Model Driven Scheduling for Virtualized Workloads

Proefschrift voorgelegd op 28 juni 2013 tot het behalen van de graad van Doctor in de Wetenschappen – Informatica, bij de faculteit Wetenschappen, aan de Universiteit Antwerpen.

PROMOTOREN:
prof. dr. Jan Broeckhove
dr. Kurt Vanmechelen

Sam Verboven

RESEARCH GROUP COMPUTATIONAL
MODELING AND PROGRAMMING

Dankwoord

Het behalen van een doctoraat is een opdracht die zonder hulp onmogelijk tot een goed einde kan gebracht worden. Gelukkig heb ik de voorbije jaren de kans gekregen om samen te werken met talrijke stimulerende collega's. Stuk voor stuk hebben zij bijgedragen tot mijn professionele en persoonlijke ontwikkeling. Hun geduldige hulp en steun was essentieel bij het overwinnen van de vele uitdagingen die met een doctoraat gepaard gaan. Graag zou ik hier dan ook enkele woorden van dank neerschrijven.

Allereerst zou ik graag prof. dr. Jan Broeckhove en em. prof. dr. Frans Arickx bedanken om mij de kans te geven een gevarieerde en boeiend academisch traject te starten. Bij het beginnen van mijn doctoraat heeft Frans mij niet alleen geholpen om een capabel onderzoeker te worden, ook bij het lesgeven heeft hij mij steeds met raad en daad bijgestaan. Na het emiraat van Frans heeft Jan deze begeleiding overgenomen en er voor gezorgd dat ik het begonnen traject ook succesvol kon beeïndigen. Beiden hebben ze mij steeds grote vrijheid gegeven in mijn zoektocht om interessante onderzoeksvragen te identificeren en beantwoorden.

Vervolgens zou ik graag dr. Peter Hellinckx en dr. Kurt Vanmechelen bedanken voor hun persoonlijke en vaak intensieve begeleiding. Zelfs voor de start van mijn academische carrière, bij het schrijven van mijn Master thesis, heeft Peter mij klaargestoomd voor een vlotte start als onderzoeker. Naast het onderzoek heeft hij ook bij het lesgeven steeds een belangrijke en positieve invloed gehad. Na het vertrek van Peter is Kurt een steeds belangrijkere rol gaan spelen in de begeleiding van mijn onderzoek. Een belangrijk deel van mijn resultaten zijn dan ook te danken aan zijn ervaring, inbreng, en bereidheid om zich in te werken in mijn onderzoeksdomein.

Even essentieel is het ook om de collega's te bedanken die soms minder rechtstreeks bij mijn onderzoek betrokken zijn geweest. Ook zij hebben vaak waardevolle nieuwe inzichten aangebracht of technische hulp geboden. In het bijzonder wil ik Ruben Van den Bossche bedanken om mij bij te staan op meer manieren dan ik had mogen verwachten. Hij stond steeds klaar om een frisse kijk te geven op de problemen die vaak lang konden aanslepen. Na vijf jaar een bureau en frustraties te delen is hij dan ook volledig ingewijd in de details van mijn onderzoeksdomein. Ook wil ik graag Delphine Draelants, Przemyslaw Klosiewicz, Silas De Munck, Sean Stijven, Nils De Moor en Wim Depoorter bedanken. Zonder hen zouden de voorbij jaren aan de UA ongetwijfeld minder aangenaam en productief geweest zijn.

Als laatste wil ik mijn familie en vrienden bedanken. Gedurende vele jaren hebben ze mij geholpen om op de juiste momenten te ontspannen en te relativeren. Ook in de jaren voor mijn doctoraat hebben ze bijgedragen bij het traject dat mij in staat gesteld heeft dit eindresultaat te bereiken. Bedankt iedereen!

Sam
juni 2013

Contents

CHAPTER 1

Introduction

The technological advances of the previous decades have been a catalyst for a rapid increase of digitization in all facets of society. Not too long ago, digitally storing and processing even small amounts of information was reserved for expensive research projects and large multinationals. Rising processing and storage requirements initially led to the creation of increasingly powerful *supercomputers*. An expensive resource used simultaneously by many different users for diverse workloads. These multi-user systems were the first to require *job scheduling* [**?**] to achieve high utilization and ensure fairness. Further advances and economies of scale led to the rise of *workstations*, a personal computer dedicated to the (limited) computational requirements of a single user. Workloads with more strenuous requirements were off-loaded to *datacenters*, large facilities housing many distinct computer systems. While the housing and supporting hardware costs (e.g. power, cooling, network, ...) could be shared, the physical systems were predominantly owned by a single user and dedicated to a single task. This tight coupling between ownership and availability of computational resources meant users had little means of acquiring additional resources in a flexible manner. Leaving only the expensive and slow option of purchasing and maintaining additional hardware.

This limitation was first challenged by the emergence and subsequent success of the Internet. Cheap and fast digital communication allowed computational resources to be treated like a commodity, where providers offer storage and processing power to remote users. Initial attempts to share large numbers of geographically distributed resources through a unified interface resulted in the *grid computing* paradigm [**?**]. Despite the significant advances offered by grid computing, several limitations still remain. Grids often consist of resources shared by large organizations, making it difficult for new users to acquire resources without being affiliated to one of the existing partners.

Recent evolutions in IT infrastructure management have provided the technology to remove this barrier, allowing both small and large users to instantly acquire and release computational resources. One of the factors that has enabled flexible resource provisioning is the availability of efficient *virtualization* on commodity hardware. Advances in hard- and software have made large scale virtualization

within datacenters an attractive option. Virtualization technology has lowered the complexity and costs associated with IT infrastructure deployment, management and procurement. It allows one to manage an application and its execution environment as a single entity, a *virtual machine* (VM) capturing the full configuration of an application and its execution environment. Aside from the benefits of this technology in the context of privately owned data centers, these features have also fostered the development of a new IT infrastructure delivery paradigm that is based on outsourcing. The possibility to deploy an entire environment in a low-cost and hardware-neutral manner has paved the way for *Infrastructure as a Service* (IaaS) providers to open up their large datacenters to consumers, thereby exploiting significant economies of scale.

IaaS providers have limited knowledge about the workloads being deployed on their systems. Despite the large degree of isolation between VMs, performance interference can still occur between resource-intensive workloads. To optimize profits, providers must offer each user reasonable performance guarantees while also maximizing hardware utilization. These conflicting goals create opportunities for intelligent scheduling solutions at the datacenter level. Efficient scheduling requires improved insight into the computational requirements of the respective workloads. The necessary information can be acquired indirectly by monitoring metrics that describe performance characteristics. Gathering the correct set of metrics enables the creation of robust and accurate performance models. These models provide additional information that can be used to improve scheduling decisions.

1.1 Motivation and Problem description

The thesis is divided into two sections that relate to different aspects of the aforementioned modeling problem. First, we investigate the opportunities to reduce underutilization in large scale virtualized datacenters through overbooking. A typical IaaS provider offers a limited set of resource types with varying performance characteristics. Providers claim such resources deliver a predictable level of compute capacity without providing actual guarantees. Enforcing strict Quality of Service (QoS) for workloads with variable resource requirements can lead to over provisioning and infrastructure underutilization. Scheduling workloads with lower priority and QoS guarantees alongside high-QoS workloads offers a possibility to deal with underutilization. In this thesis, we evaluate the feasibility of increasing datacenter utilization through controlled overbooking. While utilization can be significantly increased with overbooking, limited insight into the effects of performance interference obscures the impact on high-QoS workloads.

The second part of this thesis will therefore focus on managing unexpected performance interference. We present a novel approach to mitigate its impact through improved datacenter scheduling policies based on performance models. Although the hypervisor provides adequate isolation on many levels (e.g. security, faults, ...), performance interference can still be an issue, particularly on resource-intensive workloads. Each VM is allocated a partition of the available resources and requires the hypervisor's cooperation to complete certain tasks (e.g. disk or network I/O). When multiple VMs share the same hardware, bottlenecks can occur both on the hardware as well as the hypervisor level. Current datacenter schedulers often

disregard or underestimate the impact of performance interference, leaving users vulnerable to unexpected performance fluctuations. Rapid increases in the number of CPU cores per system allow more VMs to share the same hardware in efforts to increase utilization. Slower improvements in I/O performance are likely to further increase the challenges resulting from performance interference in the near future. A solution needs to be provided at the datacenter level, where VM migration between hosts offers opportunities for improved scheduling. However, this approach requires insight into the resource consumption of individual workloads and their effect on co-scheduled VMs. By providing this information through performance modeling, scheduling policies can reduce the impact of performance interference. In this thesis we present innovative solutions to create the required performance models and improve datacenter scheduling efficiency.

1.2 Objective, Research Questions and Contributions

- **Can we increase utilization in virtualized datacenters by mixing low- and high QoS workloads while maintaining QoS guarantees?**

  Our contribution analyzes this scheduling problem and evaluates to what extent such mixed service delivery is beneficial for a provider of virtualized IT infrastructure. Traditionally, providers attempt to offer resources with a predictable performance profile, which can lead to underutilization in the absence of corrective measures.. The findings of our simulation study show that through proper tuning of a limited set of parameters, the proposed scheduling algorithm allows for a significant increase in utilization without sacrificing on performance dependability.

- **What information is needed to schedule resource-intensive applications while minimizing the impact of performance interference?**

  Our contribution proposes a novel approach based on slowdown predictions and application classification. Models predicting an application's sensitivity to interference are constructed by benchmarking applications in a linear scaling scenario. These predictions provide an upper bound to the slowdown an application can expect when scheduled with workloads that require similar resources. Clustering is used to automatically determine distinct application types and quantify the impact on co-scheduling performance. The combination of application types and performance predictions provides sufficient information to take corrective measures against performance interference.

- **Which training data is needed to produce an accurate Black Box performance prediction?**

  Our contribution evaluates multiple combinations of metrics and modeling techniques using a large and diverse set of applications as training data. We investigate the amount and type of training data required, and demonstrate the predictive capacity derived from the proposed set of profiled metrics.

- **To what extent can performance gains can be made by providing performance predictions and application classification to datacenter schedulers?**

  Our contribution measures the benefit of adding several additional layers of information to the scheduling process. Initial experiments demonstrate the benefits of using performance predictions when scheduling CPU and disk intensive workloads. Further experiments demonstrate the benefit of application classification in the presence of resource-intensive applications with diverse requirements i.e CPU, disk and network resouces. Using both performance predictions and application types we implement and evaluate several interference mitigating scheduling policies.

1.3 Structure

The thesis provides an introduction to a selection of the scheduling challenges that arise in virtualized datacenters, followed by our suggested solutions . In Chapter 2 we give a detailed overview of the technical challenges involved with x86 virtualization and discuss several of the available hard- and software solutions. After this introduction, Chapter 3 presents an opportunity for increasing utilization in virtualized datacenters. Possible gains are quantified and presented along with the remaining challenges. These challenges motivate the work presented in Chapter 4 and extended in Chapter 5. In these chapters, performance prediction models and application clusters are used to improve scheduling decisions. Finally, Chapter 6 presents the main conclusions of this dissertation before presenting open research questions and interesting future directions in chapter 7.

CHAPTER 2

Virtualization

2.1 Introduction

The problems and solutions presented in this thesis are unique to virtualized data centers. As such, we must first introduce the concepts and techniques used to virtualize workloads. In this chapter, we provide an overview of the different technologies which can be put under the header of virtualization.

In its broadest form, virtualization is a technique that is used to abstract, combine or divide computing resources to allow resource requests to be described and fulfilled with minimal dependence on the underlying infrastructure. The term virtualization has been used to describe concepts spanning several areas of computer science. Due to the variations in both the abstraction levels and the underlying architectures, an all-encompassing definition would be impractical. In this chapter we will therefore borrow a somewhat tailored definition from Nanda et al. [**?**].

Definition 1. *Virtualization is a technology that combines or divides computing resources to present one or many operating environments using methodologies like hardware and software partitioning or aggregation, partial or complete machine simulation, emulation, time-sharing, and many others.*

Although virtualization can be used both for aggregation as well as partitioning, most commonly the focus is on the partitioning of (computational) resources. Aggregation is usually found in the context of storage[1] or network[2] solutions such as [**?**] and [**?**]. In this chapter, we discuss virtualization as a means of partitioning. It provides the means to abstract lower-level resources and create multiple independent and isolated virtual machines.

[1] Storage virtualization refers to the process of abstracting logical storage from physical storage e.g. using RAID to present multiple physical disks as a single network partition.

[2] Network virtualization is the process of combining hard- and software network resources into a single, software-based virtual network.

2.1.1 Terminology

The following terms are essential for describing virtualization concepts and will be used frequently throughout the entire thesis.

Host: An operating system that executes directly on top of the physical hardware and executes the virtualization software as a user program. Certain approaches forgo this layer completely, opting instead to communicate directly with the physical hardware.

Guest: An operating system that executes on top of a virtual hardware representation provided by the virtualization software.

Virtual Machine (VM): A virtual representation of a real or specifically designed (hardware) interface. The combination of the guest and the virtual hardware can also be referred to as a VM.

Virtual Machine Image: A single file that contains an execution environment and the entire configuration of one or more applications.

Virtual Machine Monitor (VMM): A software layer that provides an abstraction between the physical hardware and one or more guests. The VMM can be executed directly on the physical hardware or inside a host. The VMM is also often referred to as the *hypervisor*.

User Application: A piece of software which interacts with hardware resources through an operating system.

An example of a simple virtualization stack that illustrates the terms explained above can be found in Figure 2.1.

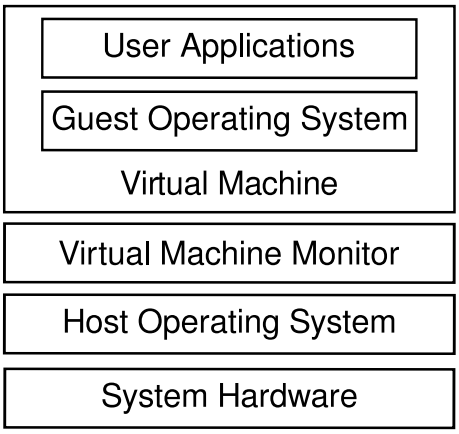

Figure 2.1: Abstract view of the layers in a virtualization stack.

2.1.2 History

The origins of virtualization can be traced back to research projects in the 1960's and early 1970's. These projects were targeted at partitioning large mainframes to improve hardware utilization and to allow for multitasking. Mainframes were expensive machines that often had to be shared between different user groups. By partitioning the system into virtual machines, multiple users could concurrently use the system each within their own operating system. These developments provided time- and resource-sharing capabilities that were revolutionary in a time when batch jobs were the only viable alternative. Each user had access to an isolated and protected virtual machine that presented itself as an instance of the physical hardware. Users could execute, develop and test applications without regard for the impact of their actions on other users.

The first successful virtual machine operating system was the CP-40 operating system for IBM's System/360 mainframe. It was quickly replaced by its successor, the CP-67 hypervisor. It provided each user with a *conversation monitor system*, in essence a single-user operating system that requested resources from the hypervisor. Virtualization became increasingly popular as a way to improve productivity during the 1970's and early 1980's, before gradually falling out of favor during the mid 1980's. The decline of virtualization is linked to the increasing popularity of client-server applications and the x86 architecture. Servers using x86 processors lacked the power to run multiple operating systems at the same time. Due to hardware being relatively inexpensive, dedicated machines were used for each separate application without much consideration. Over time, multiprocessing operating systems emerged and virtualization no longer provided the most efficient solution. In the late 1990's[3], virtualization resurfaced as a means to solve interoperability problems between a wide variety of operating systems and hardware configurations. Virtual Machines (VMs) were used to execute applications that were designed for other hardware and operating systems than the target platform. During the early 2000's, improvements in hard- and software once again made virtualization a popular solution to partition hardware. However, efficient virtualization on the popular x86 hardware first required several innovative solutions.

2.1.3 Challenges

As x86 servers became increasingly powerful, the previously underpowered machines could now theoretically support multiple operating systems running simultaneously. Unfortunately the x86 architecture, unlike its mainframe colleagues, was not designed with virtualization in mind. To clarify the problems, we first explain the workings of a virtualizable processor.

Every computer system exposes a "bare" hardware interface through which an operating system can interact with the hardware. This machine interface allows a single operating system to be in control of the hardware and execute instructions in the *highest privilege mode*. Only the first booted operating system kernel has the required permissions to perform *privileged instructions* (instructions reserved for operating systems, e.g. I/O operations). All other software (e.g. user programs or a second operating system) requires the cooperation of the first booted kernel to

[3] VMWare Inc. first launched VMware Workstation in May 1999

perform these instructions. In this scenario, when operating system code executes in a non-privileged state, it will cause a trap when performing an instruction requiring more privileges. Nanda et al. provide a definition for a virtualizable processor architecture and a privileged instruction in [**?**] .

Definition 2. *An architecture that allows any instruction inspecting or modifying machine state to be trapped when executed in any but the most privileged mode.*

Definition 3. *Privileged instructions are the subset of instructions that can only be executed on the processor when it is in the highest privilege mode, also referred to as "kernel" mode.*

These requirements provide the minimum constraints needed to create isolated virtual machines on top of the actual hardware. Processors include many privileged instructions that can affect the state of a machine, such as I/O instructions, or instructions to modify or manipulate segment registers, processor control registers, flags, etc. These instructions alter the machine state upon which an operating system depends. When running multiple operating systems concurrently, the virtualization layer needs to keep a record of each machine's state and update the state when required. To implement this functionality, a virtualization layer is added into or on top of the first booted kernel, turning it into a virtual machine monitor (VMM). When a guest executes an instruction for which it has insufficient privileges, a *trap*[4] will occur. The VMM then catches the trap, restores the machine state, executes the appropriate instruction(s) and returns the expected results.

As we previously mentioned, the x86 architecture was unfortunately not designed to be virtualizable. As a consequence, it contains instructions that produce inconsistent results when executed with insufficient privileges. Some instructions will fail silently instead of producing a trap, making it difficult to translate these instructions at runtime. It is this difficulty that initially made x86 virtualization seem improbable (for more details see section 2.3.3). Since the late 1990's however, several solutions have been devised such as binary translation, para-virtualization and hardware changes to the architecture itself.

2.1.4 Applications

Virtualization has resulted in many innovations that would have not been possible without its unique features. In this section we list some of the more interesting capabilities.

Server consolidation: Aggregating the workloads of multiple under-utilized or under-powered machines to a reduced set of powerful servers. Hardware performance has increased at such a pace that servers supporting a single application often only need a fraction of the available processing power. Consolidation can be used to lower hardware, space, cooling, maintenance, and management costs.

[4] A trap is a synchronous interrupt caused by an exceptional condition in a user process (e.g. insufficient privilege, division by zero, ...). This usually results in a switch to kernel mode where appropriate action is taken before returning to the user process.

Sandboxing: Virtual machines can be used to provide secure, isolated environments (sandboxes) for running foreign or less-trusted applications. Virtualization is an important component in building secure computing platforms. More and more sensitive data is stored and processed in digital formats, highlighting the importance of secure and isolated environments.

Snapshots: New management possibilities are created by the ability to capture the entire state of a running virtual machine and rollback or copy that configuration to other physical servers. For example, near instantaneous disaster recovery.

Live migration: VMs can be migrated to a new physical machine without any down-time. The interconnected nature of modern applications require them to be available with increasingly lower amounts of down-time. Live migrations removes down-time related to hardware maintenance.

Dynamic resource allocation: Modern hypervisors can be used to manage application performance using flexible execution environments. Through seamless changes in the allocated resources, requirements are allowed to fluctuate without needless over-provisioning.

Application consolidation: Legacy applications often require faster and consequently newer hardware while also depending on a legacy operating system. Fulfilling the dependencies of such legacy applications can be accomplished by virtualizing the runtime environment, using and sharing the faster resources through a virtualization layer.

Multiple operating systems: VMs provide the ability to simultaneously execute multiple different operating systems on single physical machine.

Virtual hardware: Hypervisors can be used to simulate unavailable hardware, e.g. virtual SCSI drives, ethernet adapters, switches and hubs.

Debugging: VMs can help to debug complicated software such as an operating system or a device driver by letting the user execute them on an emulated PC with full software controls.

Appliances: A VM image can be used to package an application together with the related operating system as an appliance, easing installation of an application and lowering the entry barrier for its use.

Testing/QA: VMs can be used to create a battery of arbitrary test scenarios that are difficult to produce or would otherwise require large amounts of dedicated hardware.

New hard- and software developments continue to improve virtualization performance and expand existing possibilities. There are many possible uses for virtualization and the list presented here is only a limited overview of the multitude of applications which can be realized.

2.2 Taxonomy

Due to the broad range of virtualization techniques, it can be helpful to create an overview which groups similar design principles, each accompanied by some well known examples. Understanding the different approaches requires a working knowledge of the basic architecture of a computer system. Each VM implementation abstracts one of of the key communication layers that make up a formal computer architecture. The source of the two major categories of VMs can be traced back to the need to abstract these layers. Figure 2.2 presents a high level view of the architecture components that relate to this discussion.

At the bottom layer we find the hardware comprising all the different components that make up a modern computer. To communicate with the hardware, the *Instruction Set Architecture* (ISA) is used. It consists of the *user ISA*, visible to an application program, and its superset, the *system ISA* used by the operating system to manage hardware resources. On top of the operating system we find the *Application Binary Interface* (ABI). The ABI allows user programs to access hardware resources via system calls to the operating system. The final layer below the application program layer is the *Application Programming Interface* (API). It allows access to the hardware resources via the ABI as well as *High Level Language* (HLL) library calls.

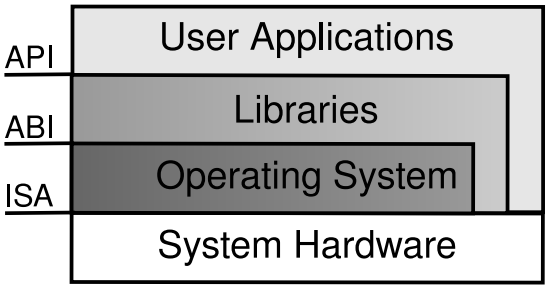

Figure 2.2: Key implementation layers in computer system architecture.

This short introduction allows us to describe the two main categories that make up this taxonomy: *Process VMs* and *System VMs*. Process VMs provide user applications with a virtual ABI environment, while system VMs provide a complete computer system by virtualizing the ISA layer. In the following sections of this chapter we focus on system VMs. To provide an exhaustive overview, we have also included a short section on process VM.

2.2.1 Process VMs

Process VMs virtualize the ABI, allowing user processes to run as VMs. They provide opportunities to manage processes, emulate instruction sets, optimize code and create cross-platform portability. In contrast to system VMs, they are designed to run a single program. This means that each VM supports a single process, possibly consisting of multiple threads. Process VMs can be divided in the following categories:

Multiprogrammed systems: The most well known provider of process VMs is the modern operating system, creating a process level VM for each concurrently executing program. It allows multiple application programs to run simultaneously by time-sharing hardware resources. Each program is managed through the system calls it uses. Programs operate under the illusion that they have exclusive access to the entire machine. A special case of a multiprogrammed system are process VMs that replicate other operating systems. A well known example is the open source WINE project, which allows Windows programs to run on top of other operating systems.

Emulators: Process VMs are also used to execute program binaries compiled for a different instruction set then the one used by the host. Two approaches can be used to emulate the original instruction set. The simplest but slowest method being *interpretation*. Each individual instruction is fetched, translated and emulated, often transforming a single instruction into many. When higher performance is required, this naive approach is replaced by *binary translation*, where entire code blocks[5] are translated to the native instruction set. Although initial translations may generate a large overhead, caching the results allows any recurring segments to execute much faster than straightforward interpretation. Because this approach is usually chosen for most emulators, they are also referred to as *dynamic binary translators*. A well known example is Rosetta, a dynamic binary translator for Mac OS X. It allows certain PowerPC applications to run unmodified on Intel-based Mac computers.

High Level VMs: Ensuring cross-platform portability is a challenging problem. Emulation based solutions require the development of an emulator to translate from and to each specific platform. Solving the problem by recompiling the code for the target platform often requires the code to be carefully constructed, modifying platform specific code segments to ensure compatibility. An easier way to achieve this goal is to design a process level VM together with an HLL application development environment. These High Level VMs do not correspond to an existing ISA but are designed for portability. Programs written in the HLL are compiled into portable code instead of ISA specific object code. This portable code can then be distributed to run on any implementation of the High Level VM. All software written in the HLL is portable to any platform, providing it is possible to implement the High Level VM on that particular target platform. A well know example of such a HLL is the Java programming language where code is compiled to "Java Bytecode" and executed on the Java VM.

Same-ISA binary optimizers: Dynamic binary translation can also be used to increase performance through code optimizations during translation. By analyzing the code on-the-fly during interpretation and translation, profiling information is gathered that can be used to optimize the code before execution.

[5] Code blocks are blocks of instructions containing one entry point, one exit point and no jump instructions.)

2.2.2 System VMs

System virtual machines (sometimes referred to as *hardware virtual machines*) allow the physical hardware of a computer system to be shared among multiple virtual machines by implementing virtualization at the ISA layer. They provide one or more simulated environments, each containing its own isolated operating system. In a system VM, the layer providing the hardware virtualization is called the *virtual machine monitor* or *hypervisor*. Its main function is managing the hardware resources to allow the concurrent execution of multiple guest OS environments and their user programs. When a guest operating system attempts to execute a privileged instruction, the VMM intercepts and performs the instruction on behalf of the guest, returning the expected results. As is the case with process VMs, a few subcategories can be distinguished. The first major division depends on the supported ISAs for guest operating systems (virtualization vs. emulation). The second is based on the location where the VMM is executed (native vs. hosted). An overview of the system VM portion of this taxonomy can be found in figure 2.3.

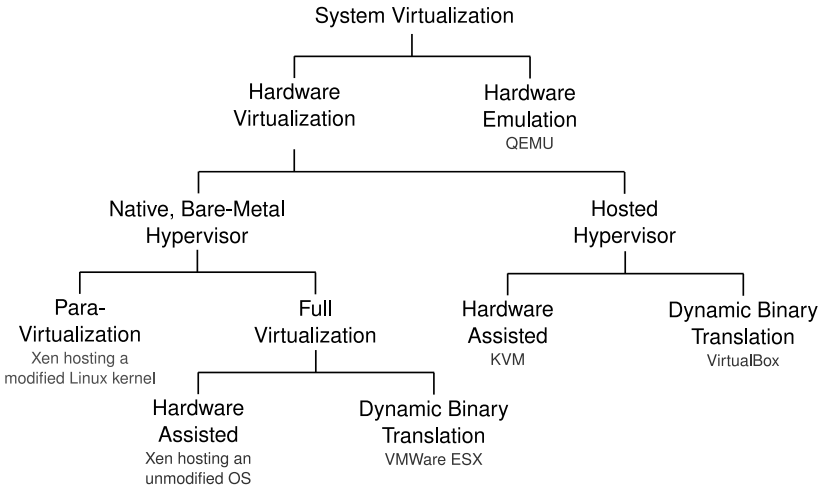

Figure 2.3: System level virtualization taxonomy (ISA)

Emulation: Through emulation, it is possible to support guest operating systems that are designed for an ISA different from the host ISA. Each instruction must be translated to the ISA of the physical machine. In conjunction with CPU emulation, all devices need to be replicated in order for an unmodified guest to be run. Binary translation is used on both the privileged and unprivileged instructions allowing the VMM to easily manage all resources. When hosting a guest that is compiled for the same ISA as the hardware, performance is severely lower than that of hardware virtualization. A popular example is QEMU[6] which can be considered a hosted VMM.

[6]QEMU also provides an accelerated mode in which a mixture of binary translation an native execution is used to improve performance.

