UNIVERSITÀ DEGLI STUDI DI ROMA "TOR VERGATA"

FACOLTA' DI INGEGNERIA

DOTTORATO DI RICERCA IN INFORMATICA ED INGEGNERIA DELL'AUTOMAZIONE

XIX Ciclo

Resources Allocation for Virtualized Architectures

Paolo Campegiani

A.A. 2008/2009

Docente Guida: Prof. Salvatore Tucci

Coordinatore: Prof. Daniel Pierre Bovet

Resources Allocation for Virtualized Architectures

Paolo Campegiani

April, 2009

Contents

List of Figures

List of Tables

Acknowledgments

I'd like to thank many people that helped me all along the pursuing of my Ph. D.

Professor Salvatore Tucci was my tutor during these years, and allowed me to freely explore my interests. He says that the best way to do research is by finding a topic of interest and developing it, and if I could say the best way to understand this vision is to apply it on everyday's work.

Professor Franco Maceri allowed to run the simulations in the scientific lab of Dipartimento di Ingegneria Civile, that it happens I also manage as system administrator hoping not causing too much long downtimes. Facing high computational problems from the system's point of view has developed in me some sensibility to performances problem that shines through this work

(or, at least, I hope so).

Professor Aurelio Simone, as the director of Scuola IaD, put me in charge of designing and administration of their network infrastructure, an exciting challenge that he generously allows me to do without interrupting my Ph. D. due work.

Professor Francesco Lo Presti is the one who more encourages me and follow my progress (or lack thereof), really helping me in finalizing this work.

I'd also like to thank people and staff of Dipartimento di Ingegneria Civile and from Scuola IaD for their kindness, and especially Eusebio Giandomenico and Marco Orazi for their suggestions and support.

1

Introduction

As many other technologies and paradigms in the computer science field, virtualization has a long history with periods with great momentum and periods when it has been put in then background. The advent of massive and economic computer power, as predicted by Moore's Law, has finally resulted in the availability of system level virtualization technologies on commodity hardware. This will lead to a complete new class of problems, from provisioning to deployment, that arise when virtualization is intended as an architectural asset that could bring value to the computing platform, an asset upon which build value added

services.

In this dissertation, I investigate a problem that could be expressed as: *given a number of virtual machines and some physical machines, each described respectively by a demand vector and a resource vector, which is the best allocation of the former to the latter, for a given metric*?

For a large data center, an example of metric would be to minimize the number of physical machines devoted to host virtual machines, giving sufficient resources to each virtual machine. By minimizing this number, the data center could increase the efficiency of the physical machines, and the underlying virtualization technology will prevent each virtual machine to interfere with others, from both a performance and a security prospective.

It will shown that this problem, stated in the most generally form, is NP-hard (it's a generalization of the classical 0/1 knapsack problem), and its complexity is daunting, requiring to define an heuristic to find an approximate solution. We will also present a genetic algorithm that appears promising in tackling this problem.

The dissertation is organized as follows.

In Chapter 2, the different virtualization techniques are presented, analyzing them from an architectural point of view,

broadly classifying them in two categories: techniques that does not rely on hardware feature to support virtualization, and techniques that leverage on.

In Chapter 3, we move from a technology point of view towards an architecture centric one, analyzing the performances problems that virtualization faces. We put virtualization as an asset of multi-tier distributed systems, and we describe it as a fundamental block for autonomic computing. Current works in this field lack of some degree of generality, and when we extend the current available frameworks.

In Chapter 4, the mapping problem is formally defined, we analyze its computational complexity, and develop some heuristics [44] to solve it, comparing them to a genetic algorithm [45] we also propose. In this chapter, we see that the mapping problem is a generalization of the knapsack problem, and we briefly analyze the scarce literature on generalization of knapsack problems.

In Chapter 5, we show simulation results for some interesting data sets.

Chapter 6 ends this dissertation, briefly recalling the results we have found and proposing future enhancements.

2

Virtualization techniques

2.1 A general definition of virtualization

Virtualization could be defined as a two phase process. In the first phase, some resources of the same kind will be grouped together, hiding physical boundaries; in the second phase, a portion is carved out from this aggregated compound and presented

to an user. There are many types of virtualization, depending on the type of the aggregated resource:

- Virtual LAN (VLAN): a VLAN [23] is a set of hosts that communicate as if they were on the same wire, unregarding their physical location. Even when the hosts are on different physical segment of the same LAN, the configuration made on network devices like switches and routers allows the hosts to share the same virtual segment, so broadcast packets are forwarded only on the VLAN. This will increase security, by avoiding unauthorized hosts to connect to the virtual segment, and allows for the definition of per-segment Quality of Service policies;

- Storage Virtualization: a bunch of storage resources (disks or tapes) are grouped together, and the access to them is selectively defined by a management function. In a Network Attached Storage (NAS) environment, and more in a Storage Area Network (NAS) [55], it is possible do carve out some resources and allow one or more hosts to access them. As a result, the hosts are computing nodes that are attached to the data. This allows for better and cheaper data consolidation, backup and security;

- Runtime environments: this is the case of many web based

applications, running on Java or Flash. As an example, when the user downloads a Java applet via the browser, the applet is executed in the context of a Java Runtime Environment (JRE) [74]. The JRE virtualizes the computing resources to the applet in the sense that the applet is written in a so called bytecode, a machine language that the JRE translates into real operations for the underlying target processor. As a result, the same applet could be executed over different processor architectures, as long as a JRE is provided. Besides this, the JRE defines a sandbox that has some security constraints, like an applet cannot access system files on the target machine.

All these examples, no way exhaustive, shows some of the benefits of virtualization. By adding an intermediate layer between physical resource and resource demand, it's possible to multiplex, demultipex and routing requests to a single management point, achieving better scalability, manageability, performances and security.

2.2 Virtualization at the operating system level

Virtualization at the operating system level has been implemented for the first time on the IBM S/360 system [8]. In an fundamental article on virtualization, Popek and Goldberg [86] defined the formal requirements for a virtualization architecture. We will base our exploration and taxonomy of virtualization techniques on that paper, so it's worthy to recap it.

First, it's defined the concept of Virtual Machine Monitor (VMM) as a layer that separate the Virtual Machine (VM) - that is, the operating system to be executed - from the underlying hardware, as shown in figure 2.1.

Some of the instructions of the VM could trap, that in the original article is defined by saving the program counter to a specified location and then jumping to the address contained in another location, where a trap routine is to be executed, with the machine registers saved. The trap routine will do its own job, then it restores the registers and return control to the address saved in the first place. It's possibile to define not blocking trap routines. This mechanism is the precursor of today's system calls, where a program request the operating system to perform an operation on hardware resources.

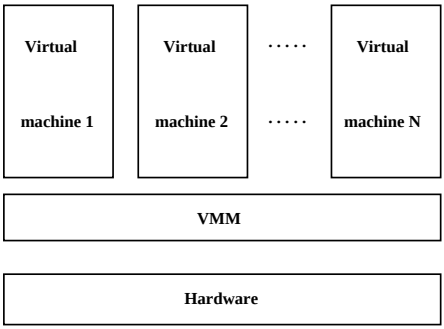

Figure 2.1: VMM architecture.

Trap are instrumental to classify instruction in three different groups:

- privileged instructions: are the instructions that causes a trap;
- sensitive instructions: they came in two different types, control sensitive instructions and behavior sensitive instructions. To define them in terms of current architectures, we define these instructions as the ones which changes the processor mode (or returns it) or which execution depends on the real memory address of their operands;
- innocuous instructions: all the remaining.

The VMM should have some properties to allow for the execution of a VM on top of it:

- Efficiency: Every instruction that is innocuous is executed directly by the underlying hardware, with no intervention of the VMM;
- Resource control: The VM cannot change its resources quota: every request for more resource is mediated by the VMM;
- Equivalence: Every program executed in the context of a VM performs in an almost indistinguishable manner, as if it were executed without a VMM interposing between the VM and the hardware. In this context, almost indistinguishable means that it's allowed a certain degree of deviation, as performances may be a bit worse and resources availability could be not identical (because the VM cannot access directly the hardware).

The work of Papek and Goldberg is fundamental as they proof the following theorem:

For any [conventional third generation] computer, a virtual machine monitor could be constructed if the set of sensitive in-

structions is a subset of the privileged instruction.

The theorem still holds for current architecture, and we use it as a criteria to discriminate between virtualizable processor architecture (or so called virtualization friendly) and the not virtualizable ones: a processor architecture is virtualizable if and only if the execution of every sensitive instruction eventually result in a trap, as the trap routine could be implemented by the VMM.

It will be shown later that, surprisingly, the Intel x86 architecture is not virtualization friendly.

2.3 Virtualization techniques

In spite of the unifying definition stated above, there are some different ways to virtualize an operating system. Broadly speaking, there is a trade off between the resulting performances and the spectrum of processors architectures that could be virtualized: to achieve speed it's usually necessary to focus on a specific instruction set and presenting the virtual machine a more generalized and less customizable abstraction of physical hardware, whilst the flexibility of having more instruction sets or virtualized resources usually incurs in performances penalties. We

identify four different virtualization techniques.

2.3.1 Binary translation

In this approach, a software layer translates operations from the virtual machine set to the physical machine set, allowing for code optimization and translation cache efficiency. The virtualization layer could do a so called cross virtualization, where the virtual machine instruction set and physical machine instruction set are completely different - requiring to completely translate the former into the latter - or a partial virtualization, where innocuous instruction are executed directly by the hardware (in a context set up by the VMM) and critical ones are translated by the VMM that operates as a resources' broker.

2.3.2 Para-virtualization

The operating system of the virtual machine is modified in such a way that every system call that should have accessed the hardware is instead mapped in an system call executed by and in the context of the VMM. The modification of the to be virtualized operating system could be unfeasible when it's released only in closed source format.

2.3.3 Hardware assisted virtualization

The instruction set has been augmented with operations that encompasses portion of machine code. This sections are executed in a virtual machine context, which is different from the physical machine context. The VMM has some degree of control over the operations made by a specific virtual machine, ranging from a no trust relationship (every I/O operation performed by the virtual machine is trapped and results in the execution of the VMM that operates as a control interface) to a total trust relationship, where the virtual machine could directly access every hardware in the system. The latter results in increased speed and diminished security.

2.3.4 Light weight virtualization

The operating system of the physical machine is changed to allow different and not-communicating namespaces for the different resource classes. As a result, there are some zones (to use a typical terminology) and each one has its own file system, users, processes namespace and hardware view. It could be argued that this approach is not a virtualization, mainly because it lacks generality (all the running instances are sharing the same operating system), but it's widely adopted to solve

some problems that otherwise require a traditional virtualization technique, while experiencing nearly no performance penalties.

2.4 VMM implementations

A number of competing products, both open and closed source, are available as VMM. In this section we see the most representative of them, focusing on the adopted virtualization technique.

2.4.1 QEMU

QEMU [16], written by Fabrice Bellard, is an open source machine emulator and virtualizer. It could operate as a virtualizer, when the virtual machine instruction set and physical machine instruction set are the same, or as an emulator, capable of translating instruction set from seven different processor architecture to some target architecture, plus virtualizing system hardware to allow for a complete operating system virtualization.

