Michael A. Murphy, Brandon Kagey, Michael Fenn, and Sebastien Goasguen. Dynamic provisioning of virtual organization clusters. In *CCGRID*, pages 364–371, 2009.

[59] Nassos Antoniou. Opennebula bug 1072. Online, 2011. http://dev.opennebula.org/issues/1072.

[60] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. Q-clouds: managing performance interference effects for qos-aware clouds. In *EuroSys*, pages 237–250, 2010.

[61] Daniel Nurmi, Rich Wolski, Chris Grzegorczyk, Graziano Obertelli, Sunil Soman, Lamia Youseff, and Dmitrii Zagorodnov. The eucalyptus open-source cloud-computing system. In *CCGRID*, pages 124–131, Washington, DC, USA, 2009. IEEE Computer Society.

[62] Open Grid Forum. Open Cloud Computing Interface (OCCI). Online, 2011. http://occi-wg.org/.

[63] OpenNebula. OpenNebula open-source toolkit for cloud-computing. Online, 2011. http://www.opennebula.org.

[64] OpenNebula Project Leads. Xml-rpc api 3.0. Online, 2011. http://opennebula.org/documentation:rel3.0:api.

[65] Simon Ostermann, Alexandru Iosup, Nezih Yigitbasi, Radu Prodan, Thomas Fahringer, and Dick H. J. Epema. A performance analysis of ec2 cloud computing services for scientific computing. In *CloudComp*, pages 115–131, 2009.

[66] Simon Ostermann, Radu Prodan, and Thomas Fahringer. Resource management for hybrid grid and cloud computing. In *Cloud Computing*, volume 0 of *Computer Communications and Networks*, pages 179–194. Springer London, 2010.

[67] Mayur R. Palankar, Adriana Iamnitchi, Matei Ripeanu, and Simson Garfinkel. Amazon s3 for science grids: a viable solution? In *DADC*, pages 55–64, 2008.

[68] Xing Pu, Ling Liu, Yiduo Mei, Sankaran Sivathanu, Younggyun Koh, and Calton Pu. Understanding performance interference of i/o workload in virtualized cloud environments. In *IEEE CLOUD*, pages 51–58, 2010.

[69] Andres Quiroz, Hyunjoo Kim, Manish Parashar, Nathan Gnanasambandam, and Naveen Sharma. Towards autonomic workload provisioning for enterprise grids and clouds. In *GRID*, pages 50–57, 2009.

[70] John J. Rehr, Fernando D. Vila, Jeffrey P. Gardner, Lucas Svec, and Micah Prange. Scientific computing in the cloud. *Computing in Science and Engineering*, 12(3):34–43, 2010.

[71] Mohsen Salehi and Rajkumar Buyya. Adapting market-oriented scheduling policies for cloud computing. In *ICA3PP*. 2010.

[72] Mohsen Amini Salehi and Rajkumar Buyya. Adapting market-oriented scheduling policies for cloud computing. In *ICA3PP (1)*, pages 351–362, 2010.

[73] Salesfoce. CRM software on-demand. Online, 2011. http://www.salesforce.com/.

[74] Love H. Seawright and Richard A. MacKinnon. Vm/370 - a study of multiplicity and usefulness. *IBM Systems Journal*, 18(1):4–17, 1979.

[75] Borja Sotomayor, Rubén S. Montero, Ignacio M. Llorente, and Ian Foster. Virtual infrastructure management in private and hybrid clouds. *IEEE Internet Computing*, 13:14–22, September 2009.

[76] Standard Performance Evaluation Corporation. SPEC's Structure. Online, 2011. http://www.spec.org/spec/.

[77] Todd Tannenbaum, Derek Wright, Karen Miller, and Miron Livny. Beowulf cluster computing with linux. chapter Condor: a distributed job scheduler, pages 307–350. MIT Press, Cambridge, MA, USA, 2002.

[78] Juan M. Tirado, Daniel Higuero, Florin Isaila, and Jesús Carretero. Predictive data grouping and placement for cloud-based elastic server infrastructures. In *CCGRID*, pages 285–294, 2011.

[79] Yohei Ueda and Toshio Nakatani. Performance variations of two open-source cloud platforms. In *IISWC*, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[80] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L. Santoni, Fernando C. M. Martins, Andrew V. Anderson, Steven M. Bennett, Alain Kagi, Felix H. Leung, and Larry Smith. Intel virtualization technology. *Computer*, 38:48–56, May 2005.

[81] David Villegas, Athanasios Antoniou, Seyed Masoud Sadjadi, and Alexandru Iosup. An analysis of provisioning and allocation policies for infrastructure-as-a-service clouds: Extended results. Technical report, Delft University of Technology, 2011.

[82] VMware. VMware ESXi and ESX Info Center. Online, 2011. http://www.vmware.com/products/vsphere/esxi-and-esx/overview.html.

[83] VMware. VMware products. Online, 2011. http://www.vmware.com/products.

[84] William Voorsluys, James Broberg, and Rajkumar Buyya. *Introduction to Cloud Computing*, chapter 1, pages 1–41. Cloud Computing: Principles and Paradigms. John Wiley & Sons, Inc., 2011.

[85] Vrije Universiteit Amsterdam. The Distributed ASCI Supercomputer 4. Online, 2011. http://www.cs.vu.nl/das4/.

[86] Edward Walker. Benchmarking Amazon EC2 for high-performance scientific computing. *LOGIN*, 33(5):18–23, Oct. 2008.

[87] Chuliang Weng, Qian Liu, Lei Yu, and Minglu Li. Dynamic adaptive scheduling for virtual machines. In *HPDC*, pages 239–250, 2011.

[88] Andrew Whitaker, Marianne Shaw, and S D Gribble. Denali : Lightweight virtual machines for distributed and networked applications. *Technical Report*, 02(Figure 1):10, 2002.

[89] WikiBooks. QEMU/Images. Online, 2011. http://en.wikibooks.org/wiki/QEMU/Images.

[90] Linlin Wu, Saurabh Kumar Garg, and Rajkumar Buyya. Sla-based resource allocation for software as a service provider (saas) in cloud computing environments. In *CCGRID*, pages 195–204, 2011.

[91] Nezih Yigitbasi, Alexandru Iosup, Dick H. J. Epema, and Simon Ostermann. C-meter: A framework for performance analysis of computing clouds. In *CCGRID*, pages 472–477, 2009.

[92] Lamia Youseff, Rich Wolski, Brent Gorda, and Ra Krintz. Paravirtualization for hpc systems. In *In Proc. Workshop on Xen in High-Performance Cluster and Grid Computing*, pages 474–486. Springer, 2006.

[93] Weikuan Yu and Jeffrey S. Vetter. Xen-based hpc: A parallel i/o perspective. In *CCGRID*, pages 154–161, 2008.

[94] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys*, pages 265–278, New York, NY, USA, 2010. ACM.