Virtualization: Hardware virtualization is limited to same ISA guests and distinguishes between privileged or kernel mode instructions and non-privileged or user mode instructions. Privileged instructions are translated by the VMM, whereas non-privileged instructions are executed directly on the hardware. Using this technique, many instructions can be executed directly without any modification, providing performance comparable to native execution. When implementing hardware virtualization, a distinction is made between native, bare-metal (Type 1) and hosted (Type 2) hypervisors.

Native, Bare-Metal Hypervisor: Type 1 hypervisors run directly on the hardware, executing the intercepted privileged instructions either directly or through a privileged guest operating system.

Hosted Hypervisor: Type 2 hypervisors rely on the underlying host operating system to perform privileged instructions. Here, the VMM is a user program inside a host operating system. Therefore it cannot execute privileged instructions directly. Intercepted instructions are modified and passed down to the host.

This defines the broad categories in which most popular virtualization software can be divided. To provide a clear distinction between approaches, the following subdivisions are added:

- Native hypervisors can be divided in *para-virtualization* and *full virtualization*.
- Both full virtualization native hypervisors and their hosted counterparts can be split in two categories. On one side we have *binary translation* and on the other we have *hardware assisted virtualization*.

Para-Virtualization: Para-virtualization presents an ISA to the virtual machine that is similar but not identical to the machine that it is replicating. The idea is that using binary translation to replace privileged instructions is too computationally expense. To overcome this problem, the guest OS is modified by replacing all problematic code in the kernel with a virtualization-aware implementation. Instead of attempting to directly execute on the hardware, the guest is aware of the VMM and will use specially provided "hooks" present in the ISA. At the expense of a modified guest kernel, a simpler VMM implementation and better performance is thus attained. A well known example of para-virtualization is the Xen hypervisor.

Dynamic Binary Translation: Dynamic Binary Translation is the emulation of one ISA on top of another by translating the instructions from the source to the target ISA. The *dynamic* qualification signifies that this translation is done on-the-fly. Code blocks are translated and the results are cached for later usage. When virtualizing guests on same-ISA hardware the VMM can use this technique to emulate only the portions of the operating system code that contain privileged instructions. All other instructions are executed without intervention by the VMM to increase performance.

An important example on the host hypervisor side is the open source VirtualBox hypervisor. Unfortunately there are no (known) open source bare-metal dynamic binary translators. In the closed source realm, VMWare's ESX is the best know example.

Hardware Assisted Virtualization: Hardware-assisted virtualization adds architectural support to specially designed processors which enables efficient virtualization. Modern x86 processors have added support in the form of Intel VT and AMD-V for Intel and AMD processors respectively, all new desktop processors made by Intel and AMD support hardware assisted virtualization. With hardware assisted virtualization, the VMM can be made much smaller and more efficient. Instead of relying on software solutions to overcome design problems in the x86 ISA, extensions to the x86 architecture allow a more traditional trap-and-emulate model (see section 2.3.2) to be used when executing privileged instructions in user mode. A well known hosted hypervisor example is the KVM hypervisor. A bare-metal example can be found in an extension made to the Xen hypervisor. Ever since version 3.0, it has been possible to use hardware acceleration in Xen to virtualize unmodified guests. For Xen this is a potentially slower alternative that can be used when unmodified guests need to co-exists on the same server as para-virtualized guests. Both KVM and the hardware accelerated Xen have relatively straightforward implementations that would not have been possible without the hardware changes made to modern processors.

2.3 Hardware Virtualization

2.3.1 Introduction

In the previous section, a taxonomy of virtualization has provided us with an overview of the different approaches to hardware virtualization. In this section, we will go into detail on the requirements, the problems and the solutions associated with hardware virtualization. We start by examining the formal requirements for a virtualizable architecture in section 2.3.2. Taking into account these requirements, the problems of the x86 architecture are explained in section 2.3.3. Several methods have been devised to work around these inherent design problems. First the software solutions using binary translation and para-virtualization are discussed in sections 2.3.5 and 2.3.4, followed by the hardware assisted approach in section 2.3.6.

2.3.2 Formal requirements

Popek et al. [**?**] define what they believe to be the formal requirements for a virtualizable computer architecture. These requirement allow an architecture to support what we now refer to as hardware virtualization. They consider three properties to decide whether an architecture is suitable for virtualization: efficiency, equivalence and resource control.

The efficiency property: *All innocuous instructions are executed by the hardware directly, with no intervention at all on the part of the control program.*

All instructions that cannot change the hardware state should be executed directly on the hardware. In contrast to an emulator, a virtual machine should not intervene and analyze every single instruction executed by the virtual processor. In order to be efficient, the majority of the instructions should be executed on the real processor without any intervention.

The equivalence property: *Any program K executing within a control program resident*[7]*, with two possible exceptions, performs in a manner indistinguishable from the case when the control program did not exist and K had whatever freedom of access to privileged instructions that the programmer had intended.*

A program K running on top of a VMM should exhibit exactly the same behavior as if it would have run on the hardware directly. There are two exceptions to this rule resulting from *timing* and resource availability problems. Because the VMM has to intervene during certain instructions the program may take longer to execute. Assumptions about the length of time it takes to execute a sequence of instructions may lead to incorrect results. The resource availability problem comes stems the fact that there is a possibility that a request for memory cannot be satisfied. This can easily occur as the VMM itself takes memory space as well as any possible other virtual machines.

[7] Referred to as the VMM or hypervisor in this chapter.

The resource control property: *It must be impossible for that arbitrary program to affect the system resources, i.e. memory, available to it; the allocator of the control program is to be invoked upon any attempt.*

It should not be possible for a virtual machine to allocate resources without the explicit assistance of the VMM. To ensure this, no direct access to the system's real resources is allowed.

In order to derive the virtualization requirements, all ISA instructions are classified into three groups:

Privileged instructions: Instructions that only work in system mode and cause a trap when executed in user mode.

Control sensitive instructions: Instructions that are used to change the configuration of resources in the system. Instructions that either change the amount of memory that is available or affect the processor mode.

Behavior sensitive instructions: Those that are dependent on the configuration of the resources. Instructions that perform differently depending on their location in memory or the mode of the processor.

Using this classification and the properties stated before, the formal requirements for a virtualizable architecture can be defined as follows:

Definition 4. *For any conventional third generation computer, a virtual machine may be constructed if the set of sensitive instructions for that computer is a subset of privileged instructions.*

In other words, the essential requirement for a virtualizable infrastructure is that all sensitive instructions must result in a trap when executed with insufficient privileges (e.g. by a guest operating system). The processor should then return control to the VMM so it can decide how to properly handle the instruction. An intuitive definition for a VMM is also given by Popek et al.:

Definition 5. *We say that a virtual machine monitor (VMM) is any control program that satisfies the three properties of efficiency, resource control and equivalence. Then functionally, the environment which any program sees when running with a virtual machine monitor present, is called a virtual machine. It is composed of the original real machine and the virtual machine monitor.*

Even though these requirements were derived under simplifying assumptions, they still represent a convenient way of determining whether a computer architecture supports efficient virtualization and provide guidelines for the design of virtualized computer architectures. As discussed in the introduction however, the x86 architecture unfortunately does not adhere to these requirements. The next section details some of the problems when virtualizing the x86 architecture, followed by the corresponding solutions.

2.3.3 x86 Virtualization

The x86 architecture contains four modes of operation ranging from 0 to 3, see Figure 2.4. They are also referred to as *rings* or *privilege levels*. Ring 0, the lowest and most privileged level, is occupied by the operating system. Rings 1 and 2 are designated for operating system services (e.g. device drivers), but are rarely used. Ring 3, the highest and least privileged level, is reserved for user applications. A *call gate* is used so calls can be made from non-privileged applications to privileged system routines, transferring control between privilege levels. In [**?**], a number of hardware requirements are formulated and distilled into three requirements a modern processor must meet in order to be a virtualizable architecture:

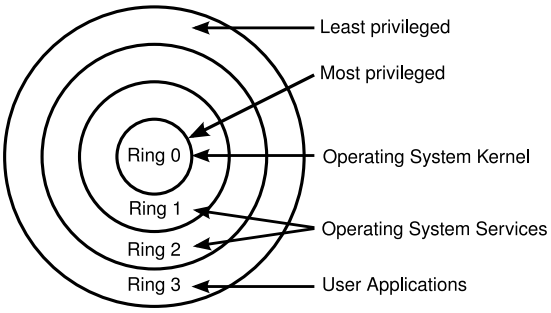

Figure 2.4: The x86 privilege level architecture.

Requirement 1: The method of executing non-privileged instructions must be roughly equivalent in both privileged and user mode.

Requirement 2: There must be a method such as a protection system or an address translation system to protect the real system and any other VMs from the active VM.

Requirement 3: There must be a way to automatically signal the VMM when a VM attempts to execute a sensitive instruction. It must also be possible for the VMM to simulate the effect of the instruction.

The first two requirements pose no problems and are handled adequately in modern x86 processors, however, requirement 3 does present a problem. The x86 ISA contains sensitive, unprivileged instructions. These instructions will execute when unprivileged without generating an interrupt or exception. This makes it impossible for a VMM to simulate the effect of the instruction. [**?**] lists a total of seventeen instructions that violate the third requirement and make the architecture unvirtualizable, they can be categorized in two groups:

- Sensitive register instructions: read or change sensitive registers and/or memory locations such as a clock register or interrupt registers:
  SGDT, SIDT, SLDT, SMSW, PUSHF, POPF

- Protection system instructions: reference the storage protection system, memory or address relocation system:
  LAR, LSL, VERR, VERW, POP, PUSH, CALL, JMP, INT n, RET, STR, MOV

Depending on the application that needs to be virtualized, these instructions may be very common. Regardless of the virtualization technique that is chosen, only applications with a limited number of privileged instructions can run at near-native speeds. In the following sections we look at different solution to overcome the x86 limitations.

2.3.4 Paravirtualisation

Paravirtualization involves modifying both the "host" and the guest operating system to replace non-virtualizable instructions with calls to the hypervisor. Besides these problematic instructions, the hypervisor also provides interfaces for other critical operations such as memory management, interrupt handling and time keeping. The goal of paravirtualization is to achieve a lower performance overhead compared to binary translation. The downside being the requirement to modify each operating system that is to be virtualized, resulting in lower compatibly and portability. A popular open source implementation is the Xen ([**?**]) project. It uses paravirtualization to virtualize the processor and memory using modified linux kernels. All I/O operations are performed using modified OS drivers. In Xen's relatively short history we can find an example of the compatibly and portability problems a paravirutalization approach may encounter. During the development of Xen 1.x, Microsoft Research, along with the University of Cambridge Operating System group, developed a port of Windows XP to allow paravirtualization using Xen. This was possible due to Microsoft's Academic Licensing Program. The terms of this license however do not allow this port to be published, limiting Xen guest support to mostly open source operating systems (e.g. Linux, OpenBSD, OpenSolaris, ...) and NetWare. An abstract view of paravirtualization is shown in figure 2.5.

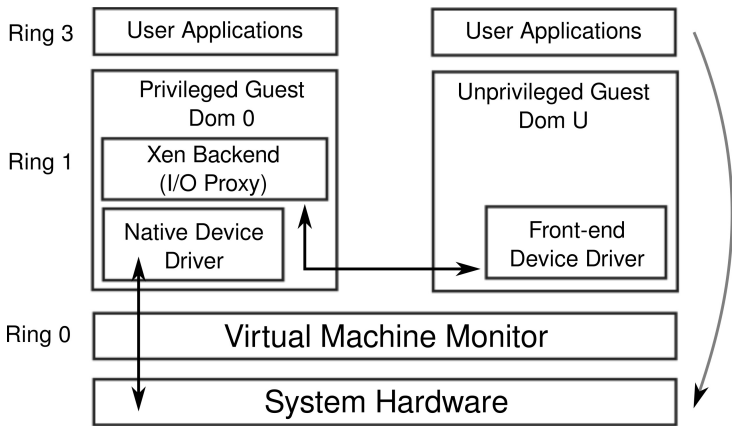

Figure 2.5: Abstract view of paravirtualization on the x86 architecture.

2.3.4.1 Xen

A Xen system is structured with the hypervisor in the lowest and most privileged layer. Above this layer are one or more guest operating systems, which the hypervisor schedules across the physical CPUs. The first (privileged) guest operating system, in Xen terminology called "domain 0" or dom0, is booted automatically after the hypervisor boots and given special management privileges and direct access to the physical hardware. The system administrator can log into dom0 in order to manage any further guest operating systems, called "domain U" or domU in Xen terminology. Next to this control interface, the hypervisor also contains the hardware interface used by dom0, a virtual CPU and memory management unit, and an event channel.

The Xen hypervisor is executed in ring 0 while the guest operating systems are located in ring 1. All privileged instructions need to be validated and executed within Xen, this is accomplished by replacing them with *hypercalls*. A hypercall is a software trap from a domain to the hypervisor causing a context switch from ring 0 to 1. Hypercalls can be compared to syscalls, software traps from an application to the kernel. Like a syscall, the hypercall is synchronous, but the return path from the hypervisor to the domain uses *event channels*. An event channel is a queue of asynchronous notifications, and notify of the same sorts of events that interrupts notify on native hardware.

Rather than emulating existing hardware devices, as is typically done in fully-virtualized environments, Xen exposes a set of clean and simple device abstractions. This allows an interface that is both efficient and satisfies the requirements for protection and isolation. Only dom0 has direct physical access to all hardware, it exports simplified generic class devices to each DomU. Rather than emulating the actual physical devices exactly, the hypervisor exposes idealized devices. For example, the guest domain views the network card as a generic network class device or the disk as a generic block class device. Dom0 runs a device driver specific to each actual physical device and then communicates with other guest domains through an asynchronous shared memory transport. The physical device driver running in Dom0 or a driver domain is called a backend, and each guest with access to the device runs a generic device frontend driver. This frontend can be configured like any normal physical device, depending on the needs of the guest. The backends provide each frontend with the illusion of a generic device that is dedicated to that domain. The backend understands the details of the real physical device and packages the generic device requests from each frontend into the appropriate form for forwarding to the hardware. Backends also implement the shared accesses required to give multiple guest domains the illusion of their own dedicated copy of the device.

2.3.5 Binary translation

Virtualization based on dynamic binary translation is used in both hosted and bare-metal solutions. It uses a combination of direct execution and binary translation techniques as shown in figure 2.6. Non-virtualizable instructions in the kernel code are translated into new sequences of instructions that have the desired effect on the virtual hardware. The VMM provides each VM with a virtual representation of all hardware components (e.g. BIOS, devices, memory, ..). The guest OS does not

require any modification and can thus remain completely unaware of the virtualization layer. The binary translation approach is the only one that does not require any hard- or software modifications when using the x86 architecture as a host. The performance can be very close to native solutions as all non-privileged instructions are executed unmodified. Code blocks containing privileged instructions are translated on the fly and the results of these translations are cached to improve future performance.

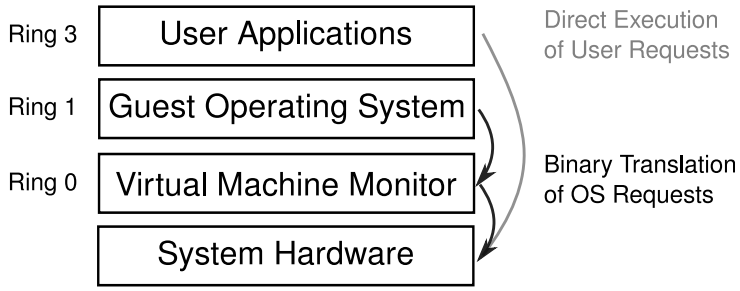

Figure 2.6: Virtualization using binary translation on the x86 architecture.

The most popular dynamic binary translation implementations are provided by VMWare (Workstation, ESX server). A popular open source alternative is the hosted VirtualBox hypervisor. We will use this open source implementation to get a detailed look at all the required components. The following section was adapted from the VirtualBox technical documentation[8]. Although VirtualBox does not require hardware virtualization, it can be configured to make use of any available hardware features. In the following section we focus on the binary translation implementation.

2.3.5.1 VirtualBox

Many processes are used to implement the GUI aspect of a workstation solution (e.g. windowing, mouse control) but to simplify the discussion we will only explain the components that are responsible for the virtualization itself. The internal component of the VirtualBox implementation are spread over many more or less separate components. Together they form a static “backend”, or black box around which various graphical, command-line and remote interfaces can be written.

From the perspective of the host operating system a virtual machine is just another process. Not much tweaking is needed to efficiently support virtualization. A ring 0 driver must be loaded in the host before VirtualBox will work, but this driver does less than one might assume. Only a few important tasks are performed by this driver:

- Allocating physical memory for the VM.
- Saving and restoring registers and descriptor tables when a host interrupt occurs while a guest’s ring 3 code is executing (e.g. when the host OS wants to reschedule).

[8] www.virtualbox.org/manual

- Switching from host ring 3 to guest context.
- Enable or disable hardware virtualization support.

Most importantly, the ring 0 driver does not interfere with the OS's scheduling or process management. The entire guest OS, including its own hundreds of processes, is only scheduled when the host OS gives the VM process a time slice. After a VM has been started, from the processor's point of view, the computer can be in one of several states :

1. The processor can be executing host ring 3 code (e.g. from other host processes), or host ring 0 code, just as it would be if VirtualBox was not running.
2. The processor can be emulating guest code (within the ring 3 host VM process). VirtualBox tries to run as much guest code natively as possible. If needed, it can (slowly) emulate guest code as a fall-back when it is not sure what the guest system is doing, or when the performance penalty of emulation is not too high. The emulator is based on QEMU and typically steps in when:
   - Guest code disables interrupts and VirtualBox cannot figure out when they will be switched back on.
   - For execution of certain single instructions. This usually happens when a guest instruction has caused a trap and needs to be emulated.
   - For any real mode code[9] (e.g. BIOS code, a DOS guest, or any operating system startup).
3. The processor can be running guest ring 3 code natively (within the ring 3 host VM process). This is by far the most efficient way to run the guest. The more we have to leave this mode, the slower the VM is compared to a native OS, because all context switches[10] are expensive.
4. The processor can be running guest ring 0 code natively. The guest only thinks it's running ring its code in ring 0, instead VirtualBox has fooled the guest OS to instead enter ring 1 to execute all non-privileged OS instructions.

2.3.5.2 Code Scanning, Analysis and Patching

As previously mentioned, the goal is to execute as much code as possible natively and use the recompiler as a fall-back only on very rare occasions. For guest ring 3, the performance caused by the recompiler is not a major problem as the number of faults is generally low. However, as was also described previously, the guest operating system is manipulated to actually execute its ring 0 code in ring 1. This causes a lot of additional instruction faults, as ring 1 is not allowed to execute any privileged instructions. With each of these faults, the VMM must step in and emulate the code to achieve the desired behavior. While this normally works perfectly well,

[9] All x86 processors boot into real mode after a reset. This ensures compatibility with the original 8086 processor and allows initialization of the registers that are required for protected mode.

[10] A context switch is the process of storing and restoring the state (context) of a CPU such that multiple processes can share a single CPU resource.

the resulting performance would be very poor since CPU faults tend to be very expensive and there will be thousands and thousands of them per second. To make things worse, running ring 0 code in ring 1 causes occasional compatibility problems. Because of design flaws in the x86 architecture that were never addressed, some system instructions that should cause faults when called in ring 1 unfortunately do not. Instead, they just behave differently. It is therefore imperative that these instructions be found and replaced.

To address these two issues, two techniques are introduced called the "Patch Manager" (PATM) and "Code Scanning and Analysis Manager" (CSAM). Before the execution of ring 0 code, the code is scanned recursively to discover problematic instructions. Wherever necessary, *in-situ*[11] patching is performed, i.e. the offending instructions are changed into a jump to hypervisor memory where a compatible replacement has been placed by a code generator. This is a very complicated task, finding and solving many different code problems. Additionally, whenever a fault occurs the offending code is analyzed to determine if it can be patched to prevent more expensive faults in the future.

2.3.6 Hardware assist

With the advent of Intel VT-x (Vanderpool) and AMD-V (SVM) in 2006, implementing a hypervisor for the x86 architecture has become significantly more straightforward. Both Intel and AMD's approach target the problem of the privileged instructions by creating a new CPU execution mode that allows the VMM to run in a new root mode "below" ring 0. As we can see in Figure 2.7, privileged instructions automatically trap to the VMM removing the need for either paravirtualization or binary translation. The guest state is stored in Virtual Machine Control Structures (Intel) or Blocks (AMD). At first glance, it may seem that there is no more need for paravirtualization or binary translation when using a modern x86 CPU.

Unfortunately, there are still some drawbacks in the current hardware implementations. There is a relatively high performance overhead when handling privileged instructions and the programming model that needs to be implemented is rather rigid. All privileged instructions must go through the computationally expensive trap-and-emulate cycle. Furthermore, the programming model does not leave much room for software flexibility in managing the frequency or cost of these expensive hypervisor to guest transitions. For example, using binary translation it's possible to chain translated code sections when multiple privileged instructions are executed consecutively whereas hardware assisted will trap each instructions separately. Because of these limitations, virtualization using first-generation hardware assist on x86 can still be slower than the traditional approaches.

Second generation hardware-assisted virtualization adds hardware support for memory virtualization, called Extended Page Tables (EPT) or Rapid Virtualization Indexing (RVI) for Intel and AMD respectively. Before, hypervisors were required to maintain shadow page tables to store the physical location of guest memory. The hypervisor intercepted guest page table updates to maintain coherent ghost page tables. Adding hardware support for Memory Management Unit virtualization removes some of the overhead incurred by applications which require frequent

[11] A Latin expression for "in place".

page tables updates. While it can improve performance significantly for specific benchmarks, the impact om application performance can vary greatly.

A popular hardware assist based implementation is the KVM[12] hypervisor, which we will discuss in the following section.

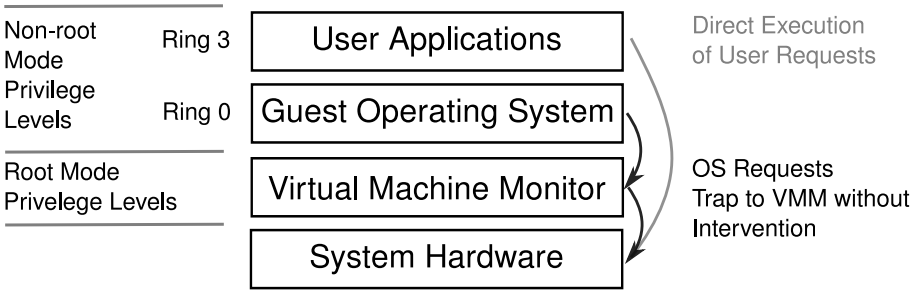

Figure 2.7: Abstract view of hardware assisted virtualization on the x86 architecture.

2.3.7 KVM

The major design difference between KVM and other, older hypervisors can be summarized into two observations:

- Using hardware assist it is possible to avoid a large amount of the implementation work that goes into a traditional hypervisor.
- A lot of effort can be avoided if we don't need to implement and perfect the basic scheduling and memory management functions of hypervisors.

The first idea should be fairly obvious, the new hardware removes the need to implement the binary rewriting or paravirtualization techniques. The second might require some further explanation. The basic scheduling functions[13] are becoming increasingly complex with multiple threads per processors, multiple processors per socket and multiple sockets per system. Likewise, managing the virtual memory associated with each virtual processor becomes more complex as more processors are added. While large efforts are made to perfect the scheduling functions in existing hypervisors, KVM has found a different and mature scheduling and memory management system: the Linux kernel. KVM focuses on virtualization. Scheduling processes, memory and I/O management are all done by Linux kernel components that have been available for a long time, are widely in use and well tested.

By adding virtualization capabilities to a standard Linux kernel, the extensive fine-tuning work that has gone into the kernel can be reused. Scheduling of processes and memory management is left to Linux. Handling I/O is left to QEMU, which can run guests in userspace and has an mature device model. Under this model, every virtual machine is a regular Linux process scheduled by the standard

[12] www.linux-kvm.org

[13] The scheduling functions determine which VM's should get physical CPU's and system resources, which physical CPU's they should be associated with, and in which order.

Linux scheduler, having its memory allocated by the Linux memory allocator. As we can see in figure 2.8, each guest consists of two parts: the userspace part (QEMU) and the guest part. The guest physical memory is mapped in the process virtual memory space, so guests can be swapped as well. Virtual processors in a VM are simply threads in the host process.

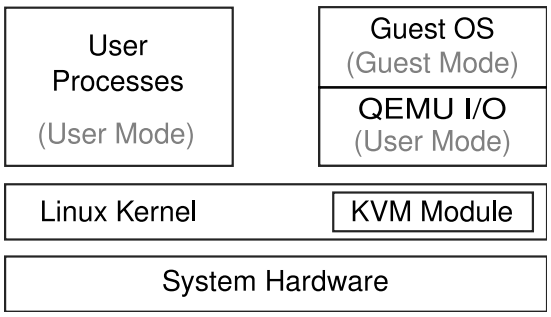

Figure 2.8: KVM based virtualization, VMs as a regular Linux process.

A normal Linux process can be two execution modes: user-mode and kernel-mode. KVM introduces a third *guest mode*. From a KVM standpoint, the different modes are used as follows:

Guest mode: Execute non-I/O guest code.

User mode: Perform I/O on behalf of the guest using QEMU devices.

Kernel mode: Switch into guest mode when the VM is scheduled, and handle any exits from guest mode due to I/O or privileged instructions.

As the KVM hypervisor is integrated into the Linux kernel through a module, it automatically receives updated hard- and software features without additional effort. This also allows the hypervisor to be a relatively minimal system, reducing the possibility for bugs or software inefficiencies.

CHAPTER 3

Multiplexing Low and High QoS Workloads

This chapter is based on "Multiplexing Low and High QoS Workloads in Virtual Environments" [?]

Abstract

In this chapter, we address the VM scheduling problem in which workloads that require guaranteed levels of CPU performance are mixed with workloads that do not require such guarantees. We introduce a framework to analyze this scheduling problem and evaluate to what extent such mixed service delivery is beneficial for a provider of virtualized IT infrastructure. Traditionally, providers offer IT resources under a guaranteed and fixed performance profile, which can lead to underutilization. The findings of our simulation study show that through proper tuning of a limited set of parameters, the proposed scheduling algorithm allows for a significant increase in utilization without sacrificing on performance dependability.

3.1 Introduction

A current trend in IT infrastructure management is the reliance on virtualization technology to mitigate the costs of application and IT infrastructure deployment, management and procurement. Virtualization technology allows one to manage an application and its execution environment as a single entity, a *virtual machine* (VM). It allows for the entire configuration of an application and its execution environment to be captured in a single file, a virtual machine *image*. These virtual machine images can be deployed in a hardware-agnostic manner on any hardware that hosts a compatible virtual machine *monitor* (VMM) such as Xen [**?**] or VMware's VMM. VM monitors thereby offer flexibility in partitioning the underlying hardware resources and ensure isolation between the different virtual machines that are running on the same hardware. Aside from the benefits of this technology in the context of privately owned data centers, these features have also fostered the development of a new IT infrastructure delivery paradigm that is based on outsourcing. The possibility to deploy an entire environment in a low-cost and hardware-neutral manner has paved the way for *cloud* [**?**, **?**] Infrastructure as a Service (IaaS) providers to open up their large datacenters to consumers, thereby exploiting significant economies of scale.

One of the most well known providers in this new market is Amazon with their Elastic Compute Cloud (EC2) [**?**] offering. As is typical for an IaaS provider, Amazon offers a discrete number of resource types or *instance types* as they are called, with varying performance characteristics. Such an instance type delivers a guaranteed level of compute capacity. An EC2 *small* instance type for example, contractually delivers a performance that is equivalent to a 2007 Opteron processor with a 1.0-1.2 GHz clock frequency. The performance guarantees in this service delivery model are crucial because the use of the compute service is paid for by the hour, and not by actual compute capacity delivered or used. The combination of these performance guarantees and the fact that virtual machine workloads can vary significantly can lead to infrastructure underutilization in absence of corrective measures. In addition, the ability to buy *reserved* instances at EC2 that have a guaranteed level of performance *and* availability, further increases the chances for underutilization. The recent addition of a spot market [**?**] for EC2 instances whereby instance types are dynamically priced and potentially terminated by the provider if their standing bid does not meet the spot price, provides an indication for this problem of (temporary) underutilization. The addition of this market mechanism changes the scheduling problem within the datacenter from one in which a given set of workloads need to be balanced out over the available hardware, to one in which the change of an admission parameter,the instance's spot price, can trigger an influx of additional VM workloads into the datacenter. These workloads run under lower availability guarantees as they can be shutdown by the provider if they cause interference with workloads that run under a high availability regime, such as the reserved instance or on-demand instances at EC2[1].