QEMU is a dynamic translator, i.e. the code translated is stored in a translation cache where it could be reused to increase efficiency. The translation process of QEMU is fully documented in [38], and it will be briefly shown here as it highlights the

general approach for binary translation.

Consider the following PowerPC instruction:

```
addi  r1,r1,-16        # r1 = r1 -16
```

that must be translated into Intel x86 code. First, there will be generated some micro operations, that are independent of the final target:

```
movl_T0_r1          # T0 = r1
addl_T0_im -16      # T0 = T0 - 16
movl_r1_T0          # r1 = T0
```

the T0 and T1 register are typically stored in host register due the optimization made by the GCC compiler. The first of the micro operation is typically coded as:

```
void op_molv_T0_r1(void) {
        To = env->regs[1];
}
```

where `env` is the structure containing the CPU state of the virtual machine.

The code generated is then translated in physical machine code by the GCC compiler, and the result will be (for an Intel x86 target):

```
# movl_T0_r1
```

```
# ebx = env->regs[1]
mov 0x4(%ebp), %ebx

# addl_T0_im -16
# ebx = ebx - 16
add $0Xfffffff0,%ebx

# movl_r1_T0
# env->regs[1]= ebx
mov %ebx, 0x4(%ebp)
```

QEMU is a dynamic translator as it uses a 16 MByte cache that holds the most recently used translation blocks (TB). After the execution of every TB, the next instruction to be executed will be determined by examining the state of the emulated CPU; if the jump point is in the cache, the code is executed directly, otherwise the translation process takes place. A TB could be patched directly to the logical following one when the jump destination is known.

More complex problem arises with self-modifying code, as the applications written for the Intel x86 architecture does not signal cache invalidation that could trigger the removing of a stale TB.

With a dynamic code translator is possible to execute an ap-

plication written for a different processor architecture, but an entire operating system requires the virtualization of the hardware. QEMU allows for a limited set of virtualized hardware. It's possible to have up to two EIDE hard disks, a basic video VGA card, one or more Fast Ethernet NIC; while it's also possible to connect directly the USB subsystem of the virtual machine to the physical USB subsystem.

The virtualized hard disks are mapped as file on the physical machine. This will result in a significant performance loss, as every I/O request made from the virtualized machine will traverse the virtualized operating system stack, resulting in a sequence of I/O operations intertwined with virtualized OS operations, and each I/O operation will ultimately result in a I/O operation made on the image file on the physical system, requiring for being made traversing again the stack of an operating system, in this case the physical one. The final result is that the data path is doubled. QEMU has some flexibility in the image file format, it's possibile to have a copy-on-write format file, but this architecture won't help for performances.

The network card emulation has some interesting features. Each virtual machine could have one or more NICs, and these NICs could be logically organized in several ways. It's possible to have two virtual machines on the same private LAN,

completely hidden from the rest of the world, bridged on the physical LAN or even on a UDP multicast network that could span several physical machines.

2.4.2 VMWare

VMWare [24] is the market leader in the virtualization field, thanks to its performances and management tools. Products from VMWare range from VMWare Player, that is only capable of run a virtual machine, to the VMWare Infrastructure suite, that has the ability to manage resources allocation, performing live backup of running virtual machines, moving them from a physical machine to another with very little service interruption.

VMWare reacted to the introduction of the open source Xen hypervisor (discussed below) by releasing its VMWare server free but closed source, to gain and maintain market share at the expenses of the newcomer. Unfortunately, the license of VMWare server dictates that benchmark are possible only when the methodology has been approved by VMWare Inc., and as a result of this there are very few scientific papers on the internals of this VMM.

One of these is [30], where the focus is in contrasting that hardware assisted virtualization (hardware VMM in the article)

has overall better performance.

To achieve maximum speed, it's imperative that, as stated in [86], most part of the code is executed directly by the underlying physical processor, but this is impossible with the Intel x86 processor architecture, as there are instructions that are sensitive but not trappable. As an example, the Current Privilege Level (CPL) could be obtained by reading the low two bits of the code segment selector register (`%cs`), and the `popf` instruction ("pop flags") executed by a privileged process could modify the IF flags that governs the interrupt delivery, an operation that an unprivileged guest cannot do [92]. As a result, it's necessary to have a binary translator that, for such virtualization unfriendly operations, simulates their execution in a virtual context. The translator adopted by VMWare is:

- Binary: its input is Intel x86 code;
- Dynamic and on demand: translation happens at runtime, and only when code is about to be executed;
- System level: the are no assumption about the guest code, the ABI is the x86 Industry Standard Architecture (ISA);
- Subsetting: the input is the full set of Intel x86 operations, the output is a subset of them (typically only user-mode

instructions);

- Adaptive: translated code is adjusted in response to guest behavior change to improve efficiency.

The last property is worthy noting. When a CPU encounters a trap for a privileged instruction, it has to jump to a trap routine (typically an operating system entry point) to deal with it, and this could be expensive. A binary translator could avoid it, by replacing the original code with a routine (that, being executed by a program, is in user mode and not in kernel mode). As an example, the `rdtsc` instruction for the Intel Pentium architecture, takes 2030 cycles for a classical trap and emulate execution, and only 216 for the binary translation. This could deal with a minor part of the sensitive instructions, as loads and stores could access sensitive data as page tables. The adopted approach is that an instruction is translated identically (i.e., not translated) and executed by the physical processor. If a trap happens, next time the same instruction will be re-translated to avoid the trap, maybe invoking an interpreter.

VMWare has put a lot of effort in the management and configuration tools, both for a single system and for an entire data center. Although the only virtualized operating system are the ones for the Intel x86 architecture, for each virtual machine is

possible to define an arbitrary number of virtualized peripherals, including storage systems, network cards, video cards and USB devices. Hard disks can be mapped into a file image, a disk partition or an iSCSI target [13] to achieve maximum performances. The network could be configurated to have a virtual machine that has an host-only network (i.e., it communicates only with the physical machine it's instantiated on), a NAT network (where the physical machine acts as a Network Address Translator), or to have a unique, externally accessible IP address.

The real value of the VMWare suites comes with the VMWare Infrastructure, that allows for a central administration of hundreds of virtual machines, over dozens of different physical machines, allowing for load balancing, high availability and live migration (moving a virtual machine from one physical node to another [82]) with little service disruption.

2.4.3 Xen

Xen [25] was originally developed at the University of Cambridge Computer Lab [26] as a framework to have an homogeneous computing environment over a high performance computing grid. Performances were so good that a company was

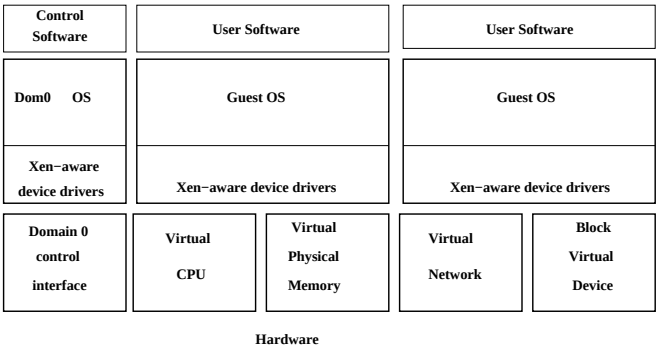

Figure 2.2: Xen architecture.

founded to gain paying customers for management tools (the hypervisor itself is released under the GNU Public License); later the company has been acquired by Citrix.

In the Xen language, both physical and virtual operating systems are called domains, with dom0 indicating the hypervisor and domU for the unprivileged domains, i.e. the virtual machines. The figure 2.2 shows the Xen architecture [36].

Xen adopts the para-virtualization approach, borrowed from the Denali system [102]: the application ABI remains unchanged, but the virtualized operating system has some modifications (in the order of thousands of lines of code), with the introduction

of hypercalls.

An hypercall is essentially a way to control interactions between a virtual machine operating system and the physical machine operating system. The hypercall interface allows domains to perform a synchronous software trap to perform a privileged operation, analogous to the system calls found in the operating system. Data transfers are managed via I/O rings, essentially a producer-consumer buffer of I/O file descriptors, with a general interface that could be used for almost every kind of I/O device interaction.

CPU scheduling between different domains is made with three different schedulers as Xen 3.0: the Borrowed Virtual Time (BVT) scheduling algorithm [59], that is work-conserving and capable of a low latency wake up when a domain receives an event; the Simple Earliest Deadline First (sEDF) that could be both work-conserving and not work-conserving, but lacks a global load balancing between different CPUs; the Credit Scheduler that is also global load balancing although not preemptive, and has a scheduling period hard-coded at 30 ms [47].

Network interfaces are quite complex [28]: the foundation of the architecture is a Virtual Firewall Router (VFR), with each domain using one or more Virtual Network Interfaces (VIF). The end result is that each domain sees one or more typical

NIC, but the administration of the VFR could be challenging.

Storage systems for the domU are modeled as Virtual Block Devices (VBD): the dom0 could map them into files, partitions or LUNs. It's also possible to black list a PCI device for the dom0, leaving it in the exclusive access of one or more domU.

Xen has the ability to perform a live migration, with very little QoS loss [49]. On [97] it's exposed an architecture that allows for migration over a MAN/WAN, at the expense of having a dedicated communication circuit. On [43] it's shown an extension that also allows for migration of the local file system (hypervisors assume that the local file system could also be accessed from the destination physical machine, requiring a NAS or SAN infrastructure).

Checkpointing, as the ability to save and restore often from a saved image that contains also the persistent state, is under development, allowing for a global checkpointing of an entire cluster of virtualized machines [54].

Xen performances are of the utmost interest, as the paravirtualization has a very low impact, at the cost of requiring to change both the dom0 and the domU operating system. This is infeasible for operating systems released only in binary form (like Microsoft's Windows line of products), but Xen also supports the hardware assisted virtualization described below.

Xen has found its way in the mainline Linux kernel, after some time where its integration with the operating system was the premiere feature of enterprise oriented Linux distribution as Red Hat RHEL and Novell SuSE server.

2.4.4 Hardware assisted virtualization

The Intel x86 architecture is not a virtualization-friendly one. As a result, until some years ago the only available hypervisors are binary translator (as VMWare) or para-virtualizer (as Xen). In 2006, Intel has announced the VT-x architecture for hardware assisted virtualization for the x86 processor family, and the VT-i for the Itanium family [12].

With the VT-x extension, there are available two new CPU operations, the VMX root operation and the VMX non-root operation.

The VMX root operation is intended for a VMM, and it's very similar to a traditional IA-32 operation. VMX non-root is intended to support and isolate the execution of a virtual machine, allowing the VMM to define a degree of trust for the virtual machine, granting some direct interactions with the hardware.

A VM entry is the transition from the VMX root operation

to the VMX non-root operation, the opposite transition is a VMX exit. The Virtual Machine Control Structure (VMCS) manages these transitions, being composed of a guest state area and an host state area. Processor state is loaded from the guest-state area on every VM entry, while it's restored from the host-state area on every VM exit. Exits happen always for some instructions, for others it depends on some variables and flags in the VMCS, that could be set only in the VMX root operation mode. As an example, the VMCS could define how to deliver interrupts (every interrupt results in a VM exit with no mask, or the guest is able to receive interrupts), choose to allow the guest to directly access some special register (that defines paging or floating point operation mode), which exceptions cause a VM exit, which I/O operations are allowed (by defining acceptable I/O port range).

This flexibility allows for a finer grain of control, because a VMM could choose to give a specific virtual machine more privileges, resulting in fewer VM exits and entries. As noted in [30], each entry or exit is analogous to a context switch, resulting in some performance losses. The exact penalty varies a lot, because it depends on the number of privileged instructions (in [30], one test is based on the virtualization of a code that creates forty thousand processes, a very uncommon appli-

cation behavior). Nevertheless, the performance problem must be addressed.

As a result of the growing concerns, the second generation of virtualization capable processor has some new features. AMD, that developed a similar architecture called Pacifica, presented the Barcelona processor, that has a third level cache and a virtualized address translation, instead of a shadow paging, that should substantially reduce the memory performance loss. Intel has instead developed the Virtualization Technology for Directed I-O [10], that allows for a direct remapping of DMA transfers and device generated interrupts.

It must be noted that an hardware assisted virtualization is the only way to virtualize an operating system that's available only in closed source form (like Microsoft Windows series), but to get the best performance it could be required to use specific drivers in the guest kernel. The so called para-virtualized (PV) drivers are drivers engineered to work optimally in a guest environment, where there's no need to access directly the hardware (and, in fact, trying to do that will usually causes a VM exit) [29].

2.4.5 Lightweight virtualization

The VMM seen so far allows for multiple and different operating systems hosted on the same physical machines, giving a high degree of flexibility. In some scenarios, there is no need for using different operating systems (or even different version of the same), it's sufficient to have multiple views of the same system. This approach is the generalization of the jail or chroot security feature found on Unix system: a process is restricted to interact with a subset of the system files, so a compromission of it wouldn't allow the attacker to manipulate others program files and resources. From the system point of view, the files namespace has been split, as two different processes may refer to different files even when they use the same (local) name. If this splitting is extended to all the system's resources, we have a lightweight (or container based) virtualization [96].

Example of this are the OpenVZ extension to Linux kernel [18], the Linux-VServer project [15] and the Solaris 10 operating system [20].

OpenVZ calls each autonomous namespace as a virtual environment (VE), called zones in Solaris. With container based virtualization there's only one operating system running on the hardware, and each container can use a specific amount of sys-

tem resources. OpenVZ defines this resources limits as bean counters, and they are in place for each possible resource type. In fact, resources management within containers is far more simple, as there's only one operating system that must be enhanced to govern that, making also possible to change these limits even at run-time. Overhead is also negligible [95], allowing for instantiating even hundreds of containers in the same physical machine, making this solutions particularly appealing for Internet Service Providers where each hosted site could coincide with a virtual container.

2.4.6 Other VMMs

There are many VMM solutions today, from research prototypes to production ready infrastructures. We cite here some of them presenting interesting features:

Terra: Terra [62] is a VMM that allows for Trusted Computing. A virtual machine could be instantiated as an open-box, allowing for data access and modification from the administrator of the physical machine, or as a closed box, where these operations are prohibited. Also, the Terra hypervisor automatically analyzes the images of a closed box virtual machines to get sure they have not been tam-

pered. This experimental approach allows for high sensitive secure virtual machines (e.g. voting machine) to be allocated over commodity hardware;

P.R.O.S.E.: the Partitioned Reliable Operating Systems [63], based on the Logical Partitioning (LPAR). The hypervisor, rHype, is a para-virtualization engine that uses a round robin fixed slot CPU scheduler. This simple scheduler reduces the OS interference [64], which happens when there's some jitter in the execution sequence of different virtual machine, a plague that is more evident on general purpose VMM like Xen or VMWare as the VMM are a component of a general purpose operating system. This lacks of strict timing coordination could easily destroy aggregated performances in a High Performance Computing scenario;

Virtual Box: it's a GPL released binary translator made by Innotek and now developed by Sun;

KVM: it's a Linux kernel module that offers hardware assisted virtualization. Due to its integration with the kernel and its limited complexity, it will be the de facto standard for virtualization with Linux in the next following years;

Lguest: it's a para-virtualizer for the Linux Kernel, made in less than 5000 lines of code [14];

Hyper-V: it's the virtualization technology made by Microsoft and made available for Windows Server 2008 and Windows Vista. It leverages on hardware support for virtualization.

2.5 Hardware virtualization

In this section, we discuss in details how a computer component could be virtualized, i.e. how it could be abstracted and presented to one or more virtual machines, preventing each one of them to access or interferes with others' data.

2.5.1 Processor

Processor virtualization is usually a simple topic. For the Popek and Goldberg principle stated above, the most portion of instructions are executed directly by the processor itself, for accuracy and performances. Only sensitive instructions require to be intercepted and somehow managed by the VMM. When this happens, there's a process analogous to a context switch: processor's current registers are saved, the handling routine is executed, and then saved registers are restored.

It's also required some level of protection for critical structures stored in memory, and this is usually done by leveraging on processors' access control mechanisms. In the Intel x86 architecture, each process could run in one of four privilege level, the less privileged numbered 3 and the most privileged numbered 0. In a no virtualized scenario, operating system runs at 0 level, and applications run at 3, leaving levels 1 and 2 unused. With a hypervisors like Xen or VMWare, the hypervisor and its operating system still running at level 0, meaning full access to memory and devices, and the virtualized machines run in an intermediate level. This is the main reason why it is difficult to virtualize an hypervisor.

2.5.2 Memory and DMA

In a modern architecture, each process has associated its own unique address space, and instructions and data are stored in a virtual address space. The virtual address space is implemented by the Memory Management Unit (MMU) that gets the virtual address and returns the physical address. This conversion is per process, meaning that two different processes will usually have the same virtual address mapped into two different physical addresses (although it's possible for two processes to share

memory; also, two threads of the same process will usually share memory).

This translation is made up by organizing the memory space of a process in a hierarchical structure, the page directory, the root of which is a part of the process context (on x86 architecture is a CPU register). A virtual address is composed of two parts, the directory part and the offset. The directory part will be combined with the page directory to determine the physical page, which is added to the offset to get the physical address [58].

This operation, called page tree walking, will require traversing the multi-level tree page table. Each Page Table Entry (PTE) has the same size of a page table, which is 4 KiB or 4 MiB on Intel x86 architecture (other architectures could have different page size coexisting in the system): as there are many of them, the PTEs are stored in memory. So, every time the directory part of the virtual address changes, it could be required to access some PTEs in memory, resulting in very poor performances. To avoid this, a MMU is equipped with a Translation Look-Aside Buffer (TLB), which is a specialized cache for virtual memory conversion lookups.

The problem associated with the TLB is that, as a cache, must from time to time to be invalidated. When a context

switch occurs, or when there is a transition between the kernel mode and the user mode (to adopt the Intel x86 nomenclature), the TLB will refer to a page table that is no longer the current page table, so it must be invalidated. As a result, the incoming memory accesses will require a page tree walking, until the TLB gets refilled.

A typical pattern on a modern system is when a process (running on user mode) requires an operation to the operating system by issuing a system call: the processor switches to kernel mode, the operating system will hopefully honor the requests, then the processor goes back to user mode and the process execution resumes. This flow has two transitions in it (the first from user mode to kernel mode, and the second from kernel mode to user mode), which in a naive TLB implementation would require two TLB invalidations. This is usually a waste of resources, because a better approach would be to selectively invalidate some of the TLB lines. If the kernel computation is small (as usually it is), the number of the referenced memory addresses is also small, so only some of the TLB lines must be invalidated.

This optimization requires that each TLB line is tagged, associating to it the page table which it refers to. Tagging is also useful for process switching (when processes are switched often) and when there's a thread switching, as in such a case no TLB

invalidation is needed.

Figure 2.3 (adapted from [57]) shows the general scheme for memory virtualization, stressing that the datapath required for converting a virtual memory address of a virtual machine to a physical address is almost doubled with respect to the no virtualized scenario: first, a virtual machine virtual memory address is translated into the physical machine guest address, and then the latter is translated into a physical machine real address.

Software memory virtualization

The VMWare hypervisor assumes that the hardware has not been enhanced for virtualization (although, when this is the case, it uses some of the available hardware features), so it works by deriving shadow structures from guest level primary structure.

Some of these structure could be mapped into the state of a virtual machine (i.e. processor state), some others as the page table directory will necessarily reside in memory. These structures are also privileged, so the VMM must protect them from unauthorized access, with the complication that modifications of these will usually not generate traps, and they could even be modified by an I/O operation, when the I/O device is memory

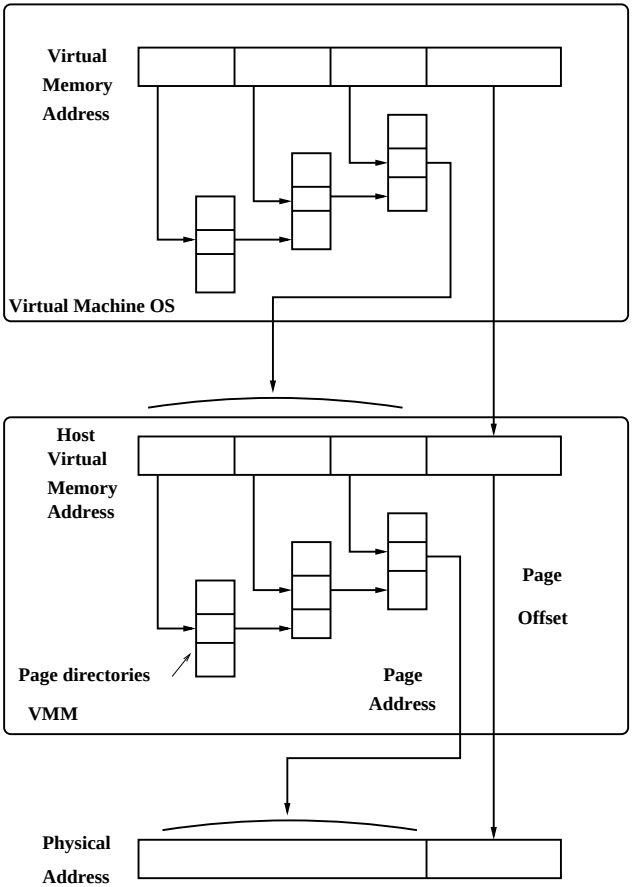

Figure 2.3: Memory virtualization datapath.

mapped.

VMWare use the hardware protection mechanism to protect and trace modification to the shadow structures [30]. If the PTEs are protected, every accesses to them will be trapped (the virtual machines are running de-privileged) and the control is transferred to the VMM. The VMM decodes the faulting instruction, emulates its effect on the primary structure, and then propagates the modification on the shadow structure. VMM must distinguish between true page faults, caused by the violation of the policy encoded by the guest PTEs (this happens when a virtualized process tries to access another virtualized process's memory space) and hidden page faults, caused by misses in the page table. True page faults are forwarded to the guest (that could faults and kills the offending process) whilst hidden page faults causes the VMM to construct an appropriate shadow PTE, and then resuming guest execution. The traces are used to keep in sync the shadow PTEs and the primary PTEs.