Scheduling workloads that have low priority and quality of service (QoS) guarantees in terms of performance, alongside with high-QoS workloads thus offers a

[1] Note that EC2 uses an indirect mechanism for this by increasing the spot price to a level that rises above the standing bid of an adequately high number of spot instance workloads. This clears them for shutdown under the contractual rules of the trading agreement.

possibility to deal with underutilization. Consider for example the addition of a batch job workload to a 4-way server that is running a VM with four cores hosting a high priority web service. The web service's spiky load pattern opens up the possibility for filling in underutilized periods with the batch workload. Such a scheduling approach must ensure that high-QoS workloads do not suffer from performance degradation caused by their multiplexing with low-QoS workloads. At the same time, enough low-QoS workloads should pass admission control in order to achieve the highest possible utilization and throughput of the infrastructure.

Although some commercial products exist, such as VMware's vSphere, that perform load balancing in a cluster for a given set of virtual machines, no definite solution exists today for tackling this problem if a decision can be made to accept additional low-QoS workloads. In this chapter, we present a simulation framework to analyze the performance of VM scheduling problems and evaluate a scheduling algorithm that is tailored towards the multiplexing of these high- and low-QoS workloads in a virtual machine context. We demonstrate that by tuning a limited set of parameters a tradeoff can be made between maximizing utilization and avoiding workload interference.

3.2 Model

3.2.1 Resource and Job Model

We explore the VM scheduling problem in a setting with one infrastructure provider P, that hosts a set of m machines M_j ($j = 1, \cdots, m$). These are considered to be identical parallel machines so each machine is able to execute any job from the set of n jobs J_i ($i = 1, \cdots, n$), and for the machine's processing capacity s_j we have, $\forall i \in \{1, \cdots, m\} : s_i = 1$. A job, which models the execution of a virtual machine instance, has a varying load pattern over time and is sequential, i.e. it runs on only one machine at a time. A job has a release time r_i, and a duration p_i. We consider two types of QoS levels for jobs. High-QoS jobs must be able to start at time r_i and should be able to allocate the full processing power of the machine on which they are deployed. These jobs are not preemptible, e.g. a virtual machine running a relational database. Low-QoS jobs can be preempted at a fixed cost c_p. In this work, we assume that job preemption requires a suspension of the virtual machine. Equivalently, a resumption of a virtual machine instigates a cost c_r. The job startup costs (c_b) and termination costs (c_t) are also modeled as we are dealing with VMs. For preemption, we only consider the case wherein a VM is swapped out of memory to make room for the other VMs that run on the server. An example of a workload that is amenable to a low-QoS regime is a virtual machine that executes low-priority batch jobs.

A machine corresponds to a virtualized core of a server that runs a virtual machine monitor. The provider P operates a cluster of such servers. A machine can accommodate more than one job at a time. We assume that the distribution and multiplexing of a VM's workload over the virtual cores of a server is managed by the virtual machine monitor and do not explicitly model this behavior. At this time, we do not yet explicitly model the overheads that such multiplexing brings in terms of technical considerations such as I/O contention for resources or cache line invalidations. Although these aspects can certainly have a significant impact on this study,

they are also very application dependent and difficult to model and simulate. In that respect, this study maps out the maximum performance that can be attained under the proposed scheduling approach.

3.2.2 VM management model and simulation framework

For managing the distribution of virtual machines over multiple servers in the cluster, a *virtual infrastructure manager* (VIM) is required. There are multiple such managers currently available such as vSphere (VMWare's commercial offering), or one of the open source alternatives such as OpenNebula [**?**] or Eucalyptus [**?**]. Depending on the capabilities of the VIM, a set of features and operations is available to manage the execution of the VM instances on the cluster. Because of its generality, we have chosen to model our scheduling problem in the context of the features offered by the OpenNebula toolkit. The open nature of the project, the emphasis on being a research platform and the generality of its feature set are the main factors that influenced this choice.

One of the schedulers already available for OpenNebula is the Haizea [**?**, **?**] scheduler. The VM operations available to the scheduler are *shutdown*, *start*, *suspend* and *resume*. The scheduler is assumed to have no knowledge of the duration p_i. In order to deal with infrastructure underutilization, we take an overbooking approach. That is, we allow the scheduler to allocate more resources than physically available on the cluster node. Such an overbooking has to be actively managed by active scheduling decisions in order to limit the interference of low-QoS loads with high-QoS loads. As the Haizea scheduler already supports many of the features required for overbooking, such as the support for differentiation between multiple job types, it is chosen as the basis for our scheduler.

All of the scheduler's decisions result in a series of commands and corresponding VM states that can be used to drive the two enactment backends available in Haizea. The first is a simulated backend used in the presented experiments, the second drives the OpenNebula virtual infrastructure engine where Haizea can be used as an alternative to the default scheduler. One of the major benefits of the second backend is that all the scheduling algorithms implemented within the extended framework are automatically compatible with OpenNebula. An advantage of this choice is that the results of our simulation studies can be verified in a real-world setting without much additional cost.

Haizea's simulation mode uses a simulation core that keeps track of all *actions* that are scheduled with a specific firing and finishing time. The simulation steps through time by subsequently adjusting the simulator's virtual clock to the time of the next action. At each step, the state of the simulated environment is updated and user code can step in to schedule new actions. A single VM operation, such as suspend, can involve one or more actions, depending on the level of detail in the VM management model. For example, one could explicitly model the time required for state checkpointing, or the I/O operation involved in storing the checkpoint.

With a configurable time frequency, our scheduler performs an *overbooking* step. In such a step all available machines are polled to obtain the active jobs and their current utilization. This information is then used to determine all the VM operations that are required, based on the scheduling policy's options. Interspersed with these fixed steps lie *management* steps. During the management steps, all events that do

not coincide with overbooking step times are performed e.g. issuing a shutdown command when a VM has finished its workload.

3.3 Scheduling Algorithm

Any overbooking scheme will have the same general goal: reduce resource wastage due to underutilization while at the same time having a minimal impact on the existing resource users. As a result of their suspend and resume capabilities VMs are uniquely suited for this goal provided they have different types of QoS requirements. A lower priority VM can be suspended and resumed at a later, more opportune time and/or location without losing any performed work. The scheduler determines the suitability of machines for low-QoS jobs and only launches the job if sufficient resources are available. For high QoS jobs, the scheduler installs reservations to make sure resources are available for the entire duration of the workload. Low-QoS jobs are queued up until machines are available.

As jobs only use a single machine, the number of jobs supported by a single cluster node can be expressed in *slots*. Each slot is equivalent to the processing capability of a single CPU core. As such, we will refer to a machine M_i as a slot in the remainder of this chapter. Slots provide a convenient abstraction to specify both the available physical resources as well as the maximum allowed amount of overbooking.

High-QoS jobs may require the full processing capacity of the reserved slots at some point in time but it is reasonable to assume this is not permanently the case. The reserved but unused resources pose both an opportunity and a challenge. There is an opportunity to increase overall utilization by scheduling in low-QoS workloads. Depending on the QoS guarantees, interference with high-QoS workloads with must be completely avoided or kept within reasonable bounds. In contrast to the EC2 approach, we want to preserve the work that has been completed in a low-QoS VM and therefore do not kill it if it is detected to interfere with high-QoS VMs. Therefore, our scheduler must take into account the overheads of suspending and resuming low-QoS VMs. Suspending as well as starting and stopping a VM can be a resource intensive operation. Depending on the configuration of the cluster, it is possible that all four major resources (CPU, memory, disk and network) are heavily taxed.

We quantify the interference between VMs by measuring the CPU utilization on a node in excess of 100%. As mentioned before, this is only one dimension of interference that can exist between VMs that are deployed on the same node. Other dimensions such as contention for disk I/O bandwidth will be investigated in later chapters.

A simple and effective method to put restrictions on the allowed ranges for overbooking is the introductions of bounds. The base algorithm determines its actions using a lower and an upper bound. The lower bound puts a limit on the maximum node utilization for nodes where new low QoS VMs are booted. The upper bound is used to decide when a VM should start suspending. Keeping in mind the overhead of starting and suspending a VM, the algorithm will not schedule more than one of these operations simultaneously on a node.

Our scheduling algorithm works in two steps: scheduling new overbooking requests and evaluating running requests. The first step, for which pseudo-code is shown in Algorithm 1, works as follows. The algorithm starts by obtaining a list

of all the nodes that can currently support an additional VM. The suitability of a node is determined by comparing the average node utilization in the previous epoch (including the loads introduced by possible overbooked VMs) with a configurable lower bound. All nodes with a utilization lower or equal to this lower bound are added to a list of overbooking candidates. After suitable candidates are found the list of low-QoS requests is updated: incoming requests are added to the back of the queue while suspended requests are added to the front. Suspended requests are ordered by initial arrival time with the oldest appearing at the front of the queue. With all necessary data gathered, VMs can be scheduled until either the available nodes or requests are exhausted.

```
Input: Set of nodes, Set of vm_requests, lower_bound
foreach Node i do
    if Utilization(i) ≤ lower_bound then
        available_nodes.add(i) ;
    end
end
Update(vm_requests) ;
while available_nodes remaining & vm_requests remaining do
    vm = vm_requests.pop() ;
    n = available_nodes.pop() ;
    Schedule(vm on Node n) ;
end
```

Algorithm 1: Adding Overbooked VMs

Since utilization is a volatile property, the conditions for overbooking will need to be evaluated at regular intervals. The pseudo code for this part of the algorithm can be found in Algorithm 2. All nodes supporting one or more overbooked VMs are evaluated. If the total utilization average in the previous epoch equals or surpasses the set upper bound, the VM that was added last will be suspended.

```
Input: Set of nodes, upper_bound
foreach Node i do
    if Utilization(i) ≥ upper_bound then
        vm = overbooked_vms(i).get_last() ;
        Suspend(vm) ;
    end
end
```

Algorithm 2: Suspending Overbooked VMs

3.4 Experiments

In this section, we evaluate the performance of the proposed scheduling algorithm. We first outline our experimental setup after which we present and discuss our results.

3.4.1 Experimental setup

Our experimental setup consists of three major aspects: the cluster used to deploy the VMs, a list of high- and low-QoS requests and the load generators attached to the requests. The cluster consists of 50 homogeneous octacore nodes. Due to lack of real-world traces, we generated a non-trivial synthetic load pattern that is reminiscent of the behavior of real-world workloads. We introduce the following three different application types[2] following [**?**]:

Noisy: Starting from a mean utilization value μ, a load pattern is generated by drawing random numbers from a normal distribution $N(\mu, 15)$. An example of a noisy load pattern for $\mu = 75$ can be found in Figure 3.1.

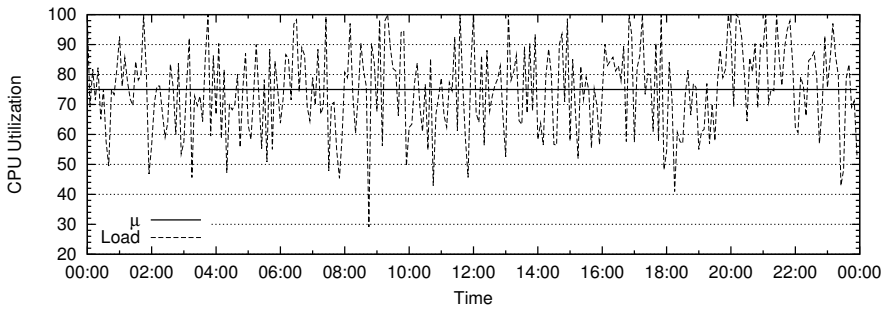

Figure 3.1: Sample noisy load pattern.

Spiky: This load pattern is based on a normal distribution with $\sigma = 5$. To add load spikes to the pattern, each drawing of the load distribution has 1% chance of generating a spike with 90% chance of having a positive one. Each spike has 50% chance of continuing into the next time step. An example spiky load pattern for $\mu = 75$ can be found in Figure 3.2.

Business: A business load pattern is slightly more complicated in that a function is used to determine the μ parameter of the normal distribution $N(\mu, 5)$ depending on the time of day. The value of μ is calculated with a piecewise function that represents utilization fluctuations coinciding with business hours. The function is configured with a minimum (min) and a maximum (max) utilization value. Utilization rises from min to max between 8.00 and 10.00 in the morning. Between 11.30 and 13.30 there is a slight drop representing lunch hours. In the evening there is a second decline dropping back to min between 16.00 and 18.00. The incremental utilization changes between min and max

[2] By manipulating a limited number of parameters we can emulate a wide range of applications.

are calculated by adjusting the amplitude and period of a $sine$ function. During weekends, the function returns the minimum value. An example business load pattern for $min = 50$ and $max = 90$ is shown in Figure 3.3.

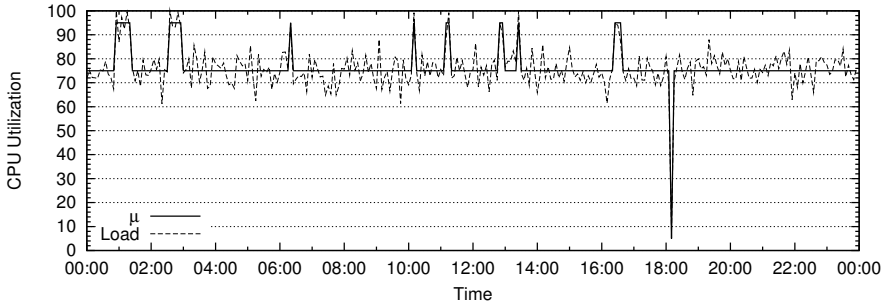

Figure 3.2: Sample spiky load pattern.

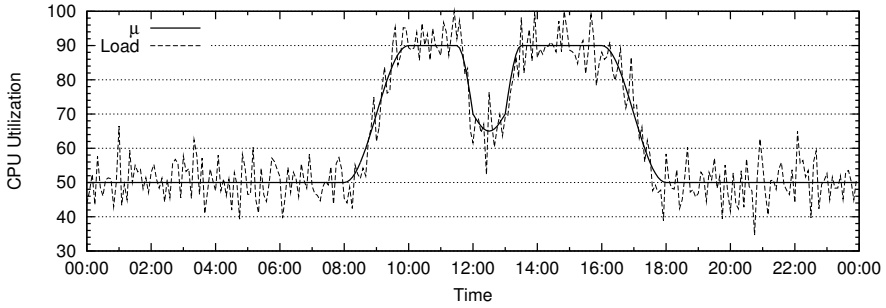

Figure 3.3: Sample business load pattern on a weekday.

Each high-QoS application has an equal probability of generating one of the three load patterns. For the spiky and noisy load patterns, μ is drawn from a normal distribution $N(75, 15)$. For the business load pattern, $min = 50$ and $max = 90$. An example of a possible workload during a weekday on a node in the cluster can be found in Figure 3.4. High-QoS applications are generated in such a manner that load patterns are randomly distributed among the different nodes on the cluster. Each low-QoS application has a noisy load pattern with $\mu = 90$ simulating CPU intensive batch jobs. Each separate application consists of a single job. High-QoS jobs are generated in such a manner that all physical slots are continuously occupied. Using 50 octacore nodes this means there are 400 high-QoS applications running at any given time, one for each core. The low-QoS job arrival rate is set a level that ensures the queue never empties. The maximum number of concurrently executing low-QoS applications depends on the overbooking slots per node.

All application runtimes are generated according to a geometric distribution. If X is the runtime in minutes, the probability is expressed in equation 3.1 for $n = 30, 60, 90, ...$ with p equaling 0.1% and 1% for respectively high- and low-QoS applications.

$$Pr[X = n] = p\left(1-p\right)^{\left(\frac{n}{30}-1\right)} \tag{3.1}$$

Preliminary tests indicated that the results for running the simulation for one week and for one month produced equivalent results. This is a logical consequence of the weekly repeating pattern. To reduce the time needed to produce results for the numerous tests, we reduced the time horizon of the simulated workload to one week. The frequency for running the overbooking logic was set to 5 minutes. The costs for VM operations were configured as $c_b = c_p = c_r = c_t = 30s$. Providing an estimate for VM operations in a cluster environment depends highly on not only the storage and network configuration but also on the target VM memory usage. The 30s estimate should be viewed as a conservative estimate in the context of a cluster using fast networked storage to provide the VM images and VM instances using 1 GB of memory. This assumption removes the need to model migration overhead when resuming VMs on different nodes.

Executing the scenario without overbooking logic results in a mean CPU utilization of 69.4% during a total of 67,200 workload hours. Every test consist of three parameters: available overbooking slots, upper- and lower bound. These are chosen in function of the relatively high average utilization on the simulated cluster. The number of overbooking slots was taken to either be 1, 2 or 3. We varied the upper and lower bounds in increments of 5 between [85, 95] and [60,80] respectively. Since each CPU core can maximally account for a utilization of 12,5%, the minimum difference between lower and upper bound is taken to be 15%. Relaxing this constraint will often result in immediately suspending the added VM once it becomes active.

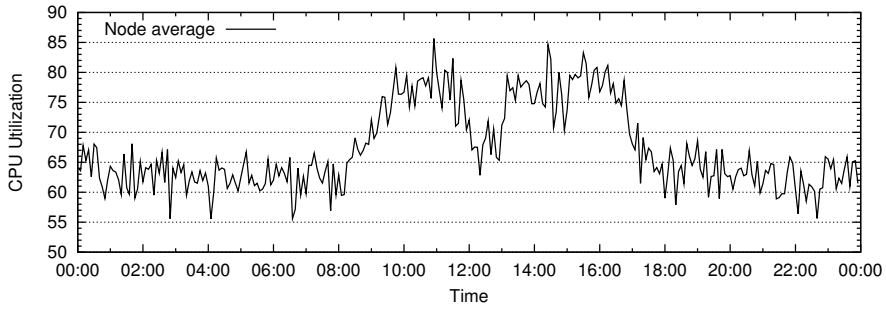

Figure 3.4: Sample load pattern on a single eight core node during a weekday.

3.4.2 Results

The outcome of the experiments is gathered into Tables 3.1-3.3, each containing the results of the test performed for a set number of overbooking slots. The first column contains upper and lower bounds. The third column shows the average utilization achieved when the overbooking logic is active. The average utilization of 69.4% achieved without overbooking, increases to more than 87% for the scenario with three slots, a lower bound of 80% and upper bound of 95%. A conservative bound configuration of 85-60 using a single overbooking slot, leads to a utilization of 73.7%. The fourth column contains the hours of workload that have executed within overbooked low-QoS VMs. This total does not include any VM operation overhead, only active VMs can contribute to the total. The fifth column shows the number of VM suspensions. The second to last column contains the number of *degradation points* if the results are interpreted without any overhead. Degradation points are all overbooking time steps where a total load was recorded that would significantly impact the high-QoS VMs. The last column contains the number of degradation points taking into account a 5% margin. The last column is introduced as a reference points, considering that performance interference is likely to occur somewhere before 100% CPU utilization.

From the results in Tables 3.1, 3.2 and 3.3 we can reach some initial conclusions with respect to the parameter variations and their results. For the purpose of this discussion we will refer to the difference between upper and lower bounds as the overbooking *window size*. *Negative effects* are considered to be a combination of increased suspensions (and the resulting resumptions) and an increase in the number of degradation points at both 95 and 100%.

We will first look at the impact caused by the number of available overbooking slots. The number of slots has a limited influence on the average utilization in the scenarios with the lowest bound values i.e. combination using an upper bound of 85%. This can be attributed to the average utilization without overbooking, which is already relatively high at 69.4%. A single low-QoS VM has the potential to increase the average utilization by 12.5%. When the upper bounds is set comparatively low, this often only allows for a single overbooked VM. When the upper bound is increased, more interesting results can be observed. Moving from a single overbooking slot up to two, higher utilization levels can be observed when using similar bound values. Comparing the achieved utilization levels with the corresponding negative effects, we find that using two slots allows higher utilization levels with a smaller tradeoff regarding negative effects . Increasing the slot number to a maximum of three overbooked VMs no longer results in significant performance increases. Using the same bounds we achieve similar utilization levels while producing identical or increased numbers of negative effects. A higher maximum increase in utilization can be achieved, but not without incurring a disproportionate increase in the number of negative effects. It would appear that using two overbooking slots is the most appropriate setting for this workload distribution.

A detailed side by side comparison of Tables 3.1-3.3 allows us to detect several trends which occur across slot and bound values. Two trends deserve further discussion, namely the effects of *increasing the lower bound* with regard to a fixed upper bound and *increasing the upper bound* with regard to fixed window sizes.

Bounds	Slots	Utilization	Hours	Suspends	> 100%	> 95%
85 - 60	1	73.7	3297.79	405	0	1
85 - 65	1	75.19	4434.55	1125	0	3
85 - 70	1	76.36	5338.98	3443	0	17
90 - 60	1	74.57	3966.83	191	0	10
90 - 65	1	76.3	5290.09	406	0	25
90 - 70	1	77.42	6150.38	1033	0	62
90 - 75	1	78.28	6806.58	2508	6	195
95 - 60	1	75.25	4491.03	144	3	144
95 - 65	1	77.29	6043.97	234	7	234
95 - 70	1	78.45	6936.92	383	13	384
95 - 75	1	79.17	7478.31	660	22	660
95 - 80	1	79.66	7864.45	1305	64	1305

Table 3.1: Results using multiple bound combinations and one overbooking slot.

Bounds	Slots	Utilization	Hours	Suspends	> 100%	> 95%
85 - 60	2	73.86	3418.20	490	0	1
85 - 65	2	75.81	4909.43	1839	0	9
85 - 70	2	77.83	6457.53	6797	3	81
90 - 60	2	75.07	4341.34	243	0	15
90 - 65	2	77.59	6277.52	746	1	49
90 - 70	2	79.87	8020.48	2507	7	196
90 - 75	2	82.13	9760.40	7461	49	868
95 - 60	2	75.84	4934.74	176	7	177
95 - 65	2	79.37	7639.71	370	18	371
95 - 70	2	82.21	9813.01	888	43	888
95 - 75	2	84.45	11528.12	2207	135	2208
95 - 80	2	86.16	12845.09	5498	577	5498

Table 3.2: Results using multiple bound combinations and two overbooking slots.

Bounds	Slots	Utilization	Hours	Suspends	> 100%	> 95%
85 - 60	3	73.88	3429.17	476.0	0.0	1.0
85 - 65	3	75.81	4915.34	1866.0	0.0	10.0
85 - 70	3	77.84	6468.15	6871.0	6.0	94.0
90 - 60	3	75.0	4289.79	242.0	0.0	16.0
90 - 65	3	77.57	6266.12	757.0	1.0	51.0
90 - 70	3	79.89	8047.17	2619.0	13.0	211.0
90 - 75	3	82.35	9921.47	8502.0	107.0	1190.0
95 - 60	3	75.88	4964.69	172.0	8.0	173.0
95 - 65	3	79.38	7643.12	376.0	18.0	376.0
95 - 70	3	82.36	9926.95	966.0	53.0	966.0
95 - 75	3	84.92	11885.76	2898.0	246.0	2899.0
95 - 80	3	87.33	13733.74	8838.0	1376.0	8838.0

Table 3.3: Results using multiple bound combinations and three overbooking slots.

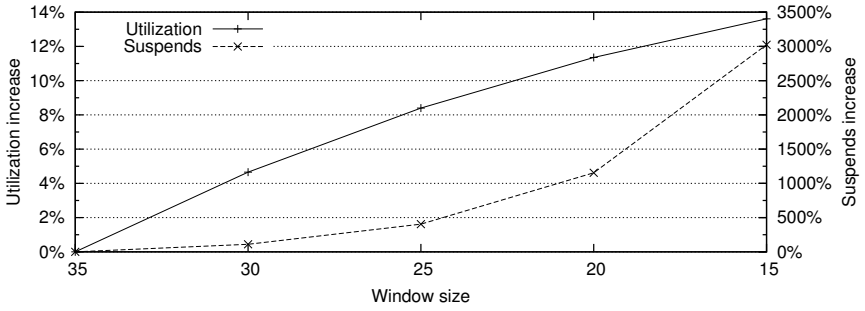

Figure 3.5: Increase in utilization and suspensions when using 2 overbooking slots and an upper bound of 95. The lower bound is increased to decrease the overbooking window.

Increasing lower bounds: The first trend is the effect obtained by increasing the lower bound and keeping all other parameters constant. This results in utilization gains that slowly decrease per step. At the same time we find there is an exponential increase in negative effects. This is illustrated in figure 3.5, the lower bound is increased in steps of 5 from 60 to 80 creating corresponding overbooking windows [35:15]. The results show that although increasing the lower bound results in increased utilization gains, these come at an increasingly higher cost. Figures 3.6 and 3.7 further show that this effect is present in all window, upper bound combinations.

Increasing upper bounds: Increasing the upper bound under a fixed window size results in a linear increase in utilization (see Figure 3.6) while suspensions (and degradation points) remain at roughly the same magnitude (see Figure

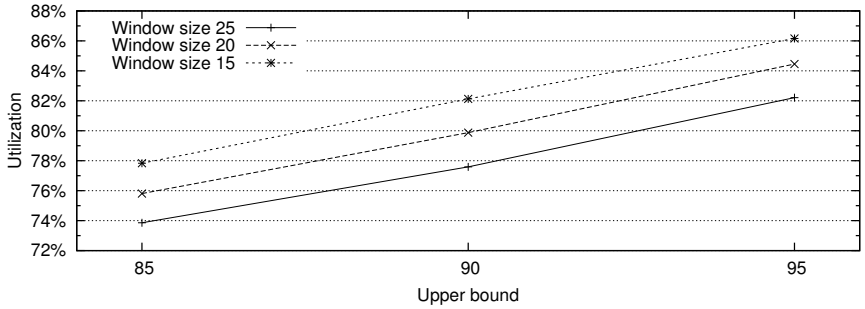

Figure 3.6: Utilization with two overbooking slots and varying upper bounds.

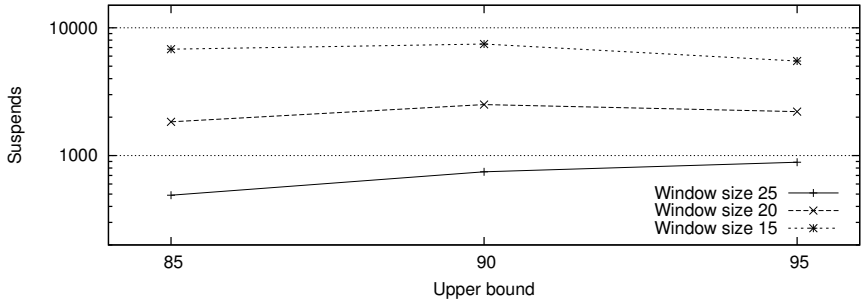

Figure 3.7: Suspensions with two overbooking slots and varying upper bounds and windows.

3.7). From these results we find that choosing a higher upper bound will increase utilization while having a limited impact on the negative effects of overbooking. Of course, these negative effects do not include any potential performance interference which might occur due to the relatively high amounts of CPU usage.