Hardware memory virtualization

In the para-virtualized approach, the virtual machine operating system is slightly modified, to made cooperation between it and the hypervisor simpler and more efficient. In the Xen hypervi-

sor, the privileged dom0 and the less privileged domU domains don't have unrestricted access to physical memory. The VMM creates its own page table for each domain, and the virtual machines construct their page table in a way that is similar for the para-virtualized and hardware virtualized case. Every time the virtual machine operating system modifies its page table, the VMM is invoked, and it will update its shadow page table.

This approach is quite expensive, for the TLB invalidation to take place and the creation and maintenance of a shadow page table structure.

Intel has defined the Extended Page Tables (EPTs) [9], and AMD the Nested Page Table (NPTs) for the Barcelona processor [100, 3]: both allow the virtual machine operating system to produce host virtual addresses from guest virtual addresses. The host virtual address is then translated into physical host address by using a per-virtual machine page tree, with a very little performance penalty, as this second step is done at processor speed without external memory accesses. At the time of this writing, these extension are not generally available, but benchmarks appear promising [68, 5, 35].

Also, the result of this complex address translation is stored into a TLB line. AMD has proposed a 1-bit tag extension with the Pacifica virtualization extension, called the Address Space

ID (ASID) [42]. This one bit tag could distinguish between VMM's address space and guests' address space, allowing the operating system to avoid flushing the entire TLB every time the VMM is entered or exited. Intel has Virtual Processor IDs (VPIDs) for the same purpose.

Even with hardware support, the entire memory addresses conversion process is quite complex, as it requires that two different memory schedulers (one for the virtual machine and one used by the VMM) must cooperate. As the memory scheduler is the most complex and tuned component of the operating system, this effort is daunting, and is for such reasons that the Linux KVM VMM [17] is gaining in popularity: There's only one scheduler, enhanced with virtualization oriented features that also leverage massively on hardware features, resulting in one single implementation to be maintained (if the running virtual machine is Linux) instead of two.

DMA Memory pinning

DMA capable devices usually side-step the CPU while transferring large amounts of data. To keep device's implementation simple, usually they don't have any idea about virtual memory, not to mention virtual machines. This will require that, during an I/O operation, the used memory region must be fixed

(pinned). This approach should be extended when an I/O operation is issued by a virtual machine, and the common approach is by the use of a locking mechanism. The VMM should manage locks to avoid conflicts and deadlocks between virtual machines.

To help virtualization of DMA function, Intel has developed the Intel Virtualization Directed I/O [10] and AMD has introduced the IOMMU unit [4].

2.5.3 Storage

In contrast to other peripherals, the virtualization of the storage is much more simpler. We stress out that in this paragraph with "storage virtualization" we define the reservation of specific portion of a system storage space (made up of local and remote disks, tapes and whatever) for the exclusive use of one or more virtual machine. The most common case is when a portion of storage space is reserved for use by a single virtual machine, analogously to the no virtualized scenario, although it's possibile that two or more virtual machines share a data storage area (as an example, a quorum disk).

All VMMs have some degree of flexibility in choosing how to carve out the area to be assigned to a virtual machine. It could be an image file, i.e. a single big file on the physical

system that the virtual machine accesses as an entire disk, with the VMM that maps the read and write requests of the virtual machine to read and write requests on the file. This approach offers a great degree of flexibility (all the storage of the virtual machines is contained in a file, which could be easily backed up and restored) and it's possible to define snapshots of the file, which are coherent point-in-time copies of the virtual machine storage, allowing for quickly restoring of the virtual machine's status. The main drawback of this approach is performance penalty: the datapath required for an I/O requests is doubled. On the other side, it's possible to assign an entire disk (or a partition on it) to a virtual machine, at the expense of some management and flexibility issues, gaining on performances as the datapath is reduced (the file system layer of the physical machine is skipped).

It has to be noted that storage availability is, nearly, the availability of the entire system, as it's the far most common cause of system outages. A careful planning of a virtual machine installation should try to balance between easy of management, backing up, migration and performances, avoiding unnecessary duplication of efforts, the most typical of it being a redundancy system like RAID doubled on the physical and virtual machine: it's usually sufficient that the virtual machine

ignores every problem related to redundancy, seeing only a simple storage system, where the VMM could better map it to a redundant data storage area.

On the same side, today's storage for server is usually remote, by using NAS or SAN infrastructures. All of them aren't virtualization-aware. As an example, a SAN server could be configured to selectively presents LUNs to a physical server, identified by a physical connection (zoning). If this LUN contains data storage for a virtual machine running into the physical server, a migration of this virtual machine will require to reconfigure the SAN server, as the LUN containing the virtual machine data should be, from now on, only accessed from the new server, whilst the old server must be disallowed to access the data, as the migration has been completed. This will require a coordination between the VMM and the SAN, and the SAN must trust the VMM, which in this scenario is usually deployed and distributed among different servers. On [76] it's presented the N_Port Identifier Virtualization extension for the Xen VMM to solve this. Others high availability solutions will program the SAN switch to selectively allow or forbid data access as the virtual machines are being moved over the infrastructure.

2.5.4 Network

Network virtualization refers to the ability to offer to each virtual machine a NIC interface, allowing it to send and receive network traffic without interference, snooping or service degradation caused by the other virtual machines.

Network interface is complex as the network traffic is unsolicited, requiring the VMM to be prepared to receive and respond to traffic that could be received at any time.

Private devices

The first approach, adopted by the IBM S/360, consisted on assigning a physical network interface to each virtual machine (this was also made for other devices such terminals, disks and so on). Transfers to and from the network card were made by channel programs, doing programmed I/O to transfer data from and to the memory.

Modern virtualization systems also allow for the private device I/O, as Xen does with the pciback module. This approach as a relative degree of flexibility, as it requires that the VMM must boot with a configuration that prevents some PCI devices (identified by their slot and PCI number) to be configured by the dom0 kernel, and then it's possible to configure a virtual

machine to directly interact with the PCI device by configuring its description file [27]. Although it's possible to reassign a PCI device to another virtual machine, the set of directly accessed devices could be changed only by a VMM reboot.

The same approach is also used by the IBM Logical Partitioning (LPAR) architecture for the Power4 processor, relying on specific processor features.

More recent approaches as the LPAR for the Power4 processor allow for isolated access at the PCI-level, leveraging on a IOMMU unit that creates a I/O page table for each device, with memory mappings from the pages owned by the virtual machine to the assigned device. As a result, for each DMA operation the processor consults the IOMMU, disabling I/O access to devices not owned by the virtual machine.

The private device approach has a clear advantage, performances maximization, but at the expense of a possible underutilization (or over-provisioning) of physical resources. Also, the DMA memory pinning problem discussed in section 2.5.2 could also severely restrict the feasibility of this approach for a given network device.

Shared devices

In the Xen architecture, the shared access to the network is made by using a virtualized spool-file interface, called an I/O domain. The VMM interprets readings on this buffer as receiving a network packet, and writing to it as sending a network packet. As the figure 2.4 shows, the Xen VMM could be decomposed in two elements, the hypervisor and the driver domain (the Xen architecture is the common approach for shared devices virtualization). The hypervisor is the abstraction layer between the virtual machine and the real hardware, and each I/O device is managed by a I/O domain, which runs a Linux kernel. Each virtual machine could communicate with a device by using a front-end driver, which then connects to a back-end driver.

As an example, when a packet is transmitted from a virtual machine, it's copied (or remapped) from the front-end driver to the back-end driver, and then queued for transmission from the NIC. An interrupt is generated when a packet is received, triggering the copy (or remap) of the packet from the back-end driver to the specific front-end driver. The back-end driver is capable of dispatching the network packet to the specific virtual machine because it inspects the packet, sees the MAC ad-

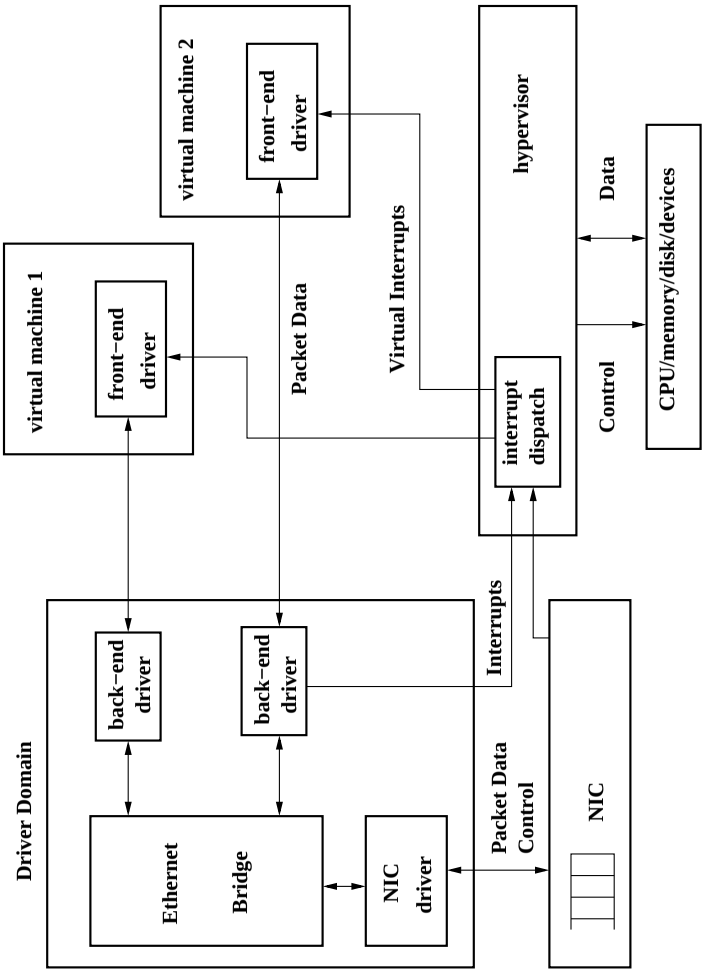

Figure 2.4: Xen architecture for shared network devices.

dress and then routes it accordingly to the destination front-end driver. After the copy of the network data, a virtual interrupt is sent to the virtual machine, which will in turn wake up the front-end driver and process the packet.

Data protection and isolation between the different virtual machine is ensured by the driver domain. This approach results in some overhead, as it's possible that data must be copied from and to memory, and the number of interrupts required to process a network packet is doubled. A specialized driver could result in a substantial increase in performance [29], by doing memory mapping and not memory copying and avoiding to check for transmission errors, as this control is also made by the back-end driver. Another problem is that the I/O domain must be scheduled to allow for packet processing, and the only way to avoid it is to move the back-end driver code into the hypervisor, resulting in a bigger hypervisor, more exposed to flaws and security related problems.

Concurrent Direct Network Access

A modern NIC interface is usually organized with more than one queue for packet transmission and reception. This is done because, to increase availability, it's usually better to bond together two or more network cards, presenting them as a unique

network device, and then configuring them with two or more IP addresses: if a network card fails, the others will continue to work, with a minimal service disruption (also, on multi-core machines, this prevents for global locking on network resources, as it's possible to assign a queue to one specific core).