In summary, we find that selecting a correct number of *overbooking slots* is an important part of achieving optimal results. There is a tipping point where extra slots will only add negative effects without additional gain in utilization. We also find that increasing the *lower bound* has diminishing effects on utilization gains while negative effects increase exponentially. On the other hand, increasing the *upper bound* in our current simulator does not add negative effects while utilization displays a steady increase. This leads us to believe that a correct upper bound will most likely depend on limiting factors not yet explored in this research[3]. We can however conclude that the upper bound should be placed as high as possible. Depending on the number of negative effects an administrator is prepared to allow, an optimal set of bounds can be chosen to maximize utilization.

[3] In multi core systems with more VMs than cores, performance degradation will occur somewhere before total utilization hits 100%.

3.5 Related Work

To deal with underutilization in batch queuing systems, backfilling techniques such as EASY [**?**] are often used. Jobs can jump ahead in the queue if they do not delay the start time of the first job in the queue. Conservative backfilling approaches [**?**] require that upon a backfill operation, no job in the queue is delayed in order to maintain fairness. A problem with these approaches is their reliance on user estimates of job runtimes which are often incorrect [**?**]. Several techniques have been proposed to model this runtime in order to tackle this problem [**?**, **?**, **?**].

Aside from backfilling, overbooking of resources is another technique to deal with underutilization. The scheduler deliberately overbooks resources in order to deal with jobs that do not use their allocated resource share fully. Sulistio et. al [**?**] developed a resource overbooking scheme for a setting in which resource reservations are made on a grid infrastructure. Whereas our work hinges on the exploitation of the volatility of VM workloads, their model attempts to deal with the binary case wherein reservations are not used at all or are canceled. They use a richer model for the cost of overbooking by introducing a penalty model that is linked to a renumeration, whereas we only consider the number of performance degradation points the schedule generates. In future work, it would be interesting to including such an application-specific penalty model to diversify the loss of value an application faces if it is subject to a degradation in performance.

An approach to overbooking non-preemptive workloads in a non-virtualized setting was proposed by Urgaonkar et al. [**?**]. They demonstrated that controlled overbooking can dramatically increase utilization on shared platforms. Resource requirements are based on detailed application profiling combined with guarantees requested by application providers. The profiling process requires all applications to run on a set of isolated nodes while being subjected to a realistic workload, this workload generates a set of parameters that must be representative for the entire application lifetime. Instead of pro-actively managing overbooking, application placement is based on a set of constraints and a probability with which these constraints may be violated.

Perhaps somewhat surprisingly, workload traces from the LCG-2 infrastructure, which supports the data processing of CERN's Large Hadron Collider, have shown that as much as 70% of the jobs run by a Tier-2 Resource center in Russia use less than 14% of CPU-time during their lifetime [**?**]. On the other hand, 98% of the jobs use less than 512MB of RAM. Cherkasova et al. thus investigate the potential of running the batch workloads in VMs and overbooking grid resources to increase utilization. The authors conclude that the use of virtualization and multiplexing multiple VMs on a single CPU core allows for a 50% reduction in the required infrastructure while rejecting less than 1% of the jobs due to resource shortage.

Birkenheuer et al. [**?**] tackle underutilization for queue-based job scheduling by modeling the probability that a backfill operation in the job queue delays the execution of the next job due to bad user runtime estimations or resource failure. A threshold is defined on this probability to decide whether a job can be used for backfilling. Birkenheuer et al. report on a 20% increase in utilization on a schedule for a workload trace of a 400 processor cluster. Their work is however not adopted to the specifics of virtual machine scheduling and only considers a single-processor case.

At the level of the VMM, priorities and weights can also be assigned to VMs such that high priority workloads maintain their resource share in the presence of low priority loads [**?**]. The VMM scheduler operates in time quanta that are in the order of tens of milliseconds to ensure the system allocates resources under the configured allocation constraints. Our approach differs from this in that we suspend virtual machines so that their memory pages can be reclaimed by other VMs. Although memory overcommitment is possible in popular VMMs such as Xen, HyperV and VMware, this can result in noticeable performance degradation if the VMs actually require the overcommitted memory [**?**, **?**].

from

3.6 Conclusion

We have introduced a scheduling algorithm which multiplexes low- and high-QoS workloads on a virtualized cluster infrastructure in order to increase the infrastructure's utilization through overbooking. By monitoring the difference between available resources and actual requirements of high-QoS workloads in terms of CPU load, an opportunity to add low-QoS workloads to a cluster node can be identified. We introduce a limited set of parameters in our scheduling policy so that a flexible tradeoff can be made between maximization of infrastructure utilization and workload interference. The results obtained from initial testing show that depending on the requirements, optimal parameters can be selected that significantly increase utilization while causing limited interference with high-QoS workloads. We identified general trends in the system's performance through parameter tuning and identified a number of guidelines to determine an optimal parameter setting.

CHAPTER 4

Model Driven Scheduling for VM Workloads

This chapter is based on "Black Box Scheduling for Resource Intensive Virtual Machine Workloads with Interference Models" [?]

Abstract

In this chapter, we address the issue of unexpected variances in performance due to unmanaged multiplexing of resource intensive VM workloads. A solution is presented using performance models based on the runtime characteristics of virtualized workloads. A set of resource intensive workloads is benchmarked with increasing degrees of multiplexing. Resource usage profiles are constructed using the metrics made available by the Xen hypervisor. Based on these profiles, performance degradation is predicted using several existing modeling techniques. In addition, we propose a novel approach using both the classification and regression capabilities of Support Vector Machines. Application clustering is used to identify several application types with distinct performance profiles. Finally, we evaluate the developed performance models by introducing several new scheduling techniques. We demonstrate that the integration of these models in the scheduling logic can significantly improve the overall performance of multiplexed workloads.

4.1 Introduction

Virtualization has become a widespread technology used to abstract, combine or divide computing resources in order to allow resource requests to be described and fulfilled with minimal dependence on the underlying physical hardware. Using virtualization, an application and its execution environment can be managed as a single entity, a *virtual machine* (VM) [**?**], of which the configuration can be captured in a single file, a virtual machine *image*. These virtual machine images can be deployed in a hardware-agnostic manner on any hardware that hosts a compatible *hypervisor* or *Virtual Machine Monitor*. The hypervisor is a software component that hosts virtual machines, also referred to as *guests*. This software layer abstracts the physical resources from the virtual machines by providing a virtual processor and other virtualized versions of system devices such as I/O devices, storage, memory, etc. [**?**, **?**]. Hypervisors thereby offer flexibility in partitioning the underlying hardware and ensure some degree of isolation between the different virtual machines sharing these resources.

Although the hypervisor provides adequate isolation on many levels (e.g. security, faults, ...) *performance interference* can still be an issue, particularly with resource intensive workloads [**?**]. Each virtual machine is allocated a subset of the available resources and requires the hypervisor's cooperation to complete certain tasks (e.g. disk or network I/O). When multiple VMs share the same hardware, bottlenecks can occur both on the hardware as well as the hypervisor level. This is partially a consequence of the isolation between the hypervisor and guests, causing multiple guest schedulers to work independently without knowledge of each other. When compared to a single operating system running comparable workloads this can cause sub-optimal results [**?**]. The evolution towards an ever increasing number of CPU cores per server and relatively slower gain in I/O performance poses additional challenges. Resource contention problems will likely become even more important in the future as more VMs share the same hardware in an effort to increase utilization. Improvements in VM scheduling and the reduction of virtualization overhead can partially mitigate these issues. In the context of datacenters where VMs can be easily migrated between hosts, the problem can be addressed at a higher level through intelligent scheduling. This approach requires more insight in the resource consumption of individual workloads and their impact on other workloads. Using this information, a more informed scheduling decision can be made which reduces the impact of resource contention.

In this chapter, we build an interference model providing the information needed to automate such scheduling decisions. Using a combination of tools, accurate metrics are obtained from a Xen-based [**?**] hypervisor hosting multiple VMs on a multi-core system. A combination of CPU usage information, cache hit/miss rates and various I/O metrics are obtained for each VM. A set of diverse applications is benchmarked in two different setups. First, we benchmark all applications in a linear scaling scenario where slowdown is calculated when multiplexing VMs with identical workloads. Interference will likely be highest when concurrent applications use similar resources. Therefore, performance prediction based on the linear scaling data can provide an upper bound on the performance degradation due to interference. Second, we benchmark all applications in a pairwise manner to obtain resource contention information for mixed workloads. To provide reference points

for less resource intensive workloads, a number of benchmarks were modified to require fewer resources. In order to reduce the required application set and model complexity, network usage was not yet taken into account, and VMs were assigned a single virtual CPU. An approach commonly used in related work [**?**, **?**, **?**].

We evaluate several techniques to predict performance and classify applications. A novel approach using both the classification and regression capabilities of Support Vector Machines is presented and evaluated. Using K-means clustering, we automatically define application types which have distinct behavior under contention. The scheduling potential of performance predictions and application types is evaluated using three new scheduling techniques. The scheduling problem tackled in this contribution comprises the placement of a static set of applications known a priori on a fixed number of servers. Additional complexity in terms of scheduling methodology, such as online arriving applications or admission control, is not taken into account, and is left for future work. Finally, we compare a coarse-grained scheduling technique based on application types to a more fine-grained prediction-based approach. We demonstrate that the use of performance predictions in the scheduling logic can significantly increase the overall performance of multiplexed workloads.

Previous work has provided important insights and advances regarding performance modeling and application classification in virtualized environments. Nonetheless, several key limitations can be found in the most closely related efforts. First, a relatively small number of application profiles is used to validate prediction techniques [**?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**]. Most use between one and five applications, potentially augmented by using different configuration options. This is a consequence of the significant effort required to obtain a more diverse training set. Second, application sets are restricted to only include benchmark applications limited by resource availability [**?**, **?**, **?**, **?**]. Our experiments show that models trained using only these application types are not reliable when used to predict the performance of applications with less strenuous resource requirements. Third, models are trained and evaluated with a limited number of concurrent workloads [**?**, **?**, **?**, **?**, **?**]. Benchmarking application sets using more than two VMs results in an exponential increase in possible configurations. However, with the increasing number of CPU cores in modern servers this restriction can reduce the relevance for practical applications. Fourth, models are often used to predict and optimize application performance by dynamically reconfiguring the resources allocated to a VM [**?**, **?**, **?**, **?**, **?**, **?**, **?**], whereas our approach models performance interference using fixed VM resource allocations. Instead of allocating additional resources, we reduce interference using high-level VM scheduling. Finally, to the best of our knowledge the combination of classification and performance interference modeling has never been analyzed by developing and evaluating a black-box VM scheduling algorithm. In this chapter we address all these elements in a single body of work. A broader and more detailed comparison with related efforts can be found in Section 4.2.

Our contributions can be summarized as follows:

- The predictive capacity of several system level metrics is evaluated with regards to the slowdown experienced by multiplexed workloads. Experiments evaluate up to nine concurrent VMs on an octa-core server.

- An extensive and diverse set of representative virtualized workloads is profiled. An uncommon feature is the evaluation of workloads not limited by resource constraints.
- Both previously tested and new non-linear modeling approaches are compared on multiple training sets.
- A novel approach using Support Vector Machines (SVM) for both classification and regression analysis is suggested to solve limitations encountered with existing modeling solutions.
- Practical applications are demonstrated by implementing and evaluating several prediction-based scheduling algorithms. The value of adding fine-grained application profiles is demonstrated.

The remainder of this chapter is organized as follows: Section 4.2 discusses existing work and recent developments in the field. An overview of the performance modeling approach is described in Section 4.3. Section 4.4 details the challenges involved in obtaining accurate metrics. In Section 4.5 we describe the benchmarks and frameworks used to gather our performance results, and our experimental setup. Several modeling techniques are explored in Section 4.6, followed by an introduction on application clustering in Section 4.7. Integration of the performance models in scheduling algorithms is discussed in Section 4.8. Our final conclusions can be found in Section 4.9.

4.2 Related Work

One of the efforts closely related to our current approach can be found in [**?**]. Koh et al. consider the impact of performance interference on competing applications in a virtual environment. They collect various system-level workload metrics and perform both application clustering as well as slowdown prediction using two concurrent VMs. Our approach differs in three major aspects. First, their experimentation was limited to two VMs on a dual-processor machine. Second, their regression analysis was limited to basic linear regression, which they show to be less accurate than Weighted Means. We have extended our modeling to non-linear regression efforts by using Support Vector Machines. Finally, we also investigate how scheduling decisions can be improved through the application of performance interference models.

Hoste et al. [**?**, **?**] also used program similarity to predict the performance of profiled applications. However, their focus was on the evaluation of new processor architectures instead of performance interference. In [**?**], predictions were made by taking a weighted average over the performance numbers of the benchmarks in the neighborhood of the application of interest. The major differences with our method can be found in the metrics used for prediction and the result they were attempting to reach. Instead of system level characteristics their approach uses microarchitecture independent metrics, i.e., the characteristics are independent of cache size, processor core configuration, etc. These metrics were then used to predict the performance of a single application on different platforms. Our approach is focused on predicting interference effects on single platform for unknown applications.

Considerable work is also being performed on analyzing the performance characteristics of representative server consolidation workloads [**?**, **?**, **?**, **?**, **?**, **?**]. In [**?**] Casazza et al. introduce vConsolidate, a benchmark that uses applications often found in virtualized datacenters. It consists of a compute intensive workload, a web server, a mail server and a database application running simultaneously on a single platform. Unfortunately vConsolidate development and maintenance has ended and external publication of results is no longer allowed. Due to the reliance on a small set of predetermined applications, it would not have been suitable to validate the techniques proposed in this chapter.

In the work of Lv et al. [**?**] an extensive analysis is made on a system capable of 64 hardware threads using vConsolidate. They found that in current many core systems the I/O backend has become a performance bottleneck. A second observation demonstrated that the current generation of resource schedulers did not provide an adequate amount of fairness when consolidating some compute intensive workloads. Their suggested solution is based on a guest-level scheduler that is aware of the internal workload, offering finer control over priorities. Iyer et al. [**?**] have also used vConsolidate to measure the effects of shared platform resources on virtual machine performance. A simple model is presented predicting the performance loss due to consolidation based on core and cache contention. Their model consists of a simple heuristic using a limited set of metrics, a large portion of the interference is attributed to unknown virtualization overheads. A comprehensive modeling approach and validation is left for future work. They propose a solution wherein a transition is made from VMs to a virtual platform architecture (VPA) that enables transparent shared resource management through architectural mechanisms for monitoring and enforcement. A survey of several virtualized consolidation benchmarks can be found in [**?**].

In [**?**, **?**], Apparao et al. use vConsolidate to analyze the impact caused by moving an application from a dedicated environment to a consolidated setting. This information is used to make a preliminary performance model using cache and core interference combined with virtualization overhead. They use a limited number of workload and system characteristics to develop a set of simple equations. Wood et al. [**?**] use a set of three micro benchmarks to create a general regression based model that maps native system usage onto virtualized performance. They validate their approach using two web applications and two different hardware platforms.

Zhang et al. [**?**] have used system-level metrics for application classification. Their approach is based on Principal Component Analysis and the k-Nearest Neighbor classifier. VM workloads are placed into one of five categories: CPU-intensive, IO-intensive, memory-intensive, network-intensive and idle. Each category is trained by a single hand picked application representing the key performance characteristics of a class. Our classification approach uses K-means clustering to automatically generate application types by grouping multiple applications. They illustrate that throughput can be increased by co-scheduling different application types, however, they do not propose a scheduling algorithm.

A large body of research has focused on managing virtualized application performance using fine-grained allocation of VM resources [**?**, **?**, **?**, **?**, **?**, **?**, **?**]. Whereas our approach uses virtual environments with fixed resource allocations, these works focus on dynamically reconfiguring the resources allocated to a VM in response to application requirements. Padala et al. [**?**] have designed a feedback control systems

to dynamically adjust the CPU resources allocated to individual tiers in multi-tier applications. Both Pandala et al. [**?**] and Nathuji et al. [**?**] use multi-input, multi output models to capture the relation between application performance and the amount of resources allocated to a VM. These models are used to mitigate VM performance interference effects by allocating additional resources when needed. Xu et al. [**?**] present an approach using controllers at both the VM and resource-pool level. They use fuzzy logic to optimize application performance by dynamically allocating VM resources. Rao et al. [**?**] have introduced VCONF, a reinforcement learning (RL) based approach to automatically configure VM resource allocation. They use a combination of model-based RL algorithms and artificial neural networks (ANN) to optimize both individual VM configurations and system wide performance.

Kundu et al. [**?**] presented an ANN-based iterative modeling technique that predicts application performance at a given allocation level of partitionable resources. Input data is gathered by enforcing several upper limit combinations of CPU allocation, memory usage and disk bandwidth. They hypothesize that linear regression might not be suitable for performance predictions of virtualized applications, using ANN instead. This is attributed to the complex non-linear relationships between inputs and outputs. The resulting model is used to make a prediction about the minimum amount of resources required by a VM to achieve SLA guarantees. However, their approach is limited to predicting the performance of known applications using variations of previously profiled parameters e.g. reducing the disk bandwidth. In [**?**] they further extend this approach and improve their results by adding SVM-based models and K-means clustering to create sub-models. Their experiments indicate that creating multiple sub models can substantially reduce prediction errors.

Attempts have also been made to increase the performance isolation between virtual machines. Most notably Gupta et al. [**?**] who implemented XenMon to monitor per VM resource usage, including the work done in separate device driver domains (DDD). Gupta et al. use this information to implement a VM scheduler based on the aggregated CPU usage information from guests and their DDD's. A separate control mechanism is used to enforce a limit on CPU time consumed by a DDD on behalf of a particular guest. Their approach differs from our solution since it only takes into account interference caused by CPU contention.

The challenges posed by resource contention in single operating systems show interesting similarities to the consolidation of virtualized workloads. The work of Chandra et al. [**?**] demonstrates and models the impact of L2 cache misses on co-scheduled threads in a multi-processor architecture.

4.3 Approach

In this section, an overview is given of the challenges faced when modeling our problem domain and the solutions devised to address these problems. The approach described in this chapter is a compromise between achieving an accurate and useful model and having a reasonable and repeatable method for obtaining the required data. Building a performance model requires a representative set of applications, resulting in a large set of benchmarks for each hardware configuration. Server hardware is also evolving towards systems containing many CPUs that are capable of supporting large numbers of concurrent VMs. Given the large application set and the increasing number of VMs that can be co-scheduled, it is clear that benchmarking

all possible configurations is not feasible. A subset of benchmarks is required that can provide a useful model while only requiring a fraction of all possible setups to be considered.

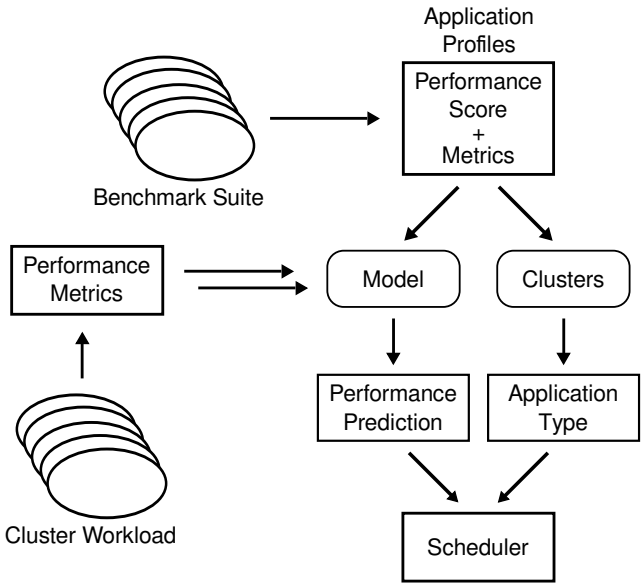

Figure 4.1: Schematic overview of the steps involved in the performance modeling approach.

To model the impact VM workloads have on each other, a measure is needed that captures the *sensitivity* of the workload to increased amounts of resource sharing. One could also expect there to be a different impact depending on the type of resources being shared. For example, scheduling a CPU-limited and disk-limited application together would likely have a relatively low impact on performance. A method is needed to automatically determine the *application type* and which other workloads are most suited for co-scheduling.

To measure the impact of resource sharing, all applications are benchmarked in a linear scaling scenario. Slowdown is calculated when multiplexing VMs with identical workloads. Interference is likely to be highest when applications use similar resources, providing an upper bound on the performance degradation due to interference. A base measurement is taken when running a single VM workload, the number of concurrent workloads is then increased one at a time until there is more than one VM per physical CPU core. Using the measurements from these benchmarks as training data, several models can be built to predict the maximum slowdown of unknown *application profiles*. Each profile consist of several metrics describing the usage of CPU cycles, L2 cache hit rate and disk usage. Performance predictions can be made using the base measurements and the performance scores recorded for high degrees of workload parallelization.

To determine the impact of application types on resource contention, application clusters are formed grouping similar application profiles. To analyze these clusters, all applications are benchmarked in a pairwise manner. The clusters can

be used to easily identify new applications and to determine the relationship between groups. A ranking can be made using the average slowdown incurred when members of different clusters are executed together, providing an indication of the applications most suitable for co-scheduling.

A schematic overview of the separate steps required for the proposed approach can be found in Figure 4.1. The results from an extensive benchmark suite are analyzed to acquire performance scores and metrics for the desired platform. This data is then transformed into a predictive model and application clusters are defined. When a workload is presented for scheduling, the first step is to gather the required performance metrics. Using this data, performance under contention is predicted and the application type determined. This information is added to the scheduler who can use it to determine an efficient placement for all VM workloads.

4.4 Profiling

Profiling is a crucial aspect for gathering useful system-level characteristics concerning CPU, cache and disk usage. We decided to use three existing packages: XenBaked[**?**] (CPU), Xenoprof[**?**] (cache) and iostat (disk).

To get accurate CPU metrics we experimented with the XenMon monitoring tools. Several metrics can be gathered for each VM's virtual CPU (vCPU) e.g. CPU cycles *gotten* (time used for calculations) and CPU cycles *blocked* (idle or vCPU waiting on an event). Events generated by the Xen scheduler are placed into a shared trace buffer where they can be retrieved by XenBaked. XenMon then parses this information into a user friendly format. Unfortunately, the observed measurements were inaccurate when benchmarking on multi-core systems, particularly when the CPU reached 100% utilization. Analysis showed that the problem was caused by out-of-order events and the lack of trace data for vCPUs that were rarely scheduled out. To get accurate results for events that take place during short time intervals, each event is time stamped using the number of clock cycles that have passed. Timestamps are calculated using the cycle count from the current physical CPU. Due to clock drift not all physical CPUs will have performed the exact same number of cycles, this can cause negative durations for short events. Having multiple physical CPUs also means that a busy vCPU can occupy the same physical CPU during long intervals. XenBaked did not take this possibility into account, no events were logged and utilization appeared to be zero. For our tests, we created a modified version of XenBaked that resolves these issues.

Xenoprof is an extension to OProfile[**?**] which uses built-in CPU performance counters to obtain data about executing applications and processes. The use of built-in counters allows OProfile to accurately gather detailed information for a limited set of metrics without generating a significant overhead. Xenoprof interfaces with the Xen hypervisor to set counters and attribute events to separate VMs. In our tests we use this information to calculate the L2 cache *hit/miss ratio* for each VM.

We use the Iostat tool to retrieve I/O statistics from the paravirtualized guest kernels. For our experiments we monitor averages for the following disk metrics: read and writes per second (*r/s* and *w/s*), read and write throughput per second (*rMB/s* and *wMB/s*), average time for I/O requests issued to the device to be served (*await*) and amount of time the disk was busy servicing I/O requests (*%util*).

Metric	Measurement
gotten	Percentage of CPU cycles used for processing
blocked	Percentage of CPU cycles spent idle or waiting on an event
hit/miss	Percentage of L2 cache requests that missed
r/s	Average disk reads per second
w/s	Average disk writes per second
rMB/s	Average disk read throughput in MB per second
wMB/s	Average disk write throughput in MB per second
await	Average time needed for disk I/O requests to be served
%util	Percentage of time the disk spent servicing I/O requests

Table 4.1: Overview of the metrics obtained from profiling.

An important aspect of profiling is the minimization of the incurred overhead. To quantify this overhead we used the Linpack [**?**] performance evaluation benchmark. Results were obtained by running Linpack on all CPU cores both with and without the profiling platform enabled. Our custom Xenbaked parsing solution, Xenoprof and iostat caused respective overheads of 0.28%, 0.25% and 0.20%. The overhead when gathering all metrics averaged out at 0.60% per CPU core, sufficiently low for our purposes.

4.5 Benchmarking

Development of the benchmarking platform was done using Linpack [**?**], httperf [**?**] and a modified version of Sysbench [**?**]. To obtain a diverse set of application types we added a subset of the benchmarks found in the Phoronix Test Suite [**?**] v3.0.1. The chosen applications represent an even mix between different workload types. The full list of names and descriptions can be found in Table 4.2. Most of the Phoronix-based tests were performed using the the default settings provided. In some cases, a choice was required between multiple configuration options. Dbench was run with both one and six threads, IOzone was benchmarked in read-only mode, and FIO was benchmarked using both file and network traces. The modified version of Sysbench includes extensions allowing SQL and disk benchmarks to be limited to a specified throughput. Due to its ability to be rate limited, Httperf was used as a means to provide an additional benchmark with low requirements for all monitored resources.

Each benchmark was profiled in isolation and in parallel for a minimum of ten minutes. Results were obtained for a parallelization degree of up to nine concurrent VMs executing identical workloads. If benchmarks had different runtimes, the shorter applications were restarted until the longest running application was finished. Each benchmark was repeated a minimum of three times per profiling run. As some long running applications require initial start up procedures e.g. creating test files, some co-scheduled workloads can have performance variances depending on the starting time. If needed, restarting the application continued until the relative standard deviation of the results was lower than 3.5%.

Name	Measurement
AIO-stress	Disk read and write performance in MB/s
Apache	Number of static page requests/s a system can sustain
Build-Apache	Compilation time for the Apache HTTP Server
Build-MPlayer	Compilation time for MPlayer
Compress-Pbzip2	Time needed to compress a file using BZIP2
Crafty	Number of chess moves/s that can be calculated
C-ray	Floating point CPU performance using a raytracer
Dbench	Average disk throughput using a mixture of I/O operations
Dcraw	Time needed to convert RAW NEF image files to PPM images
Encode-FLAC	Time needed to encode a WAV file to the FLAC format
Encode-MP3	Time needed to encode a WAV file to the MP3 format
eSpeak	Time needed to read text and convert to a WAV file
ffMPEG	Time needed to convert a sample AVI to NTSC VCD
FIO	Time needed to complete recorded disk access traces
GMPbench	Arithmetic performance using the GMP library
IOzone	Disk read and write performance in MB/s
Mafft	Time needed to align 100 enzyme sequences
Mencoder	Time needed to convert an AVI file to LAVC
mrBayes	Time needed to perform a Bayesian analysis
Nero2D	Time needed for a 2D TM/TE solver using Open FMM
N-Queens	Time needed to solve N-queens problem using OpenMP
OpenSSL	Signs per second using OpenSSL RSA 4096-bit encryption
Postmark	Timed small-file disk test based on web and mail servers
POV-Ray	Timed rendering using the POV-Ray raytracer
SQLite	Time needed to perform a pre-defined numbers of insertions
Sunflow	Timed rendering using the Java based Sunflow render engine
Unpack-Linux	Time needed to extract the .tar.bz2 Linux kernel package

Table 4.2: Overview of the Phoronix applications used for benchmarking.

Results were obtained using Xen 4.0 with a paravirtualized 2.6.32.24 Linux kernel. All VMs were configured with 1GB RAM using disk images mounted via `blktap2`. To facilitate the interpretation of the results, all VMs were provisioned with a single vCPU. The benchmarking platform consisted of dual Intel Xeon L5335 quad-core CPUs, 16GB RAM and a WDC WD2500YS hard drive (7200rpm, 16MB cache).