This hardware feature is employed in the Concurrent Direct Network Access (CDNA), where each one of this queues could be assigned to a specific virtual machines, as the figure 2.5 shows.

The hypervisor treats each queue as if it were a physical network card, assigning ownership of it to a virtual machine, without the need to define an I/O domain, resulting in near zero overhead: interrupts are routed directly to the virtual machine owner of the queue, and the virtual machine reads and writes directly on the queue.

Memory protection is a bit more complex, as there's no more a driver domain that could validate memory access to the device. The problem is exacerbated in the Intel x86 architecture, where I/O devices have only physical addresses. On this architecture, the hypervisor must validate each buffer, ensuring that every virtual machine does not add to or remove from the queue packets belonging to a queue it does not own, and preventing the queue ownership from changing, considering this a privileged operation. It should be noted that these two tasks are the same

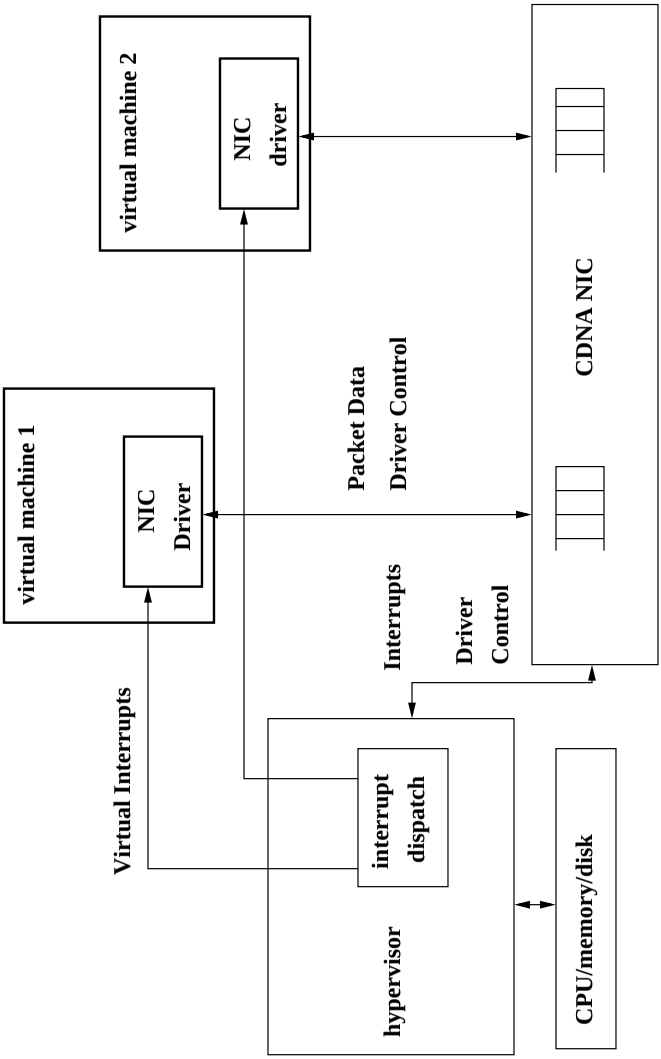

Figure 2.5: CDNA architecture.

made by a MMU with respect to memory access, so the general availability of a IOMMU will eliminate these burdens from the hypervisor. The performances of CNDA are such that the transmission throughput is linear with the increasing number of virtual machines, while the shared device approach a la Xen decreases exponentially. The performance gap for receiving is reduced, as Xen works better when demultiplexes received packets [91]. Intel has developed the Virtual Machine Device Queue to effectively implement the CDNA network virtualization [11].

2.6 Concluding remarks

Virtualization dates back in computer history, and comes in many different forms. Providing and leveraging hardware features to get the best from this approach to computation is a complex process, as there are many inter-dependencies and many different approaches, that must be evaluated against requirements and provided features.

3

Virtualization architectures

The different hypervisors that have been presented throughout Chapter 2 are merely techniques that could be used when virtualization has to be put in place in a system. In this chapter we analyze the next logical step, where virtualization is an architectural asset that brings value to a distributed system, and not only an available feature. It has to be noted that hypervisors' makers put a lot of emphasis in the server consolidation

scenario, where some (and possibly many) legacy systems are virtualized. This is a cost-savvy strategy, but virtualization has a lot more to offer when it's an integral part of a distributed system.

A distributed system is usually designed and built up (often with a trial and error approach) with a list of desired features, both measurable and not-measurable, that drive the design process. Some of the features that could take a great benefit from virtualization are:

- efficiency: the average server usually works at 10-15% of its capacity, with some temporary surges. By packing some (virtual) servers into a physical one, it's possible to cut down electricity and maintenance costs;
- availability: by leveraging on live migration, it's possible to migrate a running virtual machine from one physical host to another, in a proactive way (allowing for ordinary maintenance) or reactive (as in a disaster recovery scenario). Recovery Oriented Computing [21] is a remarkable approach to reach this goal;
- ease of deployment: by cloning a virtual machine, it's possible to install it on several different physical hosts;

- load distribution: virtualized machines could be the building block of a cluster that spans over different underlying physical machines, hiding the heterogeneous underlying hardware.

Virtualization could be effective in achieving these sometimes conflicting goals, as it defines a central management function that is implemented by another intermediate layer. Having another layer means also adding complexity, as complex interactions with the rest of the stack result in. As virtualization becomes pervasive, the fighting arena for manufacturers will progressively shift from performances to management tools: new generation of computer processors, operating systems and peripherals are designed with virtualization in mind, and the burden of hypervisors will move from making a computing platform virtualization friendly (i.e. virtualize a legacy server) to managing hundreds and thousands of different virtualized systems, that offers different services to different users.

This will be a common scenario for a large data center which today's offer include shared hosts for low-traffic sites and dedicated hosting, but both of them are far less than ideal. Shared hosting is acceptable only when both traffic and requested level of security are low, whilst dedicated hosting requires careful planning (it will take some time to change the footprint of an

installation, especially when the customer wants to downgrade) and it will usually result in a waste of money, from customer's point of view, as the average server is usually under-utilized. For this very reason a modern data center should use virtualization in its core architecture, to rapidly adapt to its customers' needs. One of the pioneers in this approach is Amazon, with the Amazon Elastic Computing Cloud (EC2) service [1].

With Amazon EC2, a customer could lease a virtual server, and pay only for the time the server is up and running. When the server is offline, the server's image is stored offline (a collateral service of Amazon, the Amazon Simple Storage Service (S3) [2] could take care of that). Amazon has a large pool of machines, so it could instantiate even thousands of server for a customer within minutes from the request. These machines could be used for the time required to perform their job, as doing a number crunching computation (that was the case when New York Times need to reprocess and convert its entire archive in electronic format [19]), acting as a backup system or giving some extra capacity power to offload some computations for a site that is experiencing a surge in traffic. A start-up company could lease its servers and expanding its pool when it's needed, concentrating its effort on the products and developing a long-term strategy for its information system in the meantime. When

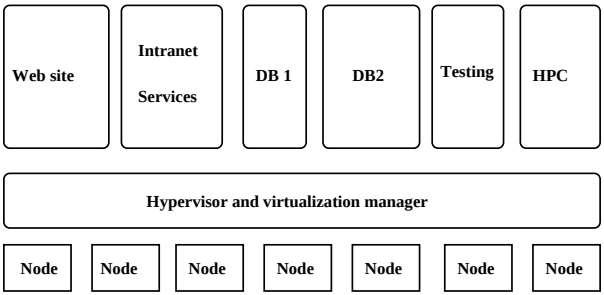

Figure 3.1: A virtualization architecture.

all cost factors are taken into account, this approach is usually cheaper in the short-term than the traditional one.

This unifying approach could be worth to be used even when the customer and the provider (of computing machines) are the same organization, i.e. by the IT department of a corporate. Instead of having many different clusters, each one devoted to a specific business function, there could be only one cluster, with some isles on top of it, each one for a high level task. These isles could be expanded or reduced with respect to their size (associated resources) according to the evolution of the business. Figure 3.1 shows an example where different area sizes of the high level functions remarks the different amount of associated resources.

At the price of a more complex setup and planning, this architecture will bring benefits to the organization that uses it. Each computer is a computing block, that is globally managed and assigned. As long as the hypervisor remains the same, it's possible to mix and match different hardware solutions, maximizing the efficiency of the infrastructure and obtaining, with less effort, high profile features like high availability, disaster recovery, rapid deployment and so on.

As a result, virtualization must be included in the design process of a distributed system. In this chapter, we see some standard techniques to develop a modern distributed system and how virtualization could be integrated in it since the design process. We consider this in the more general context of autonomic computing, that will provide a useful framework.

3.1 Reference architecture

The reference architecture we consider is a multi-tier distributed system, shown in figure 3.2.

Each tier is functionally distinct from the others. Each tier is made up of nodes of the same type and with the same associated resources (CPU, disks, memory, ...). In this architecture, incoming requests are faced by the top tier, N, which, to serve

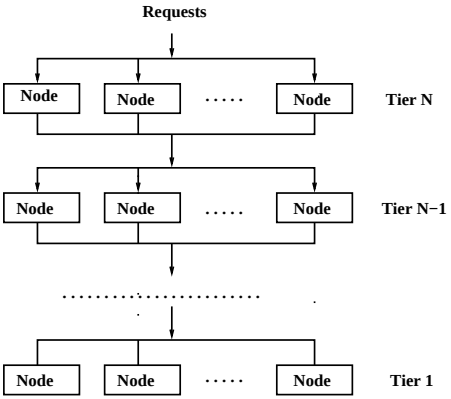

Figure 3.2: A multi-tier distributed system.

them, requires services from the next tier, $N-1$, which in turn relays to tier $N-2$ and so on. It is assumed that a request flows only from one tier to the next one to be served, although it could be possible that it doesn't need to traverse all the tiers. After reaching the last tier, the computed result flows upwards, is aggregated by the different traversed tiers, and is finally sent to the client. Requests are grouped in classes, distinguished by the amount of requests and type of resources they require by each tier in order to be served, and within the same class they are assumed statistically indistinguishable.

In the context of web services, distributed systems are usually made up of three tiers:

- front end tier, which is the connection point for external clients accessing the web service, with its nodes serving only static content. This tier usually does not suffer from scalability problems as a modern web server could serve static content in such a fast way to saturate the available Internet connection;

- application tier, where the application logic resides. This is made up of programs running in the context of an HTTP request, written in languages like PHP, Java and so on. Each customer interacting with the web service usually produces more than one service request, and these are all interdependent (as an example, a buy order follows a search catalog function), so requests are grouped into sessions. To avoid replicating session stateful information along all nodes constituting the application tier, the load distribution function must be session aware, and this introduces some limits in the scalability of this tier;

- database tier, which handles all queries to the database system. This tier is usually the most difficult to expand, as all low-level DBMS employ a shared-nothing approach,

which makes difficult to realize concurrent, write-access to the data.