4.6 Modeling

In this section, we compare different modeling techniques using several sets of training data. All data is gathered with the previously introduced benchmarking platform. Each data point in the training data consists of an application profile and a performance score. Profiles contain a set of measurements corresponding to the metrics in Table 4.1, collected by monitoring the application in isolation. Performance scores are recorded for several parallelization levels, increasing the number of identical concurrent workloads from two VMs up to a maximum of nine.

gotten	blocked	hit/miss	r/s	w/s	rMB/s	wMB/s	await	%util
17.96%	12.24%	5.03%	1.65	244.35	0.03	5.35	308ms	52.63%

Table 4.3: Sample application profile for the AIO-stress disk benchmark.

Concurrent VMs	1	2	3	4	5	6	7	8	9
Recorded Score	10.8	4.95	3.29	2.36	1.93	1.58	1.28	1.10	1.02
Normalized Score	1	2.18	3.29	4.58	5.58	6.84	8.42	9.80	10.63

Table 4.4: Sample performance slowdown scores for the AIO-stress disk benchmark.

Scores are normalized to one, the result from a standalone run on the test platform. A sample application profile and normalized performance scores can be found in Tables 4.3 and 4.4.

Each modeling approach is used to predict the potential slowdown of unknown applications for several degrees of contention. A separate model is constructed for each parallelization level. The model inputs and output can be described as follows:

Inputs : Models are trained using subsets of the available application profiles and the accompanying slowdown scores for the target parallelization level. E.g. a model predicting for a parallelization level of seven would be trained using the metrics in Table 4.3 and a slowdown score of 8.42 (Table 4.4).

Output : Using the profile of an unknown application, a predicted slowdown score is calculated for the target parallelization level.

As the number of concurrent VMs rises, applications develop initial or more substantial slowdown. Predicting slowdown for high levels of parallelization can provide an estimate for the maximum slowdown on the target platform. We compared three types of prediction techniques: Weighted Means, Linear Regression and Support Vector Machines. The section ends with a discussion on the optimal combination of training data and prediction technique.

4.6.1 Weighted Means

Weighted Means (WM) has previously been used for similar applications in both [**?**] and [**?**]. A multidimensional space is constructed where each dimension corresponds to a workload metric of the benchmarked application. To calculate the predicted score of an unknown application, a weighted average is taken over the measured performance scores of similar applications. The similarity between two applications is determined by the Euclidean distance in the multidimensional space. The weight associated with each neighbor depends on the relative distance to that neighbor. Performance scores from nearer neighbors have a higher influence on the final prediction. The N closest results from all known data points are used to calculate the weighted average.

Each predicted score Pr is a weighted combination of the normalized performance scores n_i. The similarity s_i for an application is defined as the inverse of the Euclidean distance. The weight w_i assigned to each value can then be calculated as seen in Equation 4.1. We set $N = 3$ to calculate the predicted score using Equation 4.2.

$$w_i = s_i / \sum_{i=1}^{N} s_i \tag{4.1}$$

$$Pr = \sum_{i=1}^{N} w_i * n_i \tag{4.2}$$

4.6.2 Linear Regression

Linear regression has been applied to similar problem domains in previous work [**?**, **?**]. It was shown to not be particularly suited. This can be explained by the mostly non-linear relationships between inputs and outputs. For the sake of completeness, we have evaluated the technique with our more extensive data set as it is a commonly used modeling approach.

Linear regression describes the relationship between a dependent variable y and one or more explanatory variables x. A predictive model is made under the assumption that the behavior of the dependent variable can be captured using a linear function. With the normalized scores y_i as the dependent variable, let $x_i = x_{i1}, ..., x_{in}$ denote the n explanatory variables obtained through the workload metrics. With $\beta = \beta_0, ..., \beta_n$ describing the corresponding set of regression coefficients. By adding a random error ε_i we can form the linear function in Equation 4.3. The regression coefficients express the expected change in y_i per unit change in x_{ij}. The random error accounts for the discrepancy between observed y_i values and the predicted values.

$$y_i = \beta_0 + \beta_1 x_{i1} + \cdots + \beta_n x_{in} + \varepsilon_i \tag{4.3}$$

A regression model is fitted by determining the coefficients β and the error term ε that minimize the error for the training data. Predictions are then made by applying new measurements x_i to the formula. We constructed a Linear Model (LM) using the *lm* package in R to estimate the unknown parameters with Ordinary Least Squares.

There are several limitations to using basic linear regression, two important issues can be overcome by using Generalized Linear Models (GLMs) [**?**]. The first problem results from the assumption that the dependent variables follow a normal distribution. A second limitation is caused by the requirement that the effect of the explanatory variables must be linear in nature. With GLMs, the response variables can be generated using a wide variety of distributions from the exponential family. A range of *link functions* can then be used to describe the relationship between the dependent variable and the linear model. We evaluated this regression technique using the *glm* package in R with an Inverse Gaussian distribution and an identity link. The distribution and link function were selected after testing all available distribution and link combinations on several datasets.

4.6.3 Support Vector Machines

The complex relationship between metrics and performance scores suggests that linear approximations will not be able to provide sufficient accuracy. To accurately describe this relationschip, a non-linear model is required. Support Vector Machines [**?**] (SVM) are a type of learning algorithm initially developed to solve the *classification* problem. Later the technique was extended to the domain of regression analysis [**?**]. SVM can model complex real-world problems on data sets that have a high-dimensionality, even with relatively few data points on which to train the model. These characteristics make it highly suitable for our needs. A detailed discussion on the inner workings of SVM can be found in [**?**].

The estimation accuracy of SVM depends on three hyperparameters; ϵ, C and the kernel parameters. The ϵ parameter sets the accuracy requirement for the approximated function, it should be set according to the order of magnitude of the response value. The value of ϵ affects both the model complexity and generalization capability by influencing the number of support vectors and the response smoothness. C determines the penalty associated with constraint violation. The C parameter is used to control the trade-off between model complexity and training error. The kernel parameters that can be manipulated depend on the type of kernel. For our experiments we selected a Gaussian RBF kernel which requires the σ parameter to control the kernel width.

We used the *ksvm* package in R to do both the classification and regression [**?**]. The default predictor was configured using the following parameters: $\epsilon = 1 \cdot 10^{-4}$, $C = 7$ and $\sigma = 2 \cdot 10^{-2}$. These values were obtained by selecting the best results from a grid search over the parameter space.

We encountered issues when training the SVM predictor on data sets that contain a significant amount of applications with low resource usage and no slowdown. The inclusion of this additional training data had a negative impact on the predictions of applications with significant slowdown. Several applications were also wrongly attributed slowdown by the predictor. To resolve this issue, we used the classification capabilities of SVM. A binary classification model uses all the available training data to determine if an application will likely suffer some amount of slowdown. With this prescreening, predictions are only needed for applications when we have reason to assume that they will exhibit slowdown. This assumption means we can build the prediction model using only the training data from applications with slowdown. The end result is more accurate detection of applications without slowdown and better predictions. In the following, we refer to this approach as *Class SVM*.

A popular alternative to SVM in the realm of non-linear statistical models are Artificial Neural Networks (ANN) [**?**]. For our purpose however, there are some limitations which make them less suitable as a regression technique. The biggest problem is the need for large training sets, especially when the data has high dimensionality. Other issues include a greater computational cost and the difficulty to train complex neural networks. Using our target data set, it proved time consuming and challenging to properly train the network. Results varied greatly depending on the subset of the data that was chosen. As this approach was not found suitable for our problem, we decided to omit the ANN predictions from the results section.

Application Mix	HRR Only	Httperf	Rate Limited	Size
T1	✓			30
T2	✓	✓		31
T3	✓	✓	✓	55

Table 4.5: Overview of the different application types in each training data set.

4.6.4 Application Sets

We used the applications listed in Section 4.5 to create a diverse set of training data containing several different workload types. Application profiles consist of a performance score and several metrics describing the usage of CPU cycles, L2 cache hit rate and disk usage. The first objective was to create a sufficiently large collection of benchmark applications that are resource bound by various combinations of the aforementioned metrics. In Section 4.7 we use clustering to automatically divide the training data into application types. Analysis of the clustering results demonstrates that this objective was achieved. The second objective was to add applications with limited resource usage to evaluate and improve the model's ability to predict the performance of this workload type. Results from these experiments are detailed in the following section. All previously listed modeling techniques were evaluated using the following sets of training data.

T1 : High Resource Requirement (HRR) Only - this set contains all the applications listed in Table 4.2 with the addition of Linpack, making a total of 30 applications. We use this set for the initial evaluation because its composition closely resembles the approaches found in related work. All benchmarks have high resource requirements and will tax a single resource or combination thereof to obtain a performance score.

T2 : T1 + Httperf - the second set includes all the previous applications with the addition of Httperf. This combination was used to determine the suitability of the HRR Only training data when trying to detect applications that do not exhibit slowdown.

T3 : T2 + Rate Limited - new "applications" were added using rate limited versions of the Sysbench disk and OLTP benchmarks. Test configurations were selected to ensure an even distribution across the entire resource usage spectrum. The extra training data allows the modeling techniques to determine the tipping point where an application starts exhibiting slowdown. Adding these artificial applications brings the total number of applications in the set to 55.

Both Sysbench and Httperf have several parameters that determine the resource requirements, we used the following configuration options for our benchmarks. Sysbench disk benchmarks using 128kb chunks were recorded for 1, 2, 4 and 6 operations per second. Reads, writes and read/writes were tested in separate runs. OLTP tests were performed on both a read only and a read/write database, processing between 10 and 200 requests per second. Httperf was configured to generate a ~10% load on a single vCPU and had a negligible impact on the other profiled resources. An overview of the different application sets can be found in Table 4.5.

4.6.5 Results

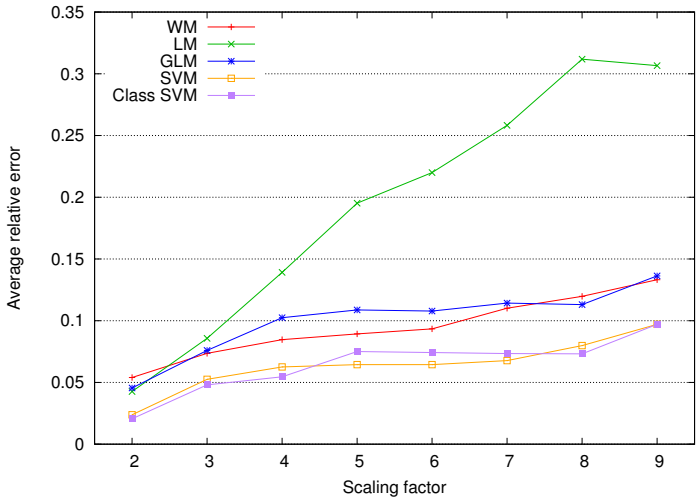

Figure 4.2: Relative prediction errors for each modeling approach using applications from T1.

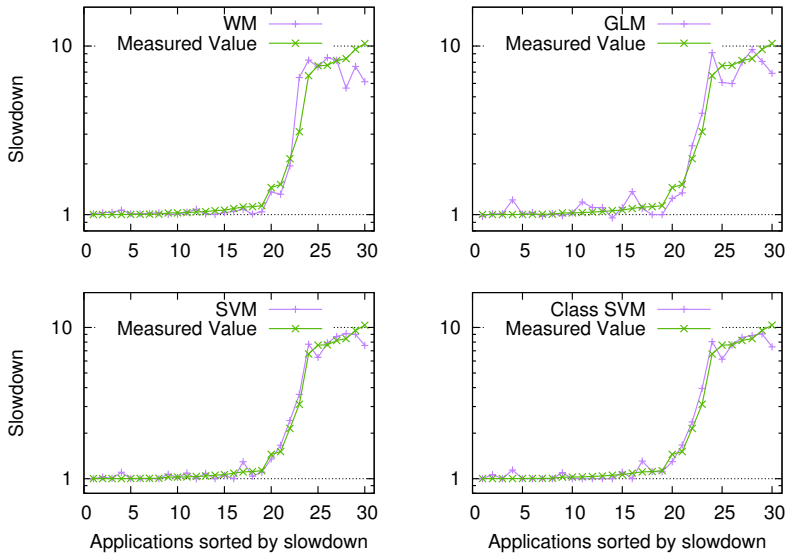

Figure 4.3: Error distribution using applications from T1 with a scaling factor of 7.

We present our results in three consecutive steps according to the training set used to evaluate the predictive capacity of each technique. Predictions are presented for scaling factors of two, where few applications have significant slowdown, up to the overbooking scenario of nine concurrent VMs, where all applications incur

slowdown. The models were trained and evaluated using both the leave-one-out cross-validation (LOOCV) method and K-fold cross-validation (CV). LOOCV creates a model per application, using the remaining profiles as training data. This is repeated until each profile is used once as validation data. Accuracy is determined by calculating the average prediction error across all applications. Despite the high computational cost, this method was chosen due to its optimal use of the relatively limited and diverse training data. One potential downside to using LOOCV is that the results can be overly optimistic. This can be the case when using homogeneous training data where removing a single observation might leave one or more very similar data points in the training set. As our training sets consist of distinct benchmark applications, this should not be the case. To confirm our conclusions, we perform K-fold CV by creating 5 folds in 10 separate runs.

The first evaluation uses applications from set T1, containing only applications with high resource requirements. The average prediction error produced by each technique can be found in Figure 4.2. As expected, we find that the basic linear model does not result in accurate predictions. Somewhat surprising are the low average errors produced by GLM. Looking at the error distribution for GLM in Figure 4.3, we find that a relatively large portion of the average error is caused by prediction errors for applications with limited slowdown. When compared to WM and SVM, GLM seems to have more problems correctly identifying applications with low and average slowdown.

As there is no large class of applications with low resources requirements, the result for both SVM and Class SVM does not differ significantly. The difference between the two never exceeds 1% in either direction. The WM approach produced a noticeably higher average slowdown error when compared to Class SVM, performing up to 4% worse for a scaling factor of seven. The higher average error is caused by less accurate predictions for applications with bigger slowdowns. WM tends to underestimate slowdown, having some difficulty identifying the threshold between moderate and high slowdown. The SVM models appear to be better suited for extrapolating predictions from the sparser sections of the training set.

For these models to have practical applications, they also need to provide accurate results for applications whose performance is not limited by their resource allocation. These non-benchmark applications are likely to require so little resources that they never experience slowdown under any scenario. To test the robustness of the T1 trained models, we added Httperf to our experiments (T2). We used the training data from T1 to predict Httperf performance with a scaling factor of 7, a scenario in which it does not experience slowdown. All techniques mistakenly identified Httperf as an application with significant slowdown. WM provided the most accurate result, still predicting almost 19% slowdown. The other techniques performed even worse, resulting in prediction errors between 300 and 600%. The T1 trained models are not capable of predicting slowdown for applications with limited resource requirements.

An extended set of training data is needed to address this problem, adding a range of applications with low resource requirements and no slowdown. We obtained additional data points using rate limited versions of the Sysbench disk and OLTP benchmarks as described in set T3. The results in Figure 4.4 summarize the effects of using T3 to train the models predicting the applications in T2. The extended training data has significantly reduced the accuracy of the linear and

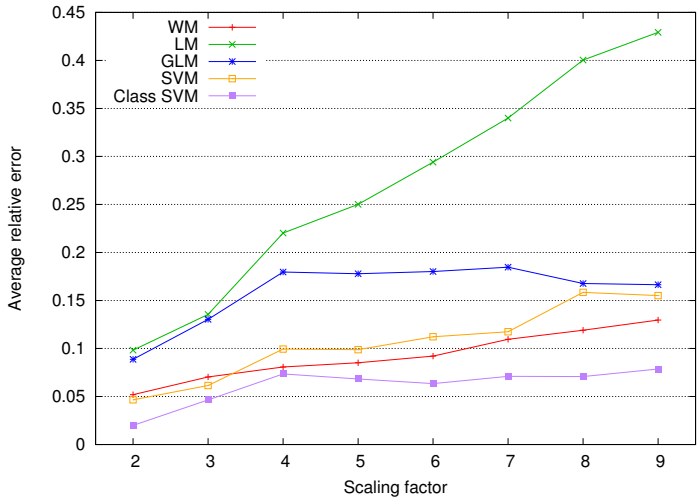

Figure 4.4: Overview of the relative prediction errors on T2 applications using T3 as the training set.

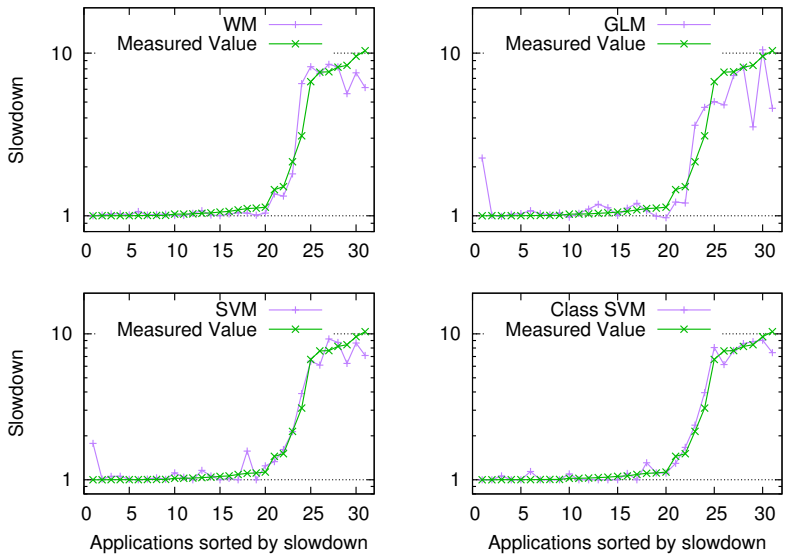

Figure 4.5: Error distribution predicting applications from T2 using T3 as the training set with a scaling factor of 7.

standard SVM predictions. In Figure 4.5 we can see that both GLM and standard SVM still fail to correctly identify Httperf (Application 1 on the X-axis) as an application without slowdown. The results for WM and Class SVM are much more promising. Httperf is now classified correctly as an application without slowdown, while all other predictions maintain similar errors. Average relative error for SVM remained

below 8% for all scaling factors, a reduction of up to 4% when compared to Weighted Means.

If we look at the error distribution in Figure 4.5 we can reach two important conclusions. First, we can see that there are still a few applications where predictions could be improved significantly. As these errors are mostly found on applications with large amounts of slowdown, it is likely that these are the result from a sparseness in the training set. There are relatively few applications with very large slowdowns and this subset of the training data is further divided by the diverse hardware bottlenecks which cause them. Second, and more importantly, we can note that all WM and Class SVM predictions are in the correct order of magnitude. With scheduling applications in mind, correctly identifying the scale of the sensitivity to interference will add valuable information.

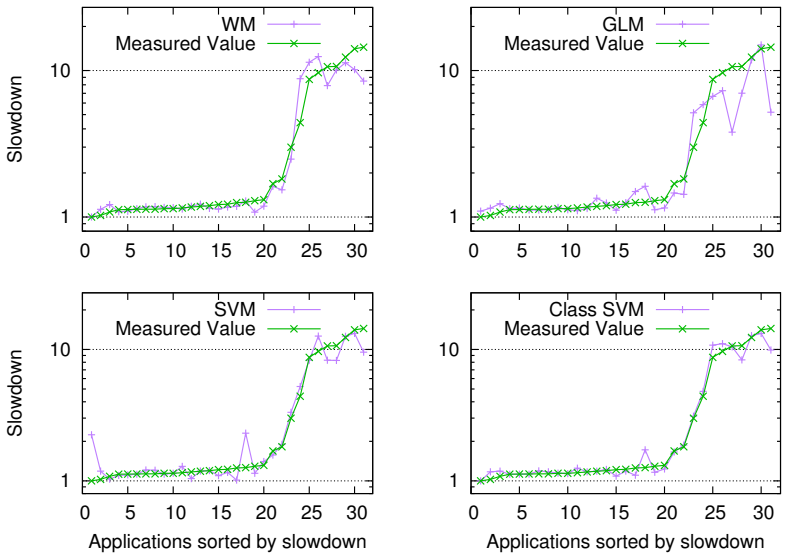

Figure 4.6: Error distribution predicting applications from T2 using T3 as the training set with a scaling factor of 9.

Even with a scaling factor of 7, there are still quite a few applications with limited slowdown. In Figure 4.6 we find the error distribution for a scaling factor of 9, overbooking the 8 CPU server. As a result, all applications except Httperf (Application 1 on the X-axis) experience slowdown. By increasing the slowdown effects we can add a new observation to the previous conclusions. Both WM and Class SVM still correctly identify Httperf. However, both are still inaccurate when predicting the applications closest to the tipping point where slowdown first starts occur. A higher density of training data is needed surrounding this tipping point.

To further validate the conclusions drawn from the LOOCV experiments, we use 5-fold CV in 10 separate runs. T2 is used to populate the validation sets with six applications per fold, T3 is used to provide the training data. In Figure 4.7, we find the results comparing LOOCV and 5-fold CV. As expected, all techniques produce moderately increased average prediction errors due to the smaller set of training

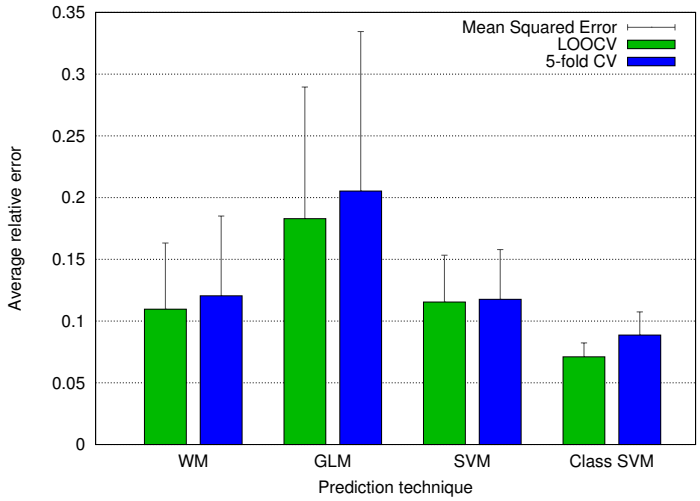

Figure 4.7: Relative prediction errors for both LOOCV and 5-fold CV. Predicting slowdown for T2 using T3 as the training set with a scaling factor of 7.

data. The Mean Squared Error, incorporating both variance and bias, increases only slightly for each technique. Our evaluation using LOOCV does not appear to be overly optimistic. Class SVM still creates the most accurate model, even outperforming the LOOCV results from other techniques. Analyzing the separate K-fold results, we find that accurate models can be constructed using a subset of the available training data. Determining the minimum set of benchmarks required for a robust model is left for future work.

4.7 Clustering

In this section, we evaluate clustering as a method to automatically determine distinct application types. Each application type has a unique performance profile and a specific impact on co-scheduled applications. In the previous section we used predictions to provide a one-dimensional view of interference, an estimate on the upper bound for slowdown. However, the performance of an application can depend on several different resource limitations. Using application clusters we can measure the average slowdown when co-scheduling different application types. This information can provide new insight in the applications types that are ideally suited for co-scheduling and which combinations should be avoided.

Cluster analysis is the process of finding a partitioning or grouping in a set of patterns, points, or objects. The goal is to create disjunct clusters where similarity between group members is maximized while similarity between groups is minimal. The definition of *similarity* can differ significantly and depends on the type of clustering algorithm. K-means [**?**] is one of the more prominent clustering algorithms, producing good results for a wide variety of applications.

The k-means algorithm is a form of centroid-based clustering where a set n of observations X=$\{x_1,...,x_n\}$ is partitioned into k clusters C=$\{c_1,...,c_k\}$. The goal of

the algorithm is to minimize the squared error between cluster members and the empirical mean for all k clusters. If we let μ_k denote the mean for cluster c_k, the error function that needs to be minimized can be found in Equation 4.4.

$$E = \sum_{i=1}^{k} \sum_{x_j \in c_i} \left\| x_j - \mu_i \right\|^2 \tag{4.4}$$

The iterative group forming process can be summarized in the following steps:

1. Provide initial cluster centers or centroids. These can be found by selecting from the observations or by creating artificial data points.
2. Calculate the distance between the data points and each centroid. Groups are formed by assigning members to the closest centroid.
3. Determine the new centroids by calculating the empirical mean for each group.
4. Repeat step 2 and 3 until the centroids do not change more than a predefined threshold in a single iteration.

We used the SciPy [**?**] package to perform the cluster analysis. The current SciPy clustering package only implements the k-means algorithm for the Euclidean distance metric. Research has shown that for high-dimensional spaces other distance metrics such as the Manhattan distance can potentially improve results [**?**]. However, initial experiments with other distance functions did not result in significant clustering differences for our data set. Results are presented for Euclidean distance as it is the most common approach and provided the clusters we were expecting to find.

4.7.1 Clusters

Dividing a data set into suitable groups and correctly categorizing new candidates can be a labor intensive process requiring extensive knowledge of the domain. K-means can simplify this process by allowing us to automatically identify clusters. Assigning new applications to the correct cluster is as simple as determining the closest centroid. The only variable that requires user input is the number of significant clusters we expect to find. If K-means is suited to the problem domain, finding the optimal k value still requires an evaluation of the resulting clusters.

To evaluate K-means, we used the performance metrics gathered from profiling each application in isolation. Clusters were built for k values from two until six. Applications with similar slowdown profiles were correctly grouped together, irregardless of the k value. By analyzing the combination of slowdown measurements and performance metrics, we were able to distinguish *four* distinct clusters or application types. Lower k values resulted in large clusters with more diverse application profiles. Higher values created small sub-types with limited differences from the larger clusters. The resulting application types can be described as follows:

C1 : Applications limited by CPU usage which have very low amounts of L2 cache misses and very limited disk usage. Slowdown in this group is relatively low because of the limited degree of resource sharing.

Cluster	Members	Slowdown	CPU Usage	Cache Misses	%util (Disk)
C1	9	2.60%	98.80%	0.33%	0.44%
C2	8	8.14%	97.29%	2.40%	1.56%
C3	6	94.24%	91.56%	1.75%	10.83%
C4	7	900.55%	49.47%	2.64%	55.80%

Table 4.6: Averages values for the k-means clusters using the applications from T1. Slowdown is presented for a scaling level of 7.

C2 : Applications limited by CPU usage which exhibit significant amounts of cache misses and low disk usage. Average slowdown is higher than in C1 and most likely caused by increased cache sharing.

C3 : Applications with mixed limiting factors. Slowdown is caused by a combination of factors, not limited solely to disk or CPU. This cluster contains a diverse set of applications, both in resource usage and incurred slowdown.

C4 : Applications limited almost entirely by disk usage. The applications in this set will all have significant amounts of slowdown when sharing resources.

An overview of the cluster composition and average resource usage can be found in Table 4.6. We can clearly see that there are large differences in the average resource usage patterns. These differences should translate into preferences when co-scheduling applications.

4.7.2 Cluster Performance

With the introduction of these four application clusters, we can evaluate the performance interference when mixing different applications types. To quantify the contention effects of co-scheduled applications, we create *application pairs* between clusters and measure the slowdown incurred for all possible combinations. The test server contains two quad-core CPUs and a single hard drive. The effects of CPU and cache contention will only occur when there are multiple VMs per physical CPU socket, requiring the simultaneous execution of several identical application pairs. Unfortunately, accurate performance measurements for disk applications can only be recorded when a single application pair is executed. Adding identical pairs leads to resource contention between instances of the same application, obscuring the results. Two test configurations are therefore required to separate the effects caused by CPU, cache and disk sharing.

For clusters C1 and C2 we need to focus on the impact of CPU and cache sharing. On our eight core system, this was achieved by co-scheduling each benchmarking pair three times for a total of six concurrent VM workloads. Averages were taken from the three identical workloads to obtain a single slowdown score per application. To evaluate clusters C3 and C4 we require a detailed view on the impact of disk sharing. Concurrently benchmarking these disk heavy workloads multiple times can lead to results which are be difficult to interpret. Therefore, each pair was scheduled separately, resulting in a total of two concurrent workloads. Results are presented

in Table 4.7 and 4.8 respectively. The rows containing the results discussed in the following paragraphs are highlighted in bold. As expected, we find that there are significant differences in the average slowdown depending on the co-scheduled application type.