More tiers could be added to this model, as an example a front-end tier could manage web authorization and access, and an inner tier could model and interact with legacy mainframe systems. In each case, although this architecture is the de facto standard for web services, scalability of it must carefully addressed: adding some extra nodes to a specific tier wouldn't necessarily let to a general performances improvement as some common capacity planning problems could arise:

- the extra nodes are not added to the tier that is the bottleneck of the distributed system;
- the extra nodes are added to a tier that has almost no need of them;
- the speed-up of the tier being increased is such that there's a waste of processing power.

The last point is critical. Each tier has associated a load distribution function, that dispatches the incoming requests from the upper tier to a specific node. This dispatching first requires some kind of strategy (as an example, in order to dispatch requests to less busy nodes it's required to monitor and collect

the load on each node) and it could be more or less efficient for the specific semantic of operations performed by the tier. As a result, adding nodes to a tier that is already experiencing the saturation of its distribution function will result in no effect. In some cases it will be required to change the strategy used to distribute the load between the nodes of the same tier, that it could in turn require to change the implementation of the software running on the nodes.

3.1.1 Modeling of multi-tier systems

Modeling of multi tier systems has attracted a lot of interest in the last several years, as pervasive web services are usually best implemented in this framework.

As for each performance model, there's a trade off between its accuracy and the feasibility of the implementation: a more precise model, that requires too much instrumentation of the real system to feed its model solver, or that has a complex model that requires a lot of processing time to be solved, is of no more than theoretical interest, as the predictions originating from it cannot be applied in an on-line system, so the level of detail in the models is chosen in a utilitarian fashion.

In [61] the focus is in the modeling of a single server, com-

prised of physical resources like CPU, memory and disks, that could interact with other servers to accomplish the requested functions.

In [98] the developed model is session-based and take into account caching effects and concurrency limit on the tiers. In this model, each tier is modeled via a queue, and the system is solved via the Mean Value Analysis algorithm ([90]). This model allows for performances prediction and dynamic capacity provisioning, and it could handle multi-class models.

In [105] the most important input for the MVA model, the average service time for the CPU at each tier, is estimated via a regression technique, minimizing the quadratic difference from observed and estimated utilization rate.

In [99] the focus is in provisioning the application tier. First, it is shown that for limited timescale a tier could be modeled as a M/G/1/PS queue, i.e. the arrival rate of requests at the application tier is described by a Poisson distribution (this of course is not true on large and different timescales, as self-similarity appears due to an high degree of correlation between arrivals. The timescales over which correlations exists are delimited by an upper bound, called Critical Time Scale [93]). Then the correct solution of the model, that requires some complex calculations making it infeasible for on-line provisioning, is compared

against three different approximations. All of these approximation methods achieve allocations with costs near the minimum possible, while simpler heuristics incur in significantly higher costs. This model could also be used to determine the optimal number of servers to bve deployed for the application tier, ignoring the provisioning of the other tiers.

The assumption that all nodes are work conserving and that the discipline is processor sharing (PS) is generally applicable, while it could not be the case that the arrivals are modeled via a Poisson distribution. To have the most general multi-tier model, we assume that each server is modeled via a G/G/1 queue. The behavior of this server is described via the following equation from queuing theory [72]:

$$\lambda_i \geq \left[s_i + \frac{\sigma_a^2 + \sigma_b^2}{2 * (d_i - s_i)} \right]^{-1} \tag{3.1}$$

where:

- λ_i is the arrival rate for tier i;
- s_i is the average service time for requests on tier i;
- d_i is the mean response time for requests on tier i;
- σ_a^2 is the variance of inter-arrival time for requests on tier i;

- $\sigma_b^2$is the variance of service times for requests on tier i.

Quantities as d_i and s_i and their variances are known or could be on-line estimated (instrumenting the distributed system to record, for each transaction, request time and completion time), so a lower bound on λ_i could be evaluated for each server. If each session has a think time of Z, by Little's Law the session arrival rate of λ could be translated in a request arrival rate of $\frac{\lambda\tau}{Z}$, where τ is the average session duration. So, when the capacity λ_i of a single server is computed, the number n_i of server required for tier i is simply defined as:

$$n_i = \left\lceil \frac{\beta_i \lambda \tau}{\lambda_i Z} \right\rceil \tag{3.2}$$

the β_i is a correction factor specific for each tier, that take into the formula competing effects as caching, load distribution and speed-up. If the speed-up behavior is not constant but a function of n, then n_i is defined as the minimum value of n_i for which it holds:

$$n_i \lambda_i Z \geq \beta_i(n_i)\lambda\tau \tag{3.3}$$

with the constraint that each tier has at least one node:

$$n_i \geq 1, \forall i \tag{3.4}$$

Equation 3.2 is general, so more tier-specific equations could lead to substantial improvements determining a lower number of server n_i, at the cost of more specific tier-knowledge. As our objective is to investigate in the allocation of this servers over physical machines, we assume that an equation like 3.2 will be sufficient for our needs.

3.2 Virtualization performances and measurement

Virtualization is an additional layer, so it has some complexity by itself (especially when is a foundation for the entire infrastructure) and an associated overhead. Recalling the definition of VMM made by Popek and Goldberg, this overhead must be negligible, but it must be evaluated.

Virtualization is making some fast progresses, and as a result the performances of an hypervisor are changing rapidly, making of no interest to report them in a dissertation where the focus is on the architectural component. Plus, very few comparative studies are conducted on this field, as some hypervisors' licenses

substantially prohibits to publish them.

In [83] the relative performances of Xen and OpenVZ are compared on some scenarios, founding that Xen has a lot of overhead, mainly due to level two cache misses, up to ten times than the ones in OpenVZ. It's generally understood that this gap will reduce as the hardware features are made available, as discussed throughout Chapter 1.

From discussion in Chapter 1 it's clear that performances are strictly dependent on the number of privileged instructions that the hypervisor has to emulate, because the innocuous instruction are executed directly by the hardware, with no performances penalty.

On [81] this formula is adopted to evaluate the slowdown of a virtualization:

$$S_v = f_p * N_e + (1 - f_p) = f_p * (N_e - 1) + 1 \qquad (3.5)$$

where:

- f_p is the fraction of privileged instruction;
- N_e is the number of instruction required to emulate a privileged instruction.

As an example, when $f = 0.1\%$ and $N = 500$ the slowdown is

$S = 1.5$, meaning that an application running on top of a virtual machine will see its execution time increased by 50%. The main advantage of the equation 3.5 lies in its simplicity. The fixed value of N_e does not take into account adaptive behavior or caching mechanism employed by the translator, and the fraction f_p of privileged instructions could be substantially reduced with some modification to the virtual machine's code base.

But in even more general terms, the pace of progresses in this field results in high performances variations, even between two releases of the same hypervisors, as soon as it leverages on some hardware features or modification of the virtualized code.

Nevertheless, it makes sense to measure the virtualization overhead, over a specific scenario and this can be done as seen in [81].

In that paper, some metrics have been collected for a physical system that comprises of two virtual machines, the first running a batch system and a Transaction Process Manager (TPM), and the second devoted to testing purposes. Table 3.1 is an excerpt that we use to show the relevant results.

To determine the CPU utilization factor for the first virtual machine, we use this formula:

Metric	Value
$T^{vmm}_{cpu,vm1}$	420 s
$T^{vmm}_{cpu,vm2}$	220 s
U^{vmm}_{cpu}	0.40

Table 3.1: Measurements for estimation of VMM overhead.

$$U^{vm1}_{cpu} = U^{vmm}_{cpu} * \frac{T^{vmm}_{cpu,vm1}}{T^{vmm}_{cpu,vm1} + T^{vmm}_{cpu,vm2}} = 0.2625 \tag{3.6}$$

Eq. 3.6 states that the utilization of the physical CPU for the first virtual machine is a fraction of the utilization of the CPU as seen by the hypervisor, U^{vmm}_{cpu}. The apportionment factor is the ratio between the total time that CPU is running the first virtual machine as measured by the VMM, $T^{vmm}_{cpu,vm1}$ and the total time that CPU is running the virtual machines, as seen by the VMM. The experiment spans over a 30 minutes interval, and if we calculate the utilization of the physical CPU for the first virtual machine as:

$$U^{vm1}_{cpu} = \frac{T^{vmm}_{cpu,vm1}}{30 * 60} = 0.2333 \tag{3.7}$$

we get a result that does not take care of the 2.92% difference

due to overhead imposed by the VMM.

This little overhead is going to be reduced as the implementation of the hypervisors are better. It has to be noted that this overhead is in fact a result of two competing phenomenons. The first is the overhead that result when privileged instructions are encountered and have to be somehow emulated by the VMM, the second are the optimization that the VMM could put in place to reduce penalties due, for example, to memory faults.

Eq. 3.6 is made with the assumption that there isn't interference between the two virtual machines competing over the same result, i.e. the flow of operation for the two machines are almost the same as if they were executed without an interposing VMM. This in turn requires that the VMM is perfectly capable of isolating the two virtual machines regarding to resources contention.

We stress that a critical point is that the VMM must take every care in avoiding the interference problem. As an example, if a virtual machine is doing a lot of unexpected I/O work with the disk (i.e. is running out of memory and it's moving pages to and from disk) this behavior must not limit the I/O disk bandwidth available for others virtual machines, requiring a global monitoring of resources consumption. This could generate a butterfly effect, as the virtualized operating system could

choose an aggressive I/O strategy that is greedy in the short term but it would have been resulted in a general performances improvement in the long term (usually the virtualized operating system has no idea that is running in a virtual machine). Limiting the ability of the virtualized operating system could be no performance wise in the long term, so the VMM should find a balance between the fairness of the resources allocation and the resulting global throughput.

3.3 Autonomic computing

Today's computer systems have an intrinsic daunting complexity that stems from the wide range of technologies and complexity of interactions. As a result, identifying problems in a production system could be quite challenging, and optimizing and tuning for performances is often out of the question. This will end up in very little efficiency, reduced availability and security problems.

In 2001, IBM has proposed autonomic computing as a long-term answer to these problems [67]. Autonomic systems could manage themselves, as the autonomic nervous systems governs human body adapting it to changing environments and repairing it, a damage could occurred, with little or now knowledge from

the high level function as the conscience.

Distinguishing properties of autonomic systems are [70]:

Self-optimization: an autonomic system continually seek ways to improve its operation, identifying options and actions that make it more efficient on performances or costs, according to some built-in metric;

Self-healing: an autonomic system identifies defective components and put them off-line, re-organizing itself to continue to work with the remaining parts;

Self-protection: an autonomic system acknowledges attacks from the outside, preventing them to have success and compromise the entire system.

These properties are usually collective identified as self-* properties. All of them are, to say the least, appealing, as they solve a great share of the problems that everyday happen and arise in a production system. Autonomic computing is still in its infantry, as many problems have yet to be addressed [56].

All the self-* properties could be implemented leveraging on virtualization. If we imagine an autonomic computing system as a system made up of independent but cooperating systems, each of them could be implemented as a virtual machine. Self-protection could take advantages by the isolation property that

an hypervisor offers, and self-healing could be more easily obtained if the failure of a component will be confined in its boundaries, allowing for shutting down the component and eventually re-initialize it (hoping the failure is transient) or migrating it over different physical resources (if the failure is due to hardware's flaws), giving some degree of flexibility over a traditional approach when the coupling between resources and components is tighter.

In this dissertation we mainly investigate how to deal with self-optimization, i.e. how to leverage on virtualization to allow an autonomic computing system to adapt itself to different workloads.

3.3.1 Self-optimization

Self-optimization is the ability of a system to adapt itself on different conditions, and as a distributed system is intended to give services to clients requesting them, the focus is on systems that adapt themselves on variable workloads. We consider the architecture proposed in [80] as a general framework we wish to extend.

The proposed architecture evolves around a QoS Controller, depicted in figure 3.3 which has four main components:

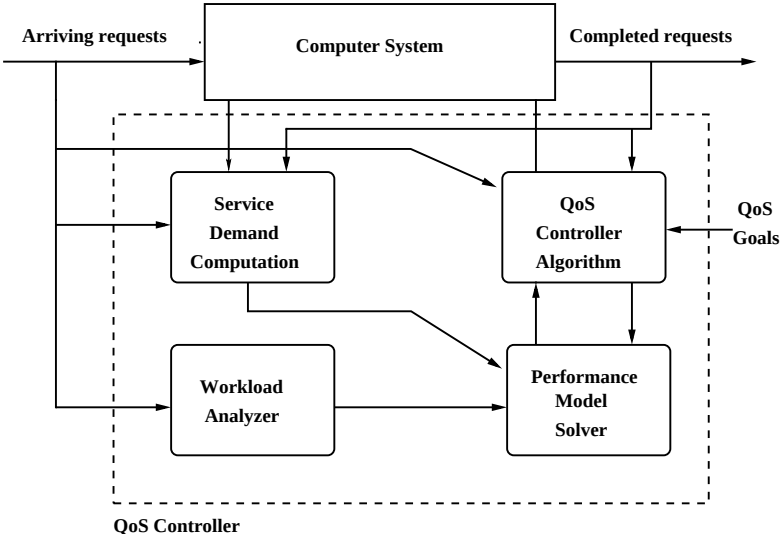

Figure 3.3: The QoS Controller.

Service Demand Computation: collects utilization data on all system resources and count of completed requests. The service demand of a request, defined as the total average service time of a request for a specific resource, can then be computed as the ratio between the resource utilization and the system throughput [78]. These service demands are used as input parameters for a Queuing Model solved by the Performance Model Solver;

Workload Analyzer: analyzes the arriving requests and computes statistics as average arrival rate. It could also use statistical techniques to forecast workload. These statistics are made on a per-interval basis, called controller intervals;

Performance Model Solver: receives requests from the QoS Controller Algorithm to solve the Queuing Model for a specific configuration of the system. Its inputs are the configuration parameter values, service demand values and workload intensity values. Its output is the QoS value for the configuration used as input, given according to some metric;

QoS Controller Algorithm: it runs the controller algorithm at the beginning of each controller interval. Its input

are the QoS goals, departure and arrival processes, and it searches a close-to-optimal solution, by a mix of analytical models and combinatorial searching techniques. Each possible solution is evaluated by the Performance Model Solver. After the best possible configuration has been discovered, it sends commands to reconfigure the system.

This architecture has been then validated for highly variable workloads [39]. In [40], this architecture has been expanded to allow the co-existence of different Application Environments (AEs) on the same physical machine. On each physical machine there's a QoS Controller analogous to the one of figure 3.3 plus a centralized global controller.

These architectures fit in the general model for autonomic computing proposed in [103].

In [101], a similar architecture has been considered for appliance based autonomic provisioning. The architecture defines some Virtual Application Environments (VAEs): a VAE spans over one or more virtual servers, and each server is defined inside a physical machine. Each VAE has a On-Demand Router that dispatches incoming request to the less loaded virtual server inside the VAE, in a round-robin fashion. A global, utility-driven, virtualization-aware model solver solves a performance model to determine the better configuration for the VAEs, for the given

(and forecasted) workload. This article is remarkable as it's the first to take into account the time required for virtual machine provisioning, i.e. the time required to activate a virtual machine and the time required for closing it, and offers a formula to evaluate the performance overhead of an hypervisor, as this datum is required by the model solver to avoid overloading a physical machine. Based on the works in [87, 88] it is assumed that the capacity available for a specific virtual machine is a fair share of the total (raw) capacity of the physical machine, attenuated by a constant factor α that takes care of overhead due to virtualization:

$$C_v = (1 - \alpha) * C_p/N \tag{3.8}$$

where:

- C_v is the capacity of the virtual machines;
- C_p is the capacity of the physical machine;
- N is the number of virtual machines instantiated over the same physical machine.

In the aforementioned article, $\alpha = 0.1$. (in [31] authors choose an overhead between 1.5 and 3, i.e. $\frac{1}{3} \leq \alpha \leq \frac{2}{3}$. This different

values are best explained by different virtualization technologies). The capacity is defined as a global index of the relative performances of a physical server, i.e. the model is monodimensional. The mapping between the physical server and the virtual machine is described by an association matrix, and the model solver returns, for a given matrix and a (forecasted) workload, a new association matrix. The time required to deploy the new association matrix is considered as a linear sum of the time required to shut down the no longer necessary virtual machines and the time required to boot the new virtual ones.

In [31] the model focuses on SLA violations, trying to minimize it. To get a solvable performance model, the probability of a service time bigger than the agreed value is bounded via the Markov Inequality [71].

Other approaches for self-optimization are possible. In [84] it's exposed a control of CPU shares of two competing virtual machines over the same physical node based on control theory.

3.3.2 Proposed extension to the model

All the previous architectures have some boundaries we wish to extend.

First, they assume that the capacity of the servers are fixed,

and so the service demand times are only input variables. This is in general not true when virtualization deploys all of its power in resource sharing: it's possible to dynamically vary the CPU share assigned to a virtual machine.

But the most important limit is that all of these models assume that the only critical parameter for modeling is the CPU power of the virtual machines ([79], [89]). A notable exception is [69] where are both considered a load dependent resource, as the CPU, and load independent resource as the main memory; but performances of a (virtual) machines stem from all the available resources. As an example, front end tier require a lot of bandwidth for connection to external clients, and a database tier is bounded by storage bandwidth. As is shown in [61], all of these (and possibly others) parameters are required to have a correct estimation of service demand times.

Consequently, if each virtual machine is described by a resources demand vector, each physical machine must by described by a dimensionally analogous resources vector, and it could be possible that a physical machine has some spare resources that are insufficient for instantiate a specific virtual machine in it.

As an example, imagine we have a physical machine with 8 GiB of RAM and four processors (for the sake of brevity we consider only two elements for the resource vector). This ma-

chine could host 4 virtual machines, each one requiring 1.8 GiB of RAM and one processor, but it cannot accommodate more than 3 virtual machines requiring 3 GiB of RAM and one processor. In the latter case, 2 GiB of RAM and one processor are free and unused, and they maybe accommodate another virtual machine, belonging to a different tier, with a more compatible resources demand vector.

Last, in the current works it is often assumed that all the applications are available on each physical machines, ready to be activated should the workload variations require it ([69]). For proprietary applications this is usually not acceptable, as the license fees are for each installed copy and not only for running copies.

For a virtualized architecture, migration times must be taken into account. In [89], it is assumed that the controller interval are a lot bigger than the time required for server migration, and this is a standard modeling option, as the controller time is in the range of 5-30 minutes.

So, in this dissertation we concentrate on an allocation problem. We assume that a multi-tier performance model solver has determined the number of nodes that must be in each tier for the current (or forecasted) workload. Each one of these nodes is described via a resource demand vector, and there are avail-

able some physical machines described by a resource vector. We want to map the former into the latter, i.e. assign each virtual machine to only one physical machine, without exceeding available resources and possibly minimizing the number of required physical machine, to achieve maximum efficiency.

In Chapter 4 we formalize our model, discuss its computational complexity and propose some algorithms to tackle it.

4

The mapping problem

As seen throughout previous chapters, virtualization could play a fundamental role in defining a distributed architecture that could self-adapt to workload variations. In Chapter 3 we have surveyed studies that deal with the problem of defining the numerousness of each of the tier comprising a multi-tier distributed system, especially in the context of web services. To the best of our knowledge, there are no studies that model these systems for more than one (or two) resources as CPU power (and available memory) and no one that deal with the mapping problem that we define as: given a set of virtual machines, each one de-

scribed by a resource demand vector, and a group of physical machines, each one described by an available resources vector, which is the best mapping of the former to the latter, i.e. how to associate each virtual machine to one and only one physical machine, without exceeding available physical resources and maximizing a given metric?

Current studies ignore completely this problem, and ofter characterize performances of a virtual machine only by its CPU power. Instead, we choose to work, for this mapping problem, in a multi-dimensional space, where we have both quantitative and qualitative characteristics of the physical (and therefore virtual) machines.

4.1 Problem formalization

We formalize the mapping problem to allow for maximum generalization.

To do so, we assume that the virtual machines are grouped together, and that for each group we must allocate one and one only machine to a physical one. Machines in the same group represent different service levels and are characterized by different resource demand vectors, and for each virtual machine there's an associated profit that is earned when the machine is

chosen to be instantiate.

We want to maximize the grand total of profits, while minimizing the number of physical machine we have to use. It's possible that, for some or even all groups, we have only one virtual machine for each group, meaning that we cannot do anything but instantiate that machine, and in such a case the problem is only to find where to instantiate it. As the virtual machines are pooled in groups, we indicate each one of them by two indexes, the first denoting the group and the second the machine in the group (i.e. the service level). Following this convention, if X is a generic scalar (or vector), X^{ij} is the scalar (or vector) pertaining to the $j-$th machine of the $i-$th group.

We stipulate that:

- G is the number of groups. Each group is composed of g_i different machines (it's possible that $g_i = 1$);
- each virtual machine is described by a $K-$dimensional demand vector $D^{ij} = (d_1^{ij}, d_2^{ij}, ...d_k^{ij})$;
- each virtual machine has an associated profit P^{ij};
- M is the number of physical machines;
- each physical machine is described by a $K-$dimensional resource vector, $R^l = (r_1^l, r_2^l, ..., r_k^l)$, $1 \leq l \leq M$;

- for each i, we have $D^{i1} \leq D^{i2} \leq ... \leq D^{ig_i}$, coordinate wise, and $P^{i1} \leq P^{i2} \leq ... \leq P^{ig_i}$.

The decision variables x_m^{ij} are defined as:

$$x_m^{ij} = \begin{cases} 0 & \text{machine } ij \text{ is not on physical machine } m \\ 1 & \text{machine } ij \text{ is on physical machine } m \end{cases} \tag{4.1}$$

We want to choose one and one only virtual machine from each group, and allocate it on a physical machine, with the constraint that we cannot exceed the available resources, maximizing the total profit earned and minimizing the number of physical machines that are used.

To do so, we define the variables u^l as:

$$u_m = \begin{cases} 0 & \text{if physical machine m is not used} \\ 1 & \text{if physical machine m is used} \end{cases} \tag{4.2}$$

Our objective function is:

$$P = \sum_{i=1}^{G} \sum_{j=1}^{g_i} \sum_{m=1}^{M} x_m^{ij} P^{ij} - C * \sum_{m=0}^{M} u_m \tag{4.3}$$

which is the total profit of the virtual machines that are instantiated minus the number of physical machines used times a convenient constant C. We assume C as a constant as the operational costs for running the infrastructure (e.g. electricity costs, maintenance fees, co-location expenses) are usually proportional to the number of machines comprising the infrastructure: as we want to maximize their usage (by allowing for different service levels) we also want not to use more than the strictly necessary.