To evaluate the CPU- and cache-intensive applications in C1 and C2, we analyze the results presented in Table 4.7. Average slowdown for C1 is fairly low, regardless of the applications that are co-scheduled. The base measurement using only C1 applications has a slowdown of just 1.07%. Even co-scheduling with C4, the worst case scenario for C1, only increases this value to 1.89%. The C2 cluster has higher averages with a best case scenario of 2.56% slowdown when coupled with C1. The relative increase in slowdown when co-scheduling other application clusters remains similarly limited. The maximum average slowdown of 3.80% is again caused by co-scheduling with C4. As expected, applications which also have a small cache and disk component are more susceptible to contention effects than those which are purely CPU limited. In general, C1 is the preferred partner with performance getting progressively worse for C2, C3 and C4. The impact of selecting a suboptimal co-scheduling partner does remain relatively limited.

To evaluate the impact of disk sharing on a single application we look at C3 and C4 using the results in Table 4.8. Again, we find that co-scheduling C1 has a relatively low impact, being the preferred partner of both C3 and C4. It is interesting to note that C3 incurred a negligible impact of 0.62%, whereas C4 has a an average slowdown of 6.19%. For applications with intensive disk usage, the mere presence of an extra workload can be an important factor influencing performance. Like before, performance gets progressively worse when co-scheduling with C2, C3 and C4. Unlike the conclusions drawn from Table 4.7, the impact of selecting a bad co-scheduling partner for C3 and C4 can have large negative effects on performance. For C4 this results in an average increase from 6.19% to 104.23%, a difference of 98%.

Summarizing these findings, we can reach two general conclusions. First, the co-scheduled application type is relatively unimportant for C1 and C2. Even co-scheduling with the most disk intensive applications result in a relatively limited increase in the performance impact. Second, applications from C3 and C4 should be spread out over as many servers as possible. Combinations with applications from C1 and C2 are preferred, whereas co-scheduling applications from C4 should be avoided at all cost. These conclusions match what we would have intuitively expected. The most important conclusion from these experiments is that the automated approach succeeds in placing applications into clearly distinct groups, with measurably different behavior when co-scheduled. This will become an increasingly important asset in the next chapter, when new network related resource usage metrics are added.

4.8 Scheduling

Scheduling workloads in virtualized datacenters is often approached as a static packing problem. The physical hardware is divided into partitions without taking into account the dynamic interactions between workloads. Complex schedules can be difficult to implement when the available information is limited to the maximum resource allocation offered to each virtual machine e.g. 1 vCPU, 1 GB RAM, 10 GB disk. Using the techniques presented in Section 4.6 and 4.7, a new layer of

Monitored \ Co-scheduled	C1	C2	C3	C4
C1	1.07%	1.18%	1.43%	1.89%
C2	2.56%	3.11%	3.28%	3.80%
C3	9.17%	11.27%	21.79%	48.82%
C4	249.08%	276.45%	468.39%	724.19%

Table 4.7: Average slowdown when co-scheduling cluster applications in a pair-wise manner. Results are presented for a total of six VMs, ensuring CPU and cache contention.

Monitored \ Co-scheduled	C1	C2	C3	C4
C1	0.38%	0.39%	0.37%	0.72%
C2	0.65%	0.47%	0.71%	2.03%
C3	0.62%	1.04%	3.03%	11.05%
C4	6.19%	7.88%	25.66%	104.23%

Table 4.8: Average slowdown when co-scheduling cluster applications in a pair-wise manner. Results are presented for a single pair of VMs, focusing on disk contention.

information can be provided during the scheduling process. Additional knowledge concerning the application type and consolidation sensitivity opens up a new range of scheduling possibilities. To demonstrate the potential impact this new information can have, we introduce and evaluate several scheduling strategies. Each technique is used to schedule a static set of applications on a fixed number of servers. For the purpose of our demonstration we focus on lowering the average slowdown experienced by co- scheduled applications. However, there are many other metrics for which a scheduling technique can be optimized using this data. An alternative could be the isolation of high-impact applications, providing negligible interference and predictable performance for a large majority of applications. In addition to performance interference, many diverse factors can also influence the optimal placement of VMs e.g. data locality or power consumption. The exploration of these additional optimizations goes beyond the scope of this chapter.

4.8.1 Algorithms

To evaluate the scheduling potential of the slowdown predictions and application clusters, we designed three different scheduling techniques. The first technique was named the *Cluster Only* scheduler and uses only the clustering information as input. The *Prediction Only* and *Bucket* schedulers are based on the slowdown predictions obtained using the best-performing *Class SVM* technique. All three techniques share the same goal: reducing average slowdown by spreading the most sensitive workloads and co-scheduling with appropriate application types. A *Random* scheduler was added for comparison.

We also developed a hybrid scheduler using both slowdown predictions and application clusters as input. With the currently presented combination of metrics and applications we were not able to consistently improve the scheduling performance. Research presented in the following chapter demonstrates the benefits of using both types of information simultaneously.

Cluster Only : Applications are divided in groups according to the clusters described in Section 4.7. Members of these groups are then sorted in descending order using the average recorded slowdown found in Table 4.6. An ordered list is created by iteratively taking a random application from the first non-empty group. The ordered applications are scheduled using the placement strategy presented in Algorithm 3. A schematic representation of this approach can be found in Figure 4.8.

Prediction Only : Applications are ordered according to their predicted slowdown, with the most sensitive applications at the front. For the subset of applications with no predicted slowdown, the order is random. The placement strategy is identical to the *Cluster Only* scheduler, the only difference being the information used to order the applications.

Bucket : Each server candidate is given a weight that corresponds to the slowdown predictions of the applications already assigned to the server. Additionally, a maximum number of workloads per server is enforced to avoid overbooking. Applications are sorted according to their predicted slowdown in descending order. Each new application is scheduled on the server with the lowest weight, provided that the server can still accept an additional workload. The server's weight is increased with the predicted slowdown, e.g 1.2 for a predicted slowdown of 20%. High level pseudo code of this placement strategy can be found in Algorithm 4.

Random : Applications are randomly selected and spread evenly across the servers.

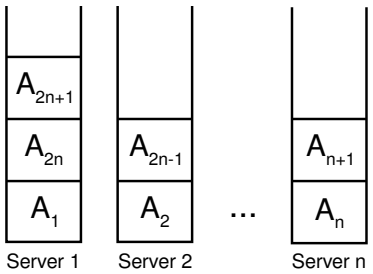

Figure 4.8: Schematic representation of the placement strategy presented as Algorithm 3.

4.8.2 Setup

All scheduling techniques are evaluated on the hardware platform described in section 4.5. The experimental setup consists of two application sets scheduled on

```
Input: Servers, Applications, Profiling Data
SortedApplications ← Sort(ProfilingData, Applications);
Direction ← 1;
CurrentServer ← 1;
foreach Application ∈ SortedApplications do
    Schedule(Application, CurrentServer);
    CurrentServer ← (CurrentServer + Direction);
    if CurrentServer = #Servers + 1 then
        Direction ← −1;
        CurrentServer ← #Servers;
    else if CurrentServer = 0 then
        Direction ← 1;
        CurrentServer ← 1;
    end
end
```

Algorithm 3: Placement strategy used by the *Cluster* and *Prediction Only* schedulers. Sort() orders applications from high to low slowdown using clusters or slowdown predictions respectively.

```
Input: Servers, Applications, Profiling Data
SortedApplications ← Sort(ProfilingData, Applications);
CurrentServer ← 1;
foreach Application ∈ SortedApplications do
    foreach Candidate ∈ Servers do
        if Weight(CurrentServer) > Weight(Candidate) then
            CurrentServer ← Candidate;
        end
    end
    Slowdown ← PredictedSlowdown(Application);
    AddWeight(CurrentServer, Slowdown);
    Schedule(CurrentServer, Application);
end
```

Algorithm 4: Placement strategy used by the *Bucket* scheduler. Sort() orders applications from high to low predicted slowdown.

four different hardware configurations ranging from 8 to 14 servers. Each server can accommodate a maximum of seven workloads without overbooking, leaving a single CPU core for the hypervisor.

For each experiment, we conducted 10 iterations. This is sufficient to obtain statistically significant results. The following application sets are used to evaluate each scheduling technique:

S1 : 53 applications[1] consisting of an even mix from all application types.

S2 : 48 applications[2] with an emphasis on applications with significant slowdown.

[1] $S1 = 2 * (C1 + C2 + C3) + C4$

[2] $S2 = C1 + C2 + 3 * C3 + C4$

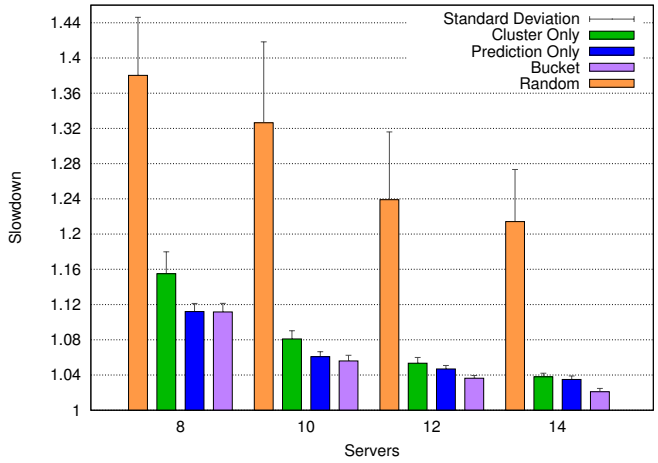

Figure 4.9: Average slowdown per scheduler using S1

Servers	8	10	12	14
Random	38.0% $\sigma = 6.6^{-2}$	32.6% $\sigma = 9.2^{-2}$	23.9% $\sigma = 7.7^{-2}$	21.4% $\sigma = 5.9^{-2}$
Cluster Only	15.5% $\sigma = 2.5^{-2}$	8.1% $\sigma = 0.9^{-2}$	5.3% $\sigma = 0.7^{-2}$	3.8% $\sigma = 0.4^{-2}$
Prediction Only	11.2% $\sigma = 0.9^{-2}$	6.1% $\sigma = 0.6^{-2}$	4.7% $\sigma = 0.4^{-2}$	3.6% $\sigma = 0.4^{-2}$
Bucket	11.2% $\sigma = 1.0^{-2}$	5.6% $\sigma = 0.6^{-2}$	3.6% $\sigma = 0.3^{-2}$	2.1% $\sigma = 0.4^{-2}$

Table 4.9: Average slowdown per scheduler using S1.

4.8.3 Results

The purpose of these experiments is to demonstrate the scheduling improvements that can be made by moving from a coarse- to a fine-grained classification of applications. Adding information about application types already adds valuable knowledge to the scheduling process. With these results we show that further improvements can be made by using accurate slowdown predictions. Average slowdown per application is presented for S1 and S2 in Figures 4.9 and 4.10, details can be found in Tables 4.9 and 4.10. To put the performance of the *Cluster Only* scheduler in the proper perspective, results from a *Random* scheduling policy were also included. We discuss our results in two parts. In the first, we look at the limitations of scheduling with application clusters only. In the second part, we discuss the differences between the placement strategies.

All non-random scheduling algorithms use a sorted list as the input for their placement strategy. When we use applications types to create this list, the order is

determined by the average slowdown per cluster. Differences within clusters cannot be taken into account. To determine the impact of this limitation we compare the results from the *Cluster Only* and *Prediction Only* schedulers. In Figure 4.9 we can see that the *Cluster Only* scheduler consistently performs worse with a maximum difference of over 4% when using 8 servers. When more servers are added, members from the same cluster are less likely to be co-scheduled, reducing the chance for suboptimal placement decisions. This effect can be attributed to the diverse application set used in S1, resulting in small clusters per application type. Scheduling a less diverse application mix reduces this advantage. In Figure 4.10 we created a scenario where 50% of the applications belong to the same application type. Scheduling with S2 on 14 servers, the *Prediction Only* scheduler manages to reduce average slowdown to 3.3%, whereas the *Cluster Only* scheduler still averages over 10%.

Slowdown predictions are a powerful tool that can be used for more than simple application ordering. The *Bucket* scheduler uses application slowdown predictions to not only influence the placement order, but also the preferred server. Applications with large predicted slowdowns receive fewer and relatively lighter co-scheduled workloads. In Figures 4.9 and 4.10 we find that this approach can further reduce the average slowdown when sufficient servers are available. Scheduling S1 on 8 servers is the only case where we do not observe any improvement when compared to the *Prediction Only* scheduler. In all other cases, we managed to improve the average slowdown by using a more intelligent placement strategy. Scheduling with S1 on 14 servers even manages to reduce average slowdown from an already low 3.6% to just 2.1%.

4.9 Conclusion

In this chapter, we addressed the issue of consolidating resource intensive VMs and the resulting variances in workload performance. We provided new and valuable information to datacenter schedulers by introducing accurate performance models and correct classification of applications. We developed a number of innovative scheduling techniques that build upon this information and demonstrated the resulting benefits.

To gather the required training data we proposed an extensive benchmark suite and an accompanying profiling framework. The resulting performance scores and metrics provided the input for both application modeling and clustering. Several modeling techniques were evaluated on different training sets. We found that SVM performed best when the training data contained only resource intensive applications. However, when adding applications with low resource requirements none of the modeling techniques provided sufficiently accurate predictions. To address this issue, we extended the training set with a range of data points representing applications with low resource requirements. Results were further improved by using the classification abilities of SVM to create the *Class SVM* technique. The combination of *Class SVM* and the extended data set significantly outperformed all other techniques. Average relative prediction error remained below 8% for all scaling factors, a reduction of up to 5% when compared to Weighted Means.

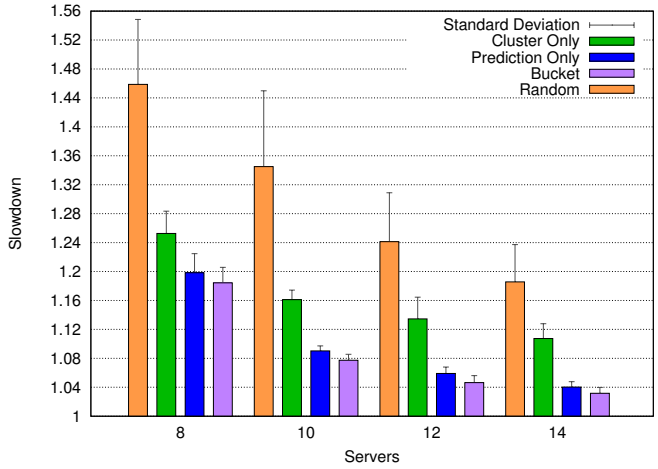

Figure 4.10: Average slowdown per scheduler using S2

Servers	8	10	12	14
Random	45.9% $\sigma = 9.0^{-2}$	34.5% $\sigma = 10.5^{-2}$	24.1% $\sigma = 6.8^{-2}$	18.6% $\sigma = 5.2^{-2}$
Cluster Only	25.3% $\sigma = 3.1^{-2}$	16.1% $\sigma = 1.3^{-2}$	13.5% $\sigma = 3.0^{-2}$	10.7% $\sigma = 2.0^{-2}$
Prediction Only	19.8% $\sigma = 2.6^{-2}$	9.0% $\sigma = 0.7^{-2}$	5.9% $\sigma = 0.9^{-2}$	4.0% $\sigma = 0.7^{-2}$
Bucket	18.4% $\sigma = 2.1^{-2}$	7.7% $\sigma = 0.8^{-2}$	4.7% $\sigma = 1.0^{-2}$	3.2% $\sigma = 0.8^{-2}$

Table 4.10: Average slowdown per scheduler using S2.

Using application clusters we were able to automatically identify several distinct application types. With this new classification we were able to quantify the slowdown when co-scheduling different application types. Although this information provides valuable insight, we suspect more significant results can be achieved by adding new performance metrics e.g. network resources.

To determine the potential of this new information, we developed and evaluated a set of three scheduling techniques. Each technique was used to distribute two types of workload across four server configurations. We demonstrated the scheduling improvements that can be made by moving from a coarse- to a fine-grained classification of applications. Prediction based scheduling reduced the average slowdown per application consistently when compared to the cluster based approach. Results were further improved by implementing a more intelligent bucket-based placement algorithm, reducing average slowdown by up to 10%.

CHAPTER 5

Network Aware Scheduling for VM Workloads

This chapter is based on "Network Aware Scheduling for Virtual Machine Workloads with Interference Models"

Abstract

In this chapter, we add network awareness to our existing performance models based on the runtime characteristics of virtualized workloads. We increase the applicability of the available training data by adding a network component to the performance metrics and benchmarks. Using the extended set of performance profiles, we predict performance degradation with existing modeling techniques as well as combinations thereof. Application clustering is used to identify several new network-related application types with clearly defined performance profiles. Finally, we validate the added value of the improved models by introducing new scheduling techniques and comparing them to previous efforts. We demonstrate how the inclusion of network-related parameters in performance models can be used to significantly increase the performance of consolidated workloads.

5.1 Introduction

In the previous chapter, we introduced performance aware scheduling techniques based on slowdown prediction models and application classification. We demonstrated that performance prediction models can be used to greatly reduce interference effects and improve overall performance. In this chapter, we extend the proposed approach to include the performance impact incurred and caused by network-based application types.

To validate the general applicability of the prediction and classification approach, we analyze the performance interference effects of network-based resource consumption. We significantly increase the set of benchmarks by adding multiple benchmark configurations with variable network resource usage. Adding these benchmarks allows us to create an all encompassing training set which can be used to validate the modeling and classification approach. Adding a new potential source of performance interference also requires the monitoring of new performance metrics. These additional metrics create new dimensions in the performance profiles used for both modeling and classification. The main research question in this chapter can be split into two parts. First, does the inclusion of additional dimensions significantly affect the performance model accuracy? Second, are these new metrics sufficient to automatically classify useful network-related application types? To answer these questions, we use the extended application profiles to evaluate both modeling and classification accuracy. Performance predictions are evaluated using existing modeling techniques and a new hybrid approach. K-means clustering is used to automatically define new network-based application types which have distinct behavior under contention. The results from the performance analysis are validated with scheduling experiments. Two novel scheduling approaches are introduced to determine the potential benefits of combining accurate application classification with prediction models.

The remainder of this chapter is organized as follows: Section 5.2 discusses existing work and recent developments with a focus on virtualized network I/O. Section 5.3 details the extended set of metrics gathered when profiling benchmarks. In Section 5.4 we describe the new benchmark configurations, the methods used to create performance profiles and our experimental setup. Existing and new modeling techniques are evaluated in Section 5.5, followed by the identification of new application types in Section 5.6. An evaluation of the performance models using novel scheduling algorithms is discussed in Section 5.7. Our final conclusions can be found in Section 5.8.

5.2 Related Work

In this section, we provide an overview of the research concerning virtualized network I/O. Work related to the general approach can be found in Section 4.2.

In the past, hypervisors have had a strong focus on fair scheduling with regards to computationally intensive workloads. I/O scheduling was often seen as a secondary concern, resulting in sub optimal performance for I/O-intensive workloads. Mixing compute-intensive workloads with bandwidth-intensive and latency-sensitive workloads can result in significant performance degradation. Ongaro et al. [**?**] studied the impact of several hypervisor scheduler configurations on performance interference.

Their Xen based study shows that the current VMM schedulers do not achieve the same level of fairness for I/O-intensive workloads as they do for compute-intensive workloads. Aragiorgis et al. [**?**] argue that concurrent VM hardware accesses can be optimized with new scheduling concepts. Taking into account the complexity of designing and implementing a universal scheduler, they focus on a system with multiple scheduling policies that coexist and service VMs according to their workload characteristics. However, their validation is limited to multiplexing network I/O and CPU-intensive workloads. Shadi et al. [**?**] demonstrated that different combinations of disk schedulers in both the hypervisor and the VMs can also greatly influence the performance of I/O-intensive workloads. They propose a novel approach for adaptive tuning of the disk schedulers at both scheduling levels. Our approach takes a higher level view and tries to reduce the opportunities for interference. Although many opportunities exist to improve scheduling fairness, aggregated performance can be further increased by optimally distribute workloads across physical machines.

Lloyd et al. [**?**] used physical and virtual machine resource utilization statistics to build performance models. They evaluated eighteen individual resource utilization statistics by predicting service execution time using several modeling approaches. They concluded that the strength of individual predictors depends greatly on the evaluated application profile. Most notably, they found that CPU idle time and number of context switches were good predictors for I/O bound applications. Their objective was to predict which application components could be combined in a single VM to provide optimal performance while requiring the fewest number of VMs. In contrast, our approach focuses on providing detailed VM-level performance profiles to optimize datacenter scheduling.

Considerable work is also being performed on in-depth analysis of the causes and effects of performance interference in various consolidation scenarios [**?**, **?**, **?**]. Pu et al. [**?**] performed extensive experiments to measure performance interference on CPU and network I/O bound VM workloads. They found that network-intensive workloads can incur high overheads due to extensive context switches. They also concluded that multiplexing CPU and network-intensive workloads results in relatively low amounts of performance degradation. Wang et al. [**?**] measured processor sharing, packet delay, TCP/UDP throughput and packet loss among Amazon EC2 VMs. Their results demonstrated that despite the light network utilization, virtualization still causes significant throughput instability and abnormal delay variations. Mei et al. [**?**] present a detailed analysis of the different elements responsible for performance interference. They also investigated the impact of different CPU resource scheduling strategies and varying workload rates on the performance of consolidated VM workloads. Their results suggest that significant performance gains can be made by strategically multiplexing correct application types.

Performance isolation can also be influenced by hypervisor implementation differences. Hwang et al. [**?**] performed an extensive performance analysis, comparing four popular virtualization platforms; Hyper-V, KVM, vSphere and Xen. They found that the overheads incurred by each hypervisor can vary significantly depending on the application type and the resources assigned to it. Network interference was measured using the HTTP response time while simultaneously stressing CPU, memory, disk, and network resources with different benchmarks. Xen was observed to be particularly sensitive to memory and network interference. Dramatic differences were also found in the performance isolation provided by different hypervisors. At-

tempts to reduce network I/O interference have also resulted in hardware solutions [**?**]. Dong et al. explain the architectural, design and implementation considerations for Single Root I/O Virtualization in Xen. A specification that allows a PCIe device to present itself as a separate physical PCIe device for each VM.

In summary, related work has identified several factors causing performance interference and provided solutions at the server level. Our approach provides accurate performance prediction and classification to mitigate interference at the datacenter level. By adding detailed application profiles to the scheduling process, we reduce resource contention and interference without hypervisor modifications.

5.3 Profiling

In order to improve modeling accuracy, we gather an extended set of system level characteristics using the framework described in Section 4.4. Network related metrics are obtained using the *vnStat* package. *XenBaked* is used to record additional VM scheduling details. *Xenoprof* and *Iostat* provide the cache and disk metrics.

A modified version of XenBaked is used to keep records for each VM's virtual CPU e.g. cycles *gotten* and *blocked*. To improve the application profiles, *VM switches* per second are also monitored using XenBaked. The decision to include this parameters is based on the conclusions reached by Lloyd et al. [**?**]. A VM switch occurs whenever a virtual CPU is allocated a physical CPU core. Large amounts of switches can be used to determine the number of times a VM needs to relinquish control to the hypervisor. When modeling VM performance for network applications, this can be an important parameter during classification [**?**]. VnStat is used to retrieve network I/O statistics from the paravirtualized guest kernels. For our experiments, we monitor averages for the following network metrics: received and transmitted *throughput* (*rMB/s* and *tMB/s*) and *packages* per second (*rPackets/s* and *tPacket/s*). An overview of the profiled parameters can be found in Table 5.1.

5.4 Benchmarking

We extend the application set from Section 4.5 by adding five applications with a significant network resource requirement. We were able to profile a broad range of resource usage patterns using multiple configurations of the following benchmarks. Two web server benchmarks (*ApacheBench* and *Httperf*) and two synthetic bandwidth benchmarks (*Nuttcp* and *Netperf*). To compensate for the limited number of applications we took advantage of the configurability of each benchmark to create multiple data points. To add an application using a mix of network and other recourses, we use a rate limited version of Sysbench to benchmark a VM hosting an SQL database. An overview of the selected applications can be found in Table 5.2.

The web server benchmarks are tested by issuing requests with file sizes of 1kb, 10kb, 100kb and 1Mb. Additionally, a PHP page executing *phpinfo()*, a relatively CPU intensive request, is added to generate a workload with more CPU usage. The synthetic benchmarks are tested with the server component hosted on either the VM or the corresponding test client. This was done to gather profiles for both sending and receiving network data. Nuttcp is only used as a TCP streaming benchmark. Netperf profiles are made for TCP streaming and both TCP and UDP Response/Request (RR)

Metric	Measurement
gotten	Percentage of CPU cycles used for processing
blocked	Percentage of CPU cycles spent idle or waiting on an event
hit/miss	Percentage of L2 cache requests that missed
dr/s	Average disk reads per second
dw/s	Average disk writes per second
drMB/s	Average disk read throughput in MB per second
dwMB/s	Average disk write throughput in MB per second
await	Average time needed for disk I/O requests to be served
%util	Percentage of time the disk spent servicing I/O requests
switches	Average switches to a new physical CPU per second
rMB/s	Average network receive throughput in MB per second
tMB/s	Average network transmit throughput in MB per second
rPackets/s	Average packets received per second
tPackets/s	Average packets transmitted per second

Table 5.1: Extended overview of the metrics obtained from profiling.

benchmarks. Sysbench is profiled on a read only and a read/write database. As these benchmarks only provide us with data for resource limited applications, we add a variety of rate limited benchmarks. An overview of the rate limited configurations used for the network benchmarks can be found in Table 5.3

Together with the already available benchmark set from Section 4.5, this creates a large and diverse range of applications taxing all important resources. To our knowledge, the presented training set significantly surpasses any previous efforts to create a virtualized performance model. All previous benchmarks results are discarded as new metrics needed to be added to each performance profile. The application profiles in this chapter are measured using an updated benchmark platform and new OS configurations. As such, small variations in the results can be observed when compared to the previous chapter. Results are obtained using Xen 4.0 with a 2.6.32.24 Linux kernel. All VMs are configured with 1GB RAM using disk images mounted via blktap2. To facilitate the interpretation of the results all VMs are issued a single vCPU. The benchmarking platform consisted of dual Intel Xeon L5335 quad-core CPUs, 16GB RAM and a WDC WD2500YS hard drive (7200rpm, 16MB cache). All servers use a dedicated gigabit Ethernet connection.

5.5 Modeling

In this section we evaluate the modeling techniques described in Section 4.6 using two new sets of training data. We investigate the impact of adding network-related metrics and application profiles on the model accuracy. Based on initial findings, we present a new prediction approach using a combination of existing models. Training data is gathered with the benchmark platform introduced in Section 5.4. Each

Name	Measurement
ApacheBench	HTTP web server performance
Httperf	HTTP web server performance
Nuttcp	TCP bandwidth performance
Netperf	TCP and UDP bandwidth performance
Sysbench	Remote database server performance

Table 5.2: Overview of the network applications used for benchmarking.

Benchmark	Rates
Httperf	Unlimited and 1, 5, 50, 150, 300 requests/s
ApacheBench	Unlimited only
Netperf RR	Unlimited and 100 ms bursts with 50, 100, 200, 500, 1000 and 2000 ms waits
Netperf Stream	Unlimited and 100 ms bursts with 500, 1000 and 2000 ms waits
Nuttcp Stream	Unlimited and 50, 100, 200, 500 Mbps limit
Sysbench	Unlimited and 10, 30, 50, 70, 100 requests/s

Table 5.3: Overview of the different configurations used for the network benchmarks.

modeling approach is used to predict an application's sensitivity to performance interference. Scores are normalized to one, the result from a standalone run on the test platform. If a training set contains multiple profiles for a rate limited application, these configurations are omitted when training a model to predict the performance of this application. We compare three types of prediction techniques: Weighted Means (WM), Linear Regression and Support Vector Machines (SVM). Details about these techniques can be found in Section 4.6. The Class SVM modeling approach is modified to make better use of the extended training data. Additionally, we also present the results from a new hybrid approach that combines Weighted Means and the Class SVM model.