We extend the $\leq$ operator from scalar to vectors in a coordinate wise fashion: if $X = (x_1, x_2, ..., x_n), Y = (y_1, y_2, ..., y_n)$ we say that $X \leq Y$ iff $x_i < y_i$ for each i s.t. $1 \leq i \leq n$, so constraints are formally defined as:

$$\forall i \,, \sum_{m=1}^{M} \sum_{j=1}^{g_i} x_m^{ij} = 1 \tag{4.4}$$

$$\forall m \,, \sum_{i} \sum_{j} x_m^{ij} D^{ij} \leq R^m \tag{4.5}$$

$$\forall m, i, j \,, \; u_m \geq x_m^{ij} \tag{4.6}$$

Eq. 4.4 means that we choose only one virtual machine for each group, and eq. 4.5 means that, on each physical machine, we cannot allocate more resources than available ones, while eq.

4.6 relates variables u^l to x_m^{ij} according to def. 4.2 .

4.1.1 Discussion of formalization

The key issue of the proposed formalization is that we assume a fixed number of available physical machines, each one with predefined associated resources. We recall that the mapping problem arise when we have already solved a multi-tier performance model, which in turn requires to have, besides others, parameters as the service average time that is determined, analytically or by live system instrumentation, only after each tier has been characterized by its computing power. Therefore, the only potentially limiting factor for the proposed formalization is that the number of available physical machines M is fixed, and it's possibile that we would experience over-provisioning (we could be able to solve the same mapping problem with lesser physical machine) or, on the contrary, that we have too few machines to solve it.

Determining the minimum value of M that leads to a solution satisfying constraint could be quite challenging. We observe some basic facts.

First, we can define easily necessary but not sufficient conditions that would help in defining an acceptable value for M.

For each group of virtual machines, we consider that the resource vectors are non decreasing ordered, i.e. that we have $D^{i1} \leq D^{i2} \leq D^{ig_i}$, for each i. This means that, for each group i, we could consider D^{i1} as the minimum level of resources that must be instantiated by the pool of physical machines. Now, if the sum of these D^{i1} exceeds, even for only one resource, the sum of available resources provided by the physical machines ($\sum R^i$), a solution could not exist.

As an example, if we need to allocate four virtual machines, each one requiring 4 GiB of RAM, and we have 2 physical hosts of 5 GiB of RAM each, we have less memory than needed, and a mapping cannot be determined.

This is a necessary but not sufficient condition: if, instead of 2 physical hosts of 5 GiB, we have had 10 physical machines with 3 GiB of RAM each, the grand total of available memory would be of 30 GiB, but nevertheless a mapping couldn't be found, as each physical machine is too small to accommodate even just a single virtual machine.

Formally, we say that a solution does not exist if :

$$\sum_{i=1}^{G} D^{i1} > \sum_{i=1}^{M} R^i \tag{4.7}$$

or

$$\exists i \; s.t. \; D^{i1} > R^j \; \forall j \tag{4.8}$$

so we could start from a small set of physical machines, verify by eq. 4.7 and eq. 4.8 if a solution whether could exist or not. If not, we increase the set of physical machines: sooner or later we find a feasible set, and we could try to map over it. If we find a solution, we could assume that this is the smallest available set of required physical machines.

The proposed model allow for coexistence of quantitative resources (like CPU power or number of cores, memory size) and qualitative resources. As an example, we might want to deploy the multi-tier distributed system in two different areas (two different LANs, or two geographically remote sites). To do so, we extend the quantitative model, by defining two new qualitative resources, called q_1 and q_2. Resource q_1 means "allocation in the first area", whilst resource q_2 means "allocation in the second area". Recall that each tier is comprised of virtual machines of the same type, i.e. with the same resource demand vector: we extend this vector to accommodate for the new resources, and we put $q_1 = 1$ for the first half of nodes of the tier, and $q_2 = 1$ for the second half, meaning we want half nodes in the first area and half nodes in the second area. Lastly, for the physical ma-

chines located in the first area, we put the provided resource $q_1 = M/2$ and the provided resource $q_2 = 0$, and the converse for the physical machines located in the second area.

As a result, each solution of the mapping problem will map half nodes of each tier (ones for which $q_1 = 1$) on the physical machines located in the first area, and remaining nodes on physical machines located in the second area, thus giving us a geographical distribution of the system.

This approach could be applied for other qualitative resources as better connection to storage area networks, software licenses restrictions, hardware support for virtualization and so on.

4.2 The mapping problem as a generalization of the knapsack problem

The mapping problem is a generalization of the well known knapsack problem [75].

The generalization stems from these considerations:

- the knapsack problem is mono-dimensional, whilst the mapping problem is multi-dimensional;
- the knapsack problem is with only one knapsack (physical

machine), whilst the mapping problem deals with multiple knapsacks (physical machines);

- the knapsack problem doesn't group items (virtual machines), the mapping problem does.

To the best of our knowledge, there are no published studies (in the field of computing performance modeling or operational research) that tackle all these generalizations together.

Multi-dimensional knapsacks, called MDKP, are discussed in [41, 48]. Multi-knapsacks problems are studied in [60, 53, 46]. Multiple-choice knapsacks, problems where items are grouped together, are called MCKP and a minimal algorithm to solve them is shown in [85]. In [104] the algorithm is used in the context of QoS for web services.

Some intersections have been evaluated: MMKP are multiple-choice, multiple-dimensional knapsack problems, and heuristics to solve them are discussed in [32, 65, 33, 66]

The only reference we have found to a multi-dimensional, multiple-choice, multiple knapsacks problem, that henceforth we call MMMKP problem, is in [94], but only as a definition. The context there was the definition of an admission control system for multimedia servers, but in that case the only arbitrated resource was Internet bandwidth.

The MMMKP model will be simplified in a MMKP if we have only one machine for each group, i.e. $g_i = 1 \; \forall i$. Although the performance models find an estimation of the number of required nodes for each tier, they usually assume that workload would not experience transient surges. To try to accommodate for peaks in workload intensity, we could over-provision the architecture: it wouldn't change the tier's size or the architecture, but it will improve efficiency in resources using. This is even more realistic if the provider and the owner of the multi-tier system belong to the same organization, as in this scenario the solution of the mapping problem is the minimum required level of service, while every extra computing power put in use (and therefore not wasted) will be appreciated. The MMKP, as stated, is studied in the scientific literature, so we choose to concentrate on its MMMKP generalization, as the algorithms we devise to solve it are equally applicable to the MMKP problem.

Another point is that being the number M fixed or not will lead us to different problems: if M is fixed, we have a knapsack problem, otherwise we have a multiple-dimensional, multiple-choice bin packing problem, a generalization of the bin packing problem that has the same popularity of the MMMKP in the literature, being almost unknown (a survey of bin packing problems is [50]). To maintain our pragmatic approach, we prefer to

work on MMMKP, as the approaches we will develop to solve it could be easily applied to a generalization of the bin packing problem.

Last, we assume that the hypervisor technology that we adopt to manage the virtual machines suffers of no or little interference, meaning that is capable of perform a robust and fair (physical) resources sharing. If this is not the case, the mapping problem could be more easily defined as a generalization of the Generalized Assignment Problem ([51]).

4.3 Computational complexity of the mapping problem

It's easy to show that each classical knapsack problem could be formulated as a MMMKP.

First, we can generalize a mono-dimensional knapsack problem to a multi-dimensional by substitution of each item weight W_i and knapsack capacity C (both scalar) with respectively vector $W_i = (W_i, 0, ..., 0)$ and $C' = (C, 0, ..., 0)$. Then we can add more dummy knapsacks to have a multiple knapsacks generalization, and these knapsacks are described via capacity vectors $C'' = (0, 0, ..., 0)$, $C''' = (0, 0, ..., 0)$. Generalization to have a

multiple choice problem could be obtained if, for each item described via a W_i vector, we define a group, with the first element, $W_i^0 = (0, 0, ..., 0)$ and the second element $W_i^1 = W_i$. Profit for W_i^0 is 0, and profit for W_i^1 is the profit associated with the original item i in the knapsack problem.

As a result, because the knapsack problem is NP-hard [75], we get that MMMKP is NP-hard.

A knapsack problem with N items has a solution space of the size $\Theta(2^N)$ as the decisional variables associated with each item are expressed as $x_i = 1$ if we choose the item or $x_i = 0$ otherwise.

In the MMMKP, we observe that we choose one only virtual machine from each group, and the chosen one is mapped over only one of the available M physical machine. So, for a group made up of g_i elements, we have $g_i * M$ different decision variables, of which only one will be set to 1.

For all the groups, this leads to a solution space size of $\Theta(g_1 * M * g_2 * M * ... * g_G * M) = \Theta(\prod g_i * M^G)$, where we have the Θ notation as the mappings are not independent from each other (when one virtual machine is mapped over a physical machine, there are less available physical resources).

To complete analysis, if we assume that $g_i = k$ for each i, the solution space size is $\Theta(k^G * M^G)$. For $M = 1$ and $k = 2$,

we have the solution space size of a classical knapsack problem.

4.4 Optimal solution of the mapping problem

A naive approach to find the optimal solution of the mapping problem will consist of enumerating all the combinations of the decision variables: for each combination we first check if the constraints are not violated; for all the feasible combinations the associated profit P is compared against the previous maximum: if it's bigger the current configuration of the decisional variables is considered the best solution found insofar.

The enumeration will keep care of guarantee for the constraints 4.4, as for each group it consider only all the acceptable combinations that are, as seen in paragraph 4.3, in number of $g_i * M$. For each one of these combination for the first group, the combination of the second group are evaluated, and for each combination of these, the combination of the third group are evaluated and so on in an iterative way.

The enumeration tuples we produce in this approach could be represented in a hierarchical way, putting them in a decision tree (see figure 4.1).

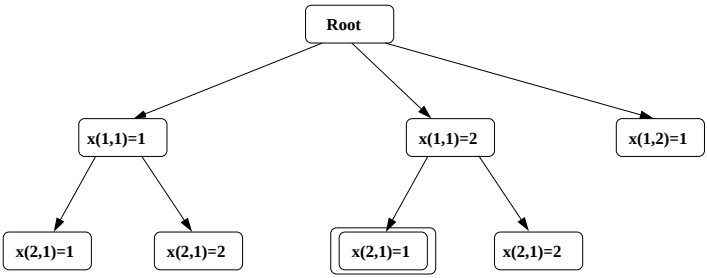

Figure 4.1: A partial decision tree for a MMMKP problem. In the double checked leaf, set decisional variables are $x_1^{11} = 2, x_1^{21} = 1$.

Each node of the tree contains the values of the decisional variables that have already been chosen, while the leafs contain the values of the decisional variables that are under evaluation. Evaluation consists of two phases. In the first phase we check if, for the (partial) solution which is described by the nodes in the path from the current leaf to the root of the tree, some of the constraints