5.5.1 Class SVM

In Section 4.6.3, we introduced the Class SVM modeling approach. A model that uses the classification capabilities of SVM to accurately identify applications which are insensitive to performance interference. Predictions are therefore only required for applications that are likely to exhibit slowdown. Using the available training data, predictions could be improved by limiting the training data for the prediction model to applications profiles with measurable slowdown. The complete set of training data was only required to build the classification model. In this chapter, we extended the training data by significantly increasing both the number of applications and profiled metrics per application. Using this extended training data, we find that

prediction accuracy is no longer increased by limiting the training data. As such, we now require the entire set of training data for the modeling step.

5.5.2 Hybrid Model

Detailed analysis of the Weighted Means and Class SVM models indicates that a complementary approach can lead to improved results. To test this hypothesis we create a hybrid scheduler, utilizing the strengths of both models. The first step uses an SVM based binary classification model to identify all applications unlikely to suffer slowdown. The second step generates both a WM and SVM based slowdown prediction for the remaining applications. An ensemble model is created by using the average of both values. The complementary results reduce the prediction errors for applications without slowdown as well as improve results for applications with significant slowdown.

5.5.3 Application Sets

We evaluated all techniques using the following sets of training data.

T1 : High Resource Requirement (HRR) Only - this set contains all the applications listed in Table 4.2 and Table 5.2. All applications in Table 5.2 are benchmarked using the unlimited rates for each configuration found in Table 5.3. Combining all benchmarks and configuration options results in a total of 50 applications. We use this set for the initial evaluation because its members tax all recourses which can limit an application's performance. All applications have high resource requirements and tax a single resource or combination thereof to obtain a performance score.

T2 : T1 + Rate Limited - new "applications" are added using rate limited versions of the Sysbench disk and OLTP benchmarks and the rate limited configurations found in Table 5.3. Test configurations are selected to ensure an even distribution across the entire resource usage spectrum. The additional training data allows the modeling techniques to determine the tipping point where an application starts exhibiting slowdown. Adding these artificial applications brings the total number of applications in the set to 142.

Details about the configuration of the Sysbench rate limited benchmarks can be found in Section 4.6.4. Configuration details about the network-based benchmarks can be found in Section 5.4.

5.5.4 Results

We present our results in two parts, according to the training set used to evaluate the predictive capacity of each technique. Previous results presented in Section 4.6.5 revealed that there are no significant differences between the relevant scaling factors. Therefore, predictions are only presented for a scaling factor of seven concurrent workloads. The models are evaluated using both the leave-one-out cross-validation (LOOCV) method and K-fold cross-validation (CV). LOOCV is chosen as the main evaluation method due to the limited size and diversity of the training data. We

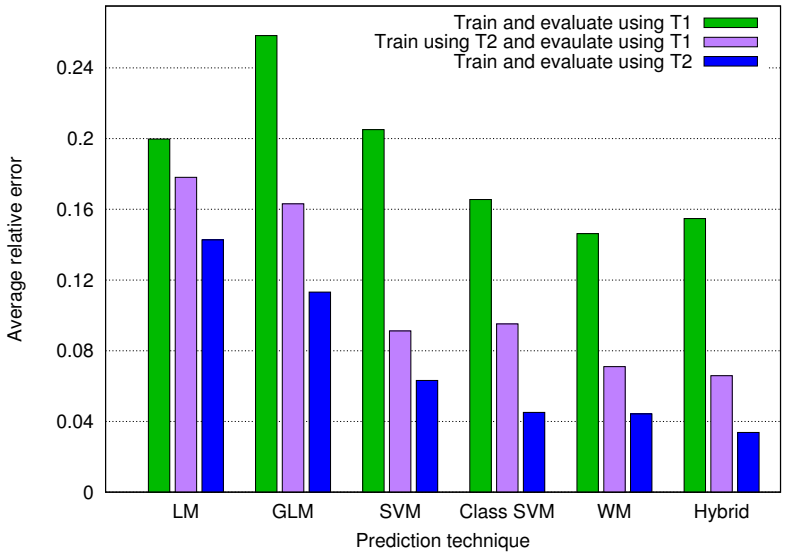

Figure 5.1: Relative prediction errors for each modeling approach using LOOCV.

analyze model accuracy by calculating the average error produced when predicting application performance using the remaining profiles as training data. LOOCV allows us to make efficient use of the available data. To confirm our conclusions, we perform K-fold cross-validation by creating 10 folds in 10 separate runs.

The first evaluation uses application profiles from set T1 as the training and evaluation data. T1 contains only applications with high resource requirements. The average prediction error produced by each technique can be found in Figure 5.1. We find that across the board, all techniques perform poorly. More training data is need to reduce the average prediction errors. However, looking at the error distribution graphs in Figure 5.2 we find that WM performs relatively well. The high average error can be attributed almost entirely to a small number of large prediction errors, overestimating the slowdown for network-based applications. Conversely, the errors produced by the SVM based techniques are predominantly caused by underestimating the interference sensitivity of network-based applications. The linear models result in both over- and underestimates across all application types. Although WM is clearly the superior option in this scenario, improved training data is required for a robust model.

An extended set of training data is needed, adding a range of applications with low resource requirements and no slowdown. We obtain additional data points using rate limited benchmarks as described in set T2. In Figure 5.1, we find the results when using T2 to train the models. Results are presented using either the applications in T1 or T2 as the evaluation set. Using the smaller T1 set for evaluation, we can separate the predictions errors for resource intensive applications with significant slowdown. Both linear approaches still produce large prediction errors and perform much worse than the WM- and SVM-based options. As there are relatively few applications without slowdown, we see little difference between regular SVM and the Class SVM approach. Both produce an average error slightly above 9%.

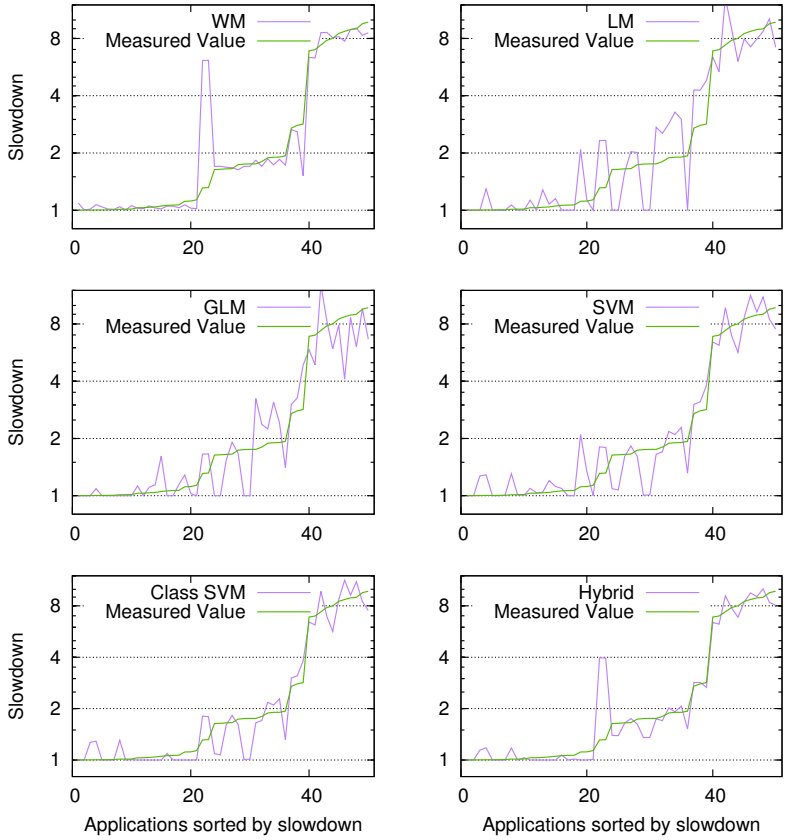

Figure 5.2: Error distribution using applications from T1 for both training and evaluation using LOOCV.

WM performs substantially better with an average error of only 7.1%. However, the Hybrid approach proves to be most accurate with an average error of only 6.5%.

Using T2 as both the training and evaluation set provides a more realistic use case where most applications do not exhibit significant slowdown. Average errors are reduced for all techniques. This can be attributed to the large number of applications without slowdown which are relatively easy to predict. Looking at the error distribution in Figure 5.3, we find that only the linear techniques have significant problems with identifying these applications. Looking at the average slowdowns in Figure 5.1, we find that the linear techniques are still inferior to all other options. Regular SVM performance is now significantly lower than Class SVM with an average error of 6.3% compared to 4.5%. Class SVM and WM provide comparable performance with both models producing an average error of 4.4%. However, comparing the Class SVM and WM graphs in Figure 5.3 we can identify different underlying causes for each number. Where WM has larger errors for applications without slowdown, Class SVM has larger errors on high slowdown applications. These distinct differences are directly responsible for the low average error produced by the Hybrid model. With an average error of just 3.3% it is clearly the superior approach.

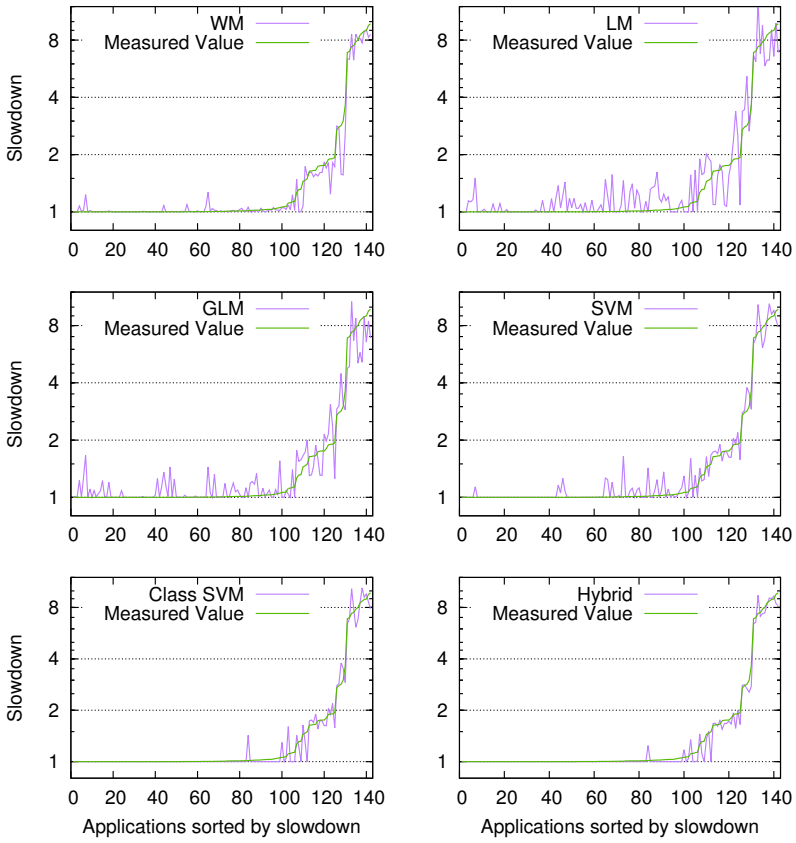

Figure 5.3: Error distribution using applications from T2 for both training and evaluation using LOOCV.

As in Section 4.6.5, we further validate these conclusions using 10-fold cross-validation with T2 for both training and evaluation. The results in Figure 5.4 confirm our conclusions. As expected, all techniques produce moderately higher average prediction errors. Class SVM and WM produce similar error averages of 5.1% and 5.2% respectively. The Hybrid approach performs best with an average prediction error of 3.9%.

5.6 Clustering

In this section, we evaluate clustering as a method to automatically distinguish between well-defined application types. Specifically, we focus on the correct identification of applications with a significant network component. In Section 4.7, we identified four clusters of applications with distinct performance profiles. Subsequently, we used these application clusters to quantify the average slowdown when co-scheduling different application types. Adding a network component allow us to detect and evaluate additional application clusters. These application types provide opportunities to identify new combinations that are ideally suited for consolidation.

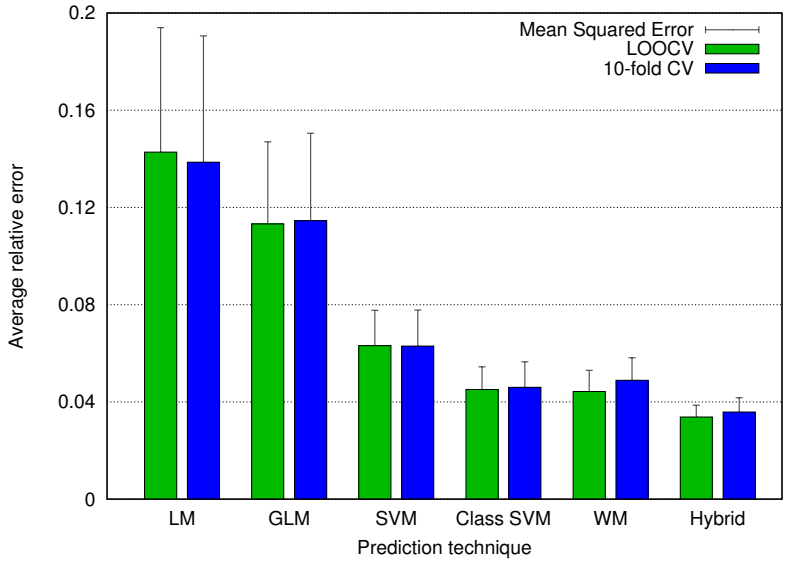

Figure 5.4: Relative prediction errors for both LOOCV and K-fold using T2 for both training and evaluation.

5.6.1 Clusters

K-means clustering was evaluated for k values from two until seven. Clustering is based on the performance metrics gathered from profiling the resource limited applications in set T1. Applications with similar slowdown profiles were correctly grouped together, irregardless of the k value. By analyzing the combination of slowdown measurements and performance metrics, we were able to distinguish either *four* large or *six* detailed clusters as the optimal choices. Lower k values resulted in clusters with relatively large differences in the included application profiles. Increasing the number of clusters beyond six split already accurately defined application types. Clustering with five groups created a single new group which should logically be split into two. Therefore, we chose to cluster into six groups to add further detail to our analysis. The resulting application types for four clusters can be described as follows:

C1 : Applications limited mainly by CPU usage. Slowdown in this group is relatively low because of the low resource sharing. These applications correspond with the C1 and C2 clusters described in Section 4.7.

C2 : Applications with mixed limiting factors. Slowdown is caused by a combination of factors, not limited solely by CPU, disk or network I/O. This cluster contains a diverse set of applications, both in resource usage and incurred slowdown. These applications correspond with the C3 clusters described in Section 4.7.

C3 : Applications characterized by their network usage. The applications in this set are limited by network-related resources in various degrees. All new network resource based applications can be found in this cluster.

C4 : Applications limited almost entirely by disk usage. The applications in this set will all have significant amounts of slowdown when sharing disk resources. These applications correspond with the C4 clusters described in Section 4.7.

An overview of the cluster composition and average resource usage can be found in Table 5.4. We can clearly distinguish large differences in the average application profiles. These differences should translate into preferences when co-scheduling applications. However, there is a large single group of applications (C3) that we can further divide in a meaningful manner. By increasing the k value from four to six, we can create two new groups containing the TCP streaming applications. These applications generate significantly higher bandwidth usages than the remaining network applications and are likely to have a distinct impact on other application types. The new application types for six clusters can be described as follows:

C3 : Applications with a significant network component but not limited by the maximum network bandwidth.

C5 : Applications limited by a combination of the available receiving bandwidth and CPU usage. Performance will most likely be limited by the amount of bandwidth required by applications sharing the network connection. However, the relatively high CPU requirements can be a contributing factor.

C6 : Applications limited almost entirely by the maximum transmitting bandwidth of the network. Performance is reduced by the amount of outgoing bandwidth required by applications sharing the network connection.

An overview of the cluster composition and average resource usage can be found in Table 5.5.

5.6.2 Cluster Performance

With the introduction of these application clusters, we can evaluate the performance variance when mixing different application types. To quantify the contention effects of co-scheduled applications, we create application pairs between clusters and measure the slowdown incurred for all possible combinations. Benchmarks are first presented for a single application pair, resulting in two concurrent VMs. To demonstrate that the conclusions are still valid under more strenuous resource contention, benchmarks were also performed using three pairs for a total of six concurrent VMs. We analyze the results using both four and six application clusters. As the conclusions from Section 4.7 about the interactions between C1, C2 and C4 are still valid, we focus on the interactions with the new network-related application types in C3, C5 and C6.

The results using four clusters can be found in Tables 5.6 and 5.7. As expected, C1, C2 and C4 behave similar to the performance figures described in Section 4.7.2. The new C3 group has a limited impact on all other application types, making it an ideal co-scheduling partner. Conversely, applications from C3 suffer limited effects from resource sharing with other applications types. Similar to the disk heavy applications, the network based applications from C3 incur large average slowdown when co-scheduled with members from the same cluster. These results are very promising

Cluster	#	Slow-down	CPU gotten	L2 hit/miss	%util (Disk)	rMB/s (Net)	tMB/s (Net)
C1	16	2.9%	96.6%	1.1%	0.9%	0	0
C2	7	44.0%	94.0%	2.4%	7.9%	0	0
C3	20	207.7%	70.4%	1.2%	0.5%	7.7	21.5
C4	7	745.2%	43.7%	2.8%	55.0%	0	0

Table 5.4: Average profiled metrics for K-means clusters with $k = 4$.

Cluster	#	Slow-down	CPU gotten	L2 hit/miss	%util (Disk)	rMB/s (Net)	tMB/s (Net)
C1	16	2.9%	96.6%	1.1%	0.9%	0	0
C2	7	44.0%	94.0%	2.4%	7.9%	0	0
C3	16	82.6%	71.0%	0.9%	0.6%	0.3	12.7
C4	7	745.2%	43.7%	2.8%	55.0%	0	0
C5	2	823.3%	91.8%	2.4%	0.2%	74.5	0.4
C6	2	593.0%	44.5%	2.2%	0.4%	0.4	112.9

Table 5.5: Averages profiled metrics for K-means clusters with $k = 6$.

1st \ 2nd	C1	C2	C3	C4
C1	0.84%	0.95%	0.78%	1.66%
C2	1.25%	2.09%	1.77%	3.80%
C3	0.71%	1.48%	**152.91%**	7.78%
C4	2.52%	**17.78%**	3.46%	**110.23%**

Table 5.6: Average slowdown when co-scheduling cluster applications in a pair-wise manner. Results are presented for a total of two VMs and four groups

1st \ 2nd	C1	C2	C3	C4
C1	1.68%	3.04%	2.23%	2.45%
C2	10.84%	24.42%	9.75%	33.59%
C3	51.87%	63.51%	**641.86%**	75.96%
C4	250.13%	**432.33%**	229.68%	**703.31%**

Table 5.7: Average slowdown when co-scheduling cluster applications in a pair-wise manner. Results are presented for a total of six VMs and four groups

1st \ 2nd	C1	C2	C3	C4	C5	C6
C1	0.84%	0.95%	0.76%	1.66%	0.85%	0.99%
C2	1.25%	2.09%	1.70%	3.80%	2.35%	2.00%
C3	0.81%	1.49%	11.70%	7.45%	**117.00%**	**1999.11%**
C4	2.52%	**17.78%**	2.51%	**110.23%**	13.56%	3.82%
C5	0.33%	2.77%	1.51%	**19.23**%	**64.46%**	**117.97%**
C6	0.00%	0.00%	0.56%	0.00%	0.89%	**96.19%**

Table 5.8: Average slowdown when co-scheduling cluster applications in a pair-wise manner. Results are presented for a total of two VMs and six groups

1st \ 2nd	C1	C2	C3	C4	C5	C6
C1	1.7%	3.0%	2.2%	2.4%	2.2%	2.5%
C2	10.8%	24.4%	9.4%	33.6%	10.4%	12.6%
C3	16.8%	25.8%	57.8%	37.0%	**1937.1%**	**5523.6%**
C4	250.1%	**432.3%**	228.7%	**703.3%**	229.2%	240.2%
C5	254.3%	304.8%	306.7%	**343.1**%	**680.0%**	**1882.6%**
C6	200.4%	199.2%	201.3%	198.7%	200.5%	**480.9%**

Table 5.9: Average slowdown when co-scheduling cluster applications in a pair-wise manner. Results are presented for a total of six VMs and six groups

and indicate that improved scheduling policies should be possible. However, using four clusters we have created a large single group concentrating all applications with a network component. To investigate the performance differences within this group we need to separate additional clusters.

The results using six clusters can be found in Tables 5.8 and 5.9. Splitting the network applications into three different groups leads to some interesting results. The group of moderate network applications (C3) suffers disproportionate slowdowns when paired with C6. VMs sending a TCP stream, consuming all available bandwidth, can severely starve response/request based network applications. C5 also has a major impact on C3 but the effect is far less pronounced. Applications from C3 only exhibit moderate slowdown when paired with other C3 members, even for six concurrent VMs. Another striking result can be found when co-scheduling C5 and C6 members. While three out of four combinations display the expected amounts of slowdown when co-scheduled, pairing C5 and C6 only results in significant slowdown for the C5 group. Applications in C6 do not suffer any slowdown at all when co-scheduled with non-network applications, this can be explained by their low CPU and disk requirements. The applications in C5 are slightly more sensitive to interference, particularly to co-scheduling with the disk heavy C4 cluster. Overall, these results indicate a strong need to differentiate between network application types when scheduling.

5.7 Scheduling

In this section we introduce two new scheduling techniques and compare the results to the findings in Section 4.8. Using the updated and extended information obtained in Sections 5.5 and 5.6, additional knowledge is added to the scheduling process. Improving the knowledge concerning application types and consolidation sensitivity opens up a new range of scheduling possibilities. To demonstrate the potential impact this new information can have, we introduce and evaluate several scheduling strategies. Attention is also given to the importance of correctly identifying all application types. Each technique is used to schedule a static set of applications on a fixed number of servers. In our evaluation, we focus on lowering the average slowdown experienced by co-scheduled applications.

5.7.1 Algorithms

To evaluate the scheduling potential of the updated application clusters in combination with slowdown predictions, we propose two new scheduling techniques. The first technique is named the *Sorted Cluster* scheduler and primarily uses clustering information in addition to runtime predictions. A second scheduler, the *Weighted Cluster* scheduler, improves upon this design by making more extensive use of runtime predictions. The *Bucket* scheduler introduced in Section 4.8 is also reevaluated on the new application set. This technique relies solely on runtime predictions and was found to be the best performer in the absence of network based applications. All three techniques share the same goal: reducing average slowdown by spreading the most sensitive workloads and finding appropriate consolidation candidates.

Sorted Cluster: Applications are divided into groups according to the clusters described in Section 5.6. These groups are sorted in descending order using the average slowdown found in Table 5.5. Applications within each group are sorted according to their predicted slowdown in descending order. Each potential server is given a weight by adding the recorded average slowdowns between the already assigned application types and the new candidate (see Table 5.8). Additionally, a maximum number of workloads per server is enforced to avoid overbooking. The ordered applications are scheduled by taking the slowest application from the slowest group first. Each new application is scheduled on the server with the lowest weight, provided that the server can still accept an additional workload. High level pseudo code of this placement strategy can be found in Algorithm 5.

Weighted Cluster: The scheduling and sorting process are identical to the Sorted Cluster scheduler. The sole difference lies in the manner in which the weight is calculated for each server. Instead of using only the data from Table 5.8, we also incorporate an application's slowdown prediction to scale this number. This creates a bigger weight for an application with a relatively higher slowdown prediction. The additional scaling helps to differentiate potential servers when scheduling large numbers of applications belonging to a single cluster. However, the predicted slowdowns represent a relatively high upper bound. Multiplying by such large numbers has the potential to reduce the effects

of the inter-cluster differences. To ensure only a subtle differentiation between application types, we use the much lower cube root of an application's slowdown prediction.

Input: *Servers, Applications, Profiling Data*

$Clusters \leftarrow$ Cluster($ProfilingData$, $Applications$);
$Predictions \leftarrow$ Predict($ProfilingData$, $Applications$);
$SortedApplications \leftarrow$ Sort($Clusters$, $Predictions$, $Applications$);

$CurrentServer \leftarrow 1$;
$CurrentServerWeight \leftarrow 0$;

foreach $Unscheduled \in SortedApplications$ **do**
  foreach $Candidate \in Servers$ **do**
    $CandidateWeight \leftarrow 0$;
    foreach $Scheduled \in Candidate$ **do**
      $Couple \leftarrow$ Slowdown($Unscheduled$, $Scheduled$)
      $CandidateWeight \leftarrow CandidateWeight + Couple$;
    end
    if $CurrentServerWeight > CandidateWeight$ **then**
      $CurrentServer \leftarrow Candidate$;
    end
  end
  Schedule($CurrentServer$, $Application$);
end

Algorithm 5: Placement strategy used by the *Sorted* and *Weighted Cluster* schedulers. The Slowdown() function returns different results for each scheduler.

5.7.2 Setup

We evaluate all scheduling techniques on the hardware platform described in section 5.4. The experimental setup consists of two application sets scheduled on four different hardware configurations ranging from 8 to 14 servers. Each server can accommodate a maximum of seven workloads without overbooking, leaving a single CPU core for the hypervisor. For each experiment we conduct a minimum of 5 iterations to obtain statistically significant results. The application sets used in the scheduling evaluation can be described as follows:

S1 : 50 applications consisting of an even mix from all application types. This application set corresponds to T1 described in Section 5.5.3.

S2 : Identical to S1 with the exclusion of the streaming applications assigned to clusters C5 and C6. This application set was created to analyze average slowdowns without the disproportionate effects caused by streaming applications.

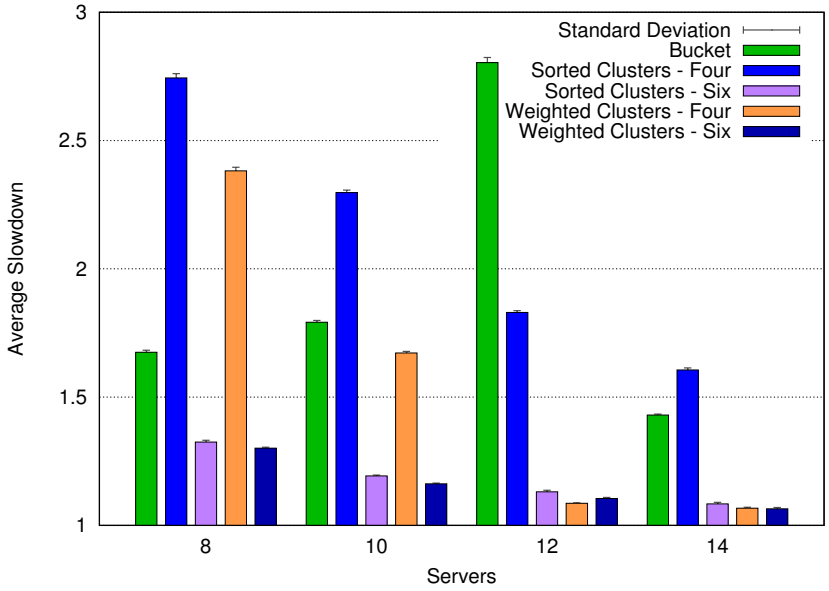

Figure 5.5: Average slowdown for each scheduler using S1.

Servers	8	10	12	14
Bucket	67.5% $\sigma = 8.1^{-3}$	79.2% $\sigma = 7.1^{-3}$	180.4% $\sigma = 20.0^{-3}$	43.0% $\sigma = 3.9^{-3}$
Sorted Cluster Four	174.4% $\sigma = 16.5^{-3}$	129.7% $\sigma = 9.9^{-3}$	83.0% $\sigma = 6.7^{-3}$	60.6% $\sigma = 8.4^{-3}$
Sorted Cluster Six	32.5% $\sigma = 6.7^{-3}$	19.3% $\sigma = 3.0^{-3}$	13.1% $\sigma = 6.0^{-3}$	8.4% $\sigma = 5.7^{-3}$
Weighted Cluster Four	138.2% $\sigma = 13.6^{-3}$	67.2% $\sigma = 5.6^{-3}$	8.6% $\sigma = 2.3^{-3}$	6.7% $\sigma = 3.8^{-3}$
Weighted Cluster Six	30.1% $\sigma = 4.3^{-3}$	16.2% $\sigma = 2.7^{-3}$	10.5% $\sigma = 4.0^{-3}$	6.5% $\sigma = 4.7^{-3}$

Table 5.10: Average slowdown for each scheduler using S1.

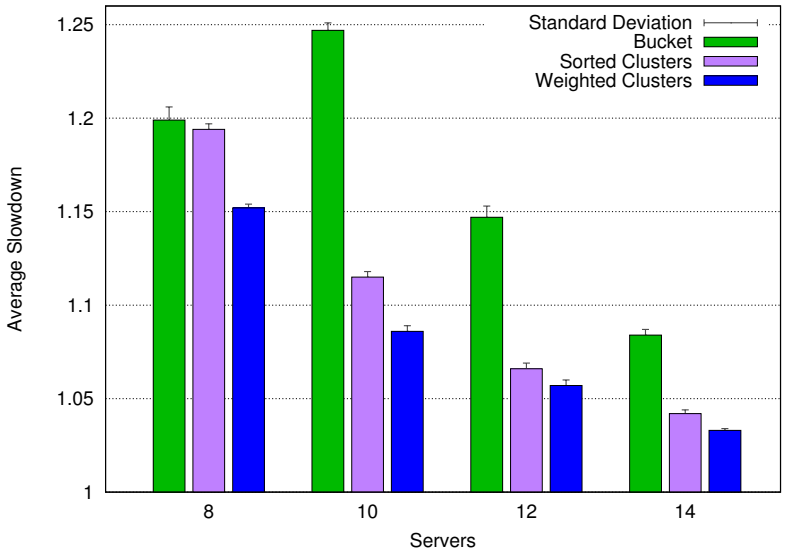

Figure 5.6: Average slowdown for each scheduler using S2.

Servers	8	10	12	14
Bucket	19.9% $\sigma = 7.3^{-3}$	24.7% $\sigma = 4.2^{-3}$	14.7% $\sigma = 6.0^{-3}$	8.4% $\sigma = 2.8^{-3}$
Sorted Cluster Four	19.4% $\sigma = 3.0^{-3}$	11.5% $\sigma = 3.0^{-3}$	6.6% $\sigma = 3.4^{-3}$	4.2% $\sigma = 2.4^{-3}$
Weighted Cluster Four	15.2% $\sigma = 1.6^{-3}$	8.6% $\sigma = 2.5^{-3}$	5.6% $\sigma = 2.9^{-3}$	3.3% $\sigma = 1.1^{-3}$

Table 5.11: Average slowdown for each scheduler using S2.

5.7.3 Results

The purpose of these experiments is to demonstrate the scheduling improvements that can be made by moving from a one-dimensional prediction based classification to a more fine-grained classification using application types. Our main focus in this section is determining if the presented application clusters can be successfully used to further reduce average slowdown. In Section 4.8 we found that adding accurate slowdown predictions adds valuable knowledge to the scheduling process. With the results presented in this section, we show that further improvements can be made by using correctly defined application types and the quantification of their interference effects. Average slowdown per application is presented for S1 and S2 in Figures 5.5 and 5.6, details can be found Tables 5.10 and 5.11. To put the performance of the newly proposed *Sorted* and *Weighted Cluster* schedulers in the proper perspective, results from the *Bucket* scheduling policy are also included. We

discuss our results in two parts. In the first, we look at the influence of the number of application clusters. In the second part, we discuss the differences between placement strategies without the streaming applications. Effectively reducing the number of identifiable application clusters to four.

In Figure 5.5 and Table 5.10 we find the average slowdown when scheduling the applications from test set S1. The *Bucket* scheduler is compared to the *Sorted* and *Weighted Cluster* schedulers using both four and six clusters. Both the *Bucket* scheduler and the schedulers based on four clusters are not able to correctly identify an important subset of the network intensive applications. The cluster performance analysis in Section 5.6.2 already highlighted the substantial impact when co-scheduling some of the network-based application types. Without this crucial information, schedulers are unable to reliably avoid the resulting performance degradation, causing high slowdown averages and unpredictable performance. Moving from four to six clusters allows the *Sorted* and *Weighted Cluster* schedulers to correctly identify the potential impact and take corrective measures. When sufficient application types are identified, the *Weighted Cluster* schedulers outperforms the *Sorted Cluster* schedulers by up to 3%. In Table 5.10 we can also note that, although most schedules result in a low standard deviation between separate runs, the *Weighted Cluster* produces the most stable results.

In Figure 5.6 and Table 5.11 we find the benchmark results using the smaller S2 application set. Due to the removal of the streaming applications, results are only presented using four application clusters. We find that the *Bucket* scheduler is still unable to provide predictable performance when network-based applications are included. In the test cases with 10 and 12 servers, average slowdown is more than two times higher than the cluster-based approaches. As expected, the *Sorted* and *Weighted Cluster* consistently reduce the average slowdown when additional servers are available to distribute workloads. The more intelligent *Weighted Cluster* scheduling strategy provides significant performance improvements for all server configurations. Compared to the *Sorted Cluster* results, average slowdown is further reduced from 19.4% to 15.2% when scheduling on eight servers. In Table 5.10 we note that the *Weighted Cluster* scheduler also results in the lowest standard deviation for all tested configurations.

Average slowdown numbers allow us to easily compare the large number of scheduler, server and application set combinations. However, this representation does not allows us to investigate the underlying slowdown distribution summarized by the average slowdown numbers. In Figure 5.7 we look at the slowdown per application using S1 on 8 servers and S2 on 8 and 14 servers. Applications are sorted according to increasing slowdown. Although we only present the results for a small number of options, similar trends can be observed for other server amounts. Several observations can be made when analyzing the results.

First, the performance differences can be attributed to a relatively small number of applications with large slowdowns. All techniques contain sufficient intelligence to manage slowdown for the majority of applications. Second, both clustering approaches require six clusters for optimal performance when scheduling with the S2 application set. When using just four clusters, the Weighted Cluster scheduler does create a much more even error distribution than the average slowdown numbers would suggest. Third, the Weighted Cluster scheduler not only results in lower average slowdowns for all configurations, it also distributes slowdown more evenly

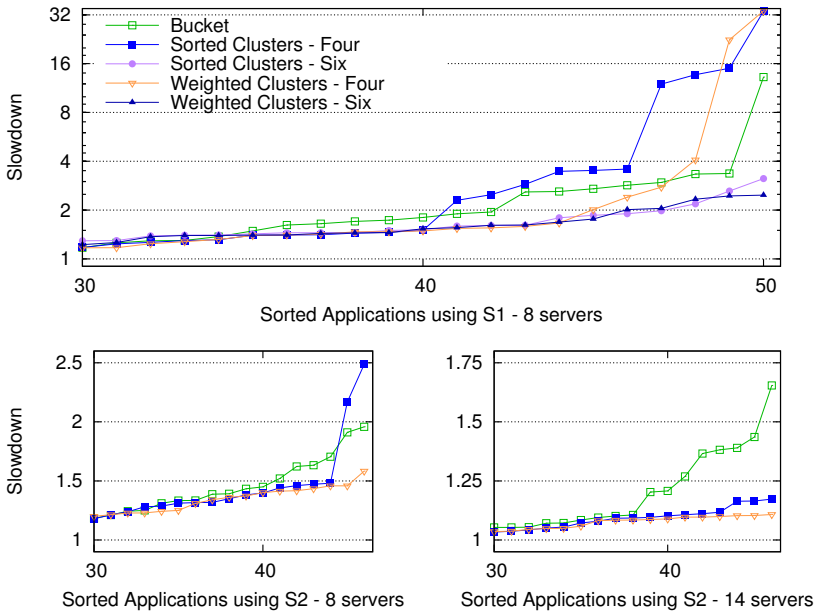

Figure 5.7: Slowdown distribution when scheduling with S1 and S2 on eight servers. Applications are sorted seperately per scheduling technique according to slowdown.

across the applications. The Sorted Cluster approach results in comparable performance for the majority of applications. The higher average slowdown can be traced to a limited set of applications that are not ideally distributed due to the lack of performance differentiation within clusters.

5.8 Conclusion

In this chapter, we addressed the issue of consolidating resource intensive VMs and the resulting variances in workload performance. We provided new and valuable information to datacenter schedulers by introducing accurate performance models and correct classification of applications. To demonstrate the potential benefits provided by this new information, we developed and evaluated a number of novel scheduling techniques. Correct identification of application types combined with consolidation preferences proves to be a powerful tool when scheduling resource intensive applications. Prediction of individual application slowdown can be used to further optimize scheduling performance.

The work in this chapter focuses on the performance interference caused by network I/O in consolidated VM workloads. Accordingly, we introduced an extensive benchmark suite and profiling framework. The gathered performance scores and metrics were used to evaluate both application modeling and clustering. Class SVM and WM were shown to have comparable performance on a realistic and extensive training set, both producing an average prediction error of 4.4%. Careful analysis of the prediction results revealed a complementary nature in the errors produced

by each techniques. A hybrid approach was introduced, further reducing the average prediction error to 3.3%. K-means clustering was used to identify several new network-related application types. With these new classifications, we were able to quantify the slowdown when consolidating different application types. We discovered an important difference in the interference effects caused by different network application types. Streaming applications can severely reduce the performance of response/request based network applications.

To determine the potential of model based scheduling, we developed and evaluated several novel scheduling techniques. Results from the scheduling experiments further highlight the importance of correctly identifying application types. Through intelligent scheduling using both application types and runtime predictions, we were able to significantly and consistently lower the average application slowdown for all tested configurations. The Weighted Cluster scheduler resulted in both lower average slowdowns as well as a more even slowdown distribution across all applications. These results conclusively demonstrate the added value of application types and runtime predictions in datacenter scheduling.

CHAPTER 6

Summary and Conclusions

The introduction of virtualization in modern datacenters has given rise to important scheduling challenges. Specifically, we have addressed the scheduling challenge where workloads with fixed performance guarantees are mixed with workloads that do not require such guarantees. A simulation framework was introduced to quantify the benefits such an approach can offer to virtualized infrastructure providers. We were able to determine that through the proper tuning of a limited set of parameters, increased utilization can be achieved without sacrificing on performance dependability. However, the full extent of the potential benefits could not be assessed due to lack of available performance interference models.

We addressed this issue using performance models based on the runtime characteristics of virtualized workloads. A benchmark framework was introduced to create resource usage profiles by gathering the metrics made available by the Xen hypervisor. An extensive and diverse set of representative virtualized workloads was profiled both in isolation and in combination with other workloads. Based on these profiles, we were able to predict performance degradation using several existing modeling techniques and novel combinations thereof. In addition, we used application clustering to identify several application types with distinct performance profiles. By introducing new scheduling techniques, we were able to achieve significant performance improvements for multiplexed workloads. We have made the following contributions:

- We introduced a simulation framework to evaluate the potential of multiplexing low- and high-QoS workloads on a virtualized cluster infrastructure. Opportunities were identified to add low-QoS workloads to a cluster node by monitoring the difference between available resources and actual requirements of high-QoS workloads. We introduced a scheduling policy based on a limited set of parameters so that a flexible tradeoff can be made between maximization of infrastructure utilization and workload interference. Depending on the requirements, optimal parameters were identified which can be selected to significantly increase utilization while causing limited interference with high-QoS workloads.

- A benchmark framework was introduced to gather performance profiles for an extensive and comprehensive set of applications. We analyzed the amount and type of training data required to build an accurate performance model for workloads with high resource requirements. Building upon the acquired insights, we further expanded the training data set by benchmarking rate limited applications. Through careful evaluation we have demonstrated the predictive capacity which can be derived from the proposed set of profiled metrics. The performance scores and metrics provide the training data required for both application modeling and clustering.

- Using the newly acquired performance profiles we created multiple sets of training data. Several existing modeling techniques were evaluated in their ability to predict an application's sensitivity to performance interference. Initially, we achieved the lowest average prediction errors by combining the prediction and classification capabilities of Support Vector Machines. Further experiments using an extended set of training data resulted in a similar average prediction error for Weighted Means. By carefully analyzing the nature of the largest errors produced by each techniques, we were able to identify an opportunity for a hybrid approach. Using this hybrid approach we were able to further reduce the average prediction error. We have successfully demonstrated that it is possible to create a robust and useful performance prediction model using relatively limited training data.

- Performance predictions do not provide insight into the complex interactions created by multiplexing different application types. A complete datacenter scheduling approach requires a method for assigning applications to a predefined application type with well known interference properties. An automated approach to create application clusters was presented using K-means clustering and the aforementioned performance profiles. Application clusters were used to automatically identify several distinct application types. With this new classification, we were able to quantify the slowdown when consolidating different application types. As expected, applications with significant disk resource requirements incur performance degradation when co-scheduled with similar application types. We also demonstrate a strong need to differentiate between network application types when multiplexing workloads. Network streaming benchmarks constitute entirely separate application types with unique performance interference effects on co-scheduled workloads.

- With the newly acquired information, datacenter scheduling can be improved in a controlled and deterministic manner. Each application can be assigned to an application type with a well defined interference profile. Applications within the same cluster can be further differentiated by their sensitivity to performance interference. To determine the potential of this new information, we developed and evaluated several novel scheduling techniques. First, we demonstrated the potential of using performance predictions or application types for non-network related applications. Average slowdown was up to five times lower than the random distribution of applications. Scheduling with slowdown predictions resulted in average slowdowns that were up to

three times lower than the cluster only approach. These experiments conclusively demonstrated the need to differentiate between applications contained within the same cluster. Second, we demonstrated that further scheduling improvements can be made by moving from a coarse- to a fine-grained classification of applications. To achieve this fine-grained classification, a new set of applications was added to the training data. By adding network related applications we were able to distinguish additional application types. Using the extended set of applications and their diverse interference interactions, we were able to evaluate new scheduling policies. These intelligent scheduling strategies resulted in significant performance improvements for all tested configurations.

In conclusion, we can state the following. Managing performance interference between virtualized workloads is an important challenge for modern datacenters. Without proactive measures, resources sharing workloads are susceptible to unexpected performance fluctuations. However, provided sufficient information is available for all workloads, performance interference can be successfully mitigated using datacenter scheduling. Due to the endless number of possible workload combinations, it is not feasible to gather performance measurements for all options. An efficient solution is needed to determine the consequences of consolidating different workload configurations. Using a relatively limited set of benchmark configurations, detailed workload profiles can be created by monitoring the resource usage of VMs in various degrees of consolidation. Using these profiles, models can be created to predict a workload's sensitivity to performance interference. Additionally, clustering allows workloads to be automatically categorized into application types with distinct performance characteristics. These new layers of information provide practical insight into the performance impact of workload consolidation. Performance can be optimized by separating workloads with similar resource requirements and high sensitivity to performance interference.

CHAPTER 7

Future Research

This thesis identified and solved several scheduling challenges in virtualized datacenters. Using performance models and application classification we developed a number of innovative scheduling techniques and demonstrated the potential benefits. The provided solutions incorporate several research domains, as such we can identify many potential avenues for future work.

- The clustering and scheduling experiments are currently limited to applications with considerable resource requirements. Adding applications with low resources requirements improved the prediction results but were not taken into account for the subsequent sections. Initial experiments with clustering these applications into several application types were very promising. Nonetheless, further research is still required before they can be used in scheduling.
- Preliminary results indicate that there can be large performance variances depending on the direction of the network data associated with a co-scheduled VM. Schedulers can be extended to consider whether a network heavy application is primarily sending or receiving data. The current dataset mostly consists of applications with relatively balanced network requirements. Further experiments and additional applications are required to reach a definitive conclusion.
- The clustering approach has already indicated that relying solely on predicted interference upper bounds can be ineffective for network based applications. We speculate that to truly create accurate network based application types we need to introduce a third classifier. By analyzing the network packets themselves we would be able to create more fine grained application types.
- The current approach relies on performance metrics which are measured by running applications in isolation during the profiling stage. Predictions and classification could be made dynamic, based on the metrics measured from concurrently running VMs.

- Creating and evaluating datacenter schedulers can be greatly simplified by using a simulated environment. However, we found that performance gains are difficult to quantify due to lack of available performance interference models. The acquired application profiles and performance models can be used to accurately simulate the interactions between workloads in a virtualized datacenter, taking into account performance degradation due to interference.

APPENDIX A

Samenvatting

Model-gebaseerde Plaatsing van Gevirtualiseerde Toepassingen

De introductie van virtualisatietechnologie heeft het mogelijk gemaakt om IT-infrastructuur op een efficiëntere manier te beheren. Door toepassingen te verpakken in virtuele machines werd de barrière verwijderd voor het consolideren, pauzeren en migreren van toepassingen samen met een geconfigureerd besturingssysteem. Deze ontwikkelingen geven aanleiding tot belangrijke uitdagingen met betrekking tot de optimale plaatsing van virtuele machines.

Het onderzoek dat gepresenteerd wordt in dit proefschrift kan opgedeeld worden in twee delen. In de eerste fase van het onderzoek hebben we gekeken naar de mogelijkheden om in de context van gevirtualiseerde infrastructuur (private en publieke datacenters) aanvullende computationele middelen beschikbaar te maken voor gebruikers. De beschikbare capaciteit in deze datacenters wordt toegekend aan eindgebruikers onder de vorm van virtuele machines (VM's). Deze VM's stellen een bepaalde hoeveelheid rekenkracht, geheugen en opslag ter beschikking. Vaak biedt men VM's aan met garantie dat de toegewezen middelen ten alle tijde beschikbaar zijn (zogenaamde Quality of Service (QoS) vereisten). De toepassingen die gebruik maken van deze gevirtualiseerde omgevingen vereisen echter niet steeds 100% van de gealloceerde middelen. Dit biedt mogelijkheden om aan overboeking te doen en de reeds bestaande infrastructuur beter te benutten. Door het bouwen van een simulator hebben we de mogelijkheden in kaart gebracht om met overboekte lage-QoS VM's het gemiddelde CPU gebruik op te drijven. Door gebruik te maken van een beperkt aantal parameters kan de plaatsing van virtuele machines op een efficiënte manier gestuurd worden om zowel de QoS vereisten te vrijwaren als de beschikbare hardware zo goed mogelijk te benutten.

We concludeerden dat het CPU-gebruik aanzienlijk verhoogd kan worden door het gebruik van overboeking. Echter, een beperkte inzage in de invloed van prestatie-interferentie maakt het moeilijk om de impact op applicaties met hoge-QoS vereisten te beheren. Het tweede deel van dit proefschrift zal zich daarom richten op het in kaart brengen van onverwachte prestatie-interferentie. We presenteren een nieuwe benadering om de gevolgen ervan te beperken door een beter planningsbeleid voor datacenters aan te bieden dat gebaseerd is op prestatiemodellen.

Hoewel virtualisatie zorgt voor voldoende isolatie op vele niveaus (bijv. veiligheid, storingen, ...), kan prestatie-interferentie nog steeds voor problemen zorgen, met name voor applicaties met hoge prestatievereisten. Elke VM krijgt een deel van de beschikbare bronnen toegewezen en maakt gebruik van de virtualisatielaag om bepaalde taken voltooien (bijvoorbeeld schijf of netwerk I/O). Wanneer meerdere VM's dezelfde hardware delen kunnen knelpunten ontstaan, zowel in de hardware als in de virtualisatie laag. Huidige datacenter planners negeren of onderschatten vaak de impact van prestatie-interferentie, waardoor gebruikers blootgesteld worden aan onverwachte prestatieschommelingen. Een oplossing moet verstrekt worden op het datacenter niveau, waar VM-migraties tussen systemen de mogelijk bieden voor een betere planning. Echter, voor deze benadering is inzicht nodig in het performantiepatroon van de afzonderlijke applicaties en hun effect op de VM's die het systeem delen. Via de integratie van deze informatie, door middel van prestatiemodellering, kan een intelligente plaatsing de impact van prestatie-interferentie sterk reduceren. In dit proefschrift presenteren we enkele innovatieve oplossingen om de gewenste performantiemodellen te creëren en de efficiëntie van datacenterplanning te verbeteren.

APPENDIX B

Publications

This chapter provides a short summary of the publications which were not incorporated in the main dissertation. The research presented in these papers mainly deals with grid computing.

- **Dynamic Grid Scheduling Using Job Runtime Requirements and Variable Resource Availability**
  S. Verboven, P. Hellinckx, F. Arickx and en J. Broeckhove
  Proceedings of EuroPar 2008 Conference, 26–29 August, Las Palmas de Grand Canaria, Spain, pp 223–232

Abstract

We describe a scheduling technique in which estimated job runtimes and estimated resource availability are used to efficiently distribute workloads across a homogeneous grid of resources with variable availability. The objective is to increase efficiency by minimizing job failure caused by resources becoming unavailable. A fault-aware scheduling mechanism attempts to map jobs onto resources with sufficient availability. Both the scheduling technique and the implementation, called PGS (*Prediction based Grid Scheduling*), are described in detail. We present results for a set of sleep jobs, and compare workload turnaround time with a first-come, first-serve scheduling approach.

- **Runtime Prediction based Grid Scheduling of Parameter Sweep Jobs**
  S. Verboven, P. Hellinckx, F. Arickx and en J. Broeckhove
  IEEE Computer Society, APSCC 2008, 9–12 December, Yilan, Taiwan, pp. 33–38

Abstract

This paper examines the problem of predicting job runtimes by exploiting the properties of parameter sweeps. A new parameter sweep prediction framework GIPSy (*Grid Information Prediction System*) is introduced. Runtime predictions are made based on prior runtime information and the parameters used to configure each job. The main objective is providing a tool combining development, simulation and application of prediction models within one framework. Prediction results are evaluated using a quantum physics problem. A previously introduced scheduling technique and implementation, called PGS (*Prediction based Grid Scheduling*), is extended and integrated with GIPSy.

- **Runtime Prediction in Desktop Grid Scheduling**
  P. Hellinckx, S. Verboven, F. Arickx and J. Broeckhove
  In "Parallel Programming and Applications in Grid, P2P and Network-based System", F. Xhafa ed., IOS Press Series on "Advances in Parallel Computin", 2009, pp 204–231

Abstract

The efficient allocation of tasks to available resources is a key feature of resource management and scheduling systems. Mapping applications or tasks to physical systems is subject to many constraints, requirements and objectives. Some of these derive from the applications themselves e.g. complex work flows or hard deadlines, while others arise from the nature of the resources e.g. available computational capacity. If the infrastructure has dynamic resource availability, such as a desktop grid, resource uptime and application runtime become critical factors in the scheduling algorithms.

The scheduling mechanism presented in this paper maps job runtimes onto resource availability windows. The main focus lies on runtime prediction and its application in the prediction-aware mapping of jobs onto available resources. The ultimate objective is a reduction in the number of jobs prematurely interrupted due to insufficient resource availability. We introduce a comprehensive framework for efficiently executing parameter sweep applications based on both runtime prediction modeling techniques and resource availability. The feasibility of using GIPSy (Grid Information Prediction System) to improve workload turnaround times is evaluated by integrating its runtime predictions into the Prediction based Grid Scheduler (PGS).

- **Evaluating Nested Virtualization Support**
  S. Verboven, R. Van den Bossche, O. Berghmans, K. Vanmechelen and J. Broeckhove
  The Tenth IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN 2011), Innsbruck, Austria.

Abstract

Recent evolutions in the hard- and software used to virtualize x86 architectures have led to a rising popularity for numerous virtualization products and services. Virtualization has become a common layer between an operating system and physical hardware. Virtualized systems have already transparently replaced many physical server setups while virtual machines themselves are increasingly popular as a means to package, distribute and rapidly deploy software. Nested virtualization –running a virtual machine inside another virtual machine– seems a logical next step. Presently, however, there is little to no information available on nested virtualization support by widely used virtualization applications such as VMware, VirtualBox, Xen or KVM. In this contribution, we evaluate the feasibility of nested virtualization using a range of currently available hard- and software products. We briefly explain the different techniques behind the tested virtualization software and an analysis is made whether particular nested virtualization setups should be feasible. All theoretically possible options are explored and the results are presented and analyzed. We conclude with a presentation of the results of some initial performance experiments. Our results show that nested virtualization is currently possible in selected combinations with promising results regarding recent evolutions in hardware assisted virtualization.

Bibliography

Scientific resume

Education

- 1996–2002, Secondary education
- 2003–2005, Bachelor Computer Science (Kandidaat in de Informatica), University of Antwerp, Belgium
- 2005–2007, Master Computer Science (Licentiaat in de Informatica), University of Antwerp, Belgium
- 2007–2013, PhD Candidate in Computer Science

Teaching

Computer Science (BSc), University of Antwerp:

- Programming I, 2007–2010
- Introduction to Distributed Systems, 2008–2010
- Distributed Systems, 2011–2012
- Operating Systems, 2010
- Computer Systems and Architecture, 2011
- Introduction to C++, 2011
- Introduction to Programming, 2011–2012

Computer Science (Ma), University of Antwerp:

- Cluster Computing, 2008–2012
- Distributed Systems, 2009–2010

Biological and Bio-engineering Science (BSc), University of Antwerp:

- Computer Skills, 2011–2012

Publications

- Sam Verboven, Peter Hellinckx, Frans Arickx and Jan Broeckhove, Runtime Prediction based Grid Scheduling of Parameter Sweep Jobs, *IEEE Computer Society, APSCC 2008, 9–12 December, Yilan, Taiwan, pp. 33–38*
- Sam Verboven, Peter Hellinckx, Frans Arickx and Jan Broeckhove, Dynamic Grid Scheduling Using Job Runtime Requirements and Variable Resource Availability, *Proceedings of EuroPar 2008 Conference, 26–29 August, Las Palmas de Grand Canaria, Spain, pp. 223–232*
- Peter Hellinckx, Sam Verboven, Frans Arickx and Jan Broeckhove, Runtime Prediction in Desktop Grid Scheduling, *In "Parallel Programming and Applications in Grid, P2P and Network-based System", F. Xhafa ed., IOS Press Series on "Advances in Parallel Computin", 2009, pp. 204–231*
- Peter Hellinckx, Sam Verboven, Frans Arickx and Jan Broeckhove, Predicting Parameter Sweep Jobs: From Simulation to Grid Implementation, *Third International Conference on P2P, Parallel, Grid and Internet Computing (3PGIC-2009), 16–19 March 2009, Fukuoka, Japan. IEEE computer Society Press, 2009, pp. 402–408*
- Sam Verboven, Peter Hellinckx, Frans Arickx and Jan Broeckhove, Runtime Prediction based Grid Scheduling of Parameter Sweep Jobs, *Journal of Internet Technology Vol. 11, No. 1, 2010, pp. 47–54*
- Sam Verboven, Kurt Vanmechelen and Jan Broeckhove, Multiplexing Low and High QoS Workloads in Virtual Environments, *Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science, Volume 6253, 2010, pp. 175–190*
- Sam Verboven, Ruben Van den Bossche, Olivier Berghmans, Kurt Vanmechelen and Jan Broeckhove, Evaluating Nested Virtualization Support, *The Tenth IASTED International Conference on Parallel and Distributed Computing and Networks, 15–17 February, 2011, Innsbruck, Austria.*
- Sam Verboven, Kurt Vanmechelen and Jan Broeckhove, Black Box Scheduling for Resource Intensive Virtual Machine Workloads with Interference Models, *Future Generation Computer Systems, Volume 29, Issue 8, October 2013, pp. 1871–1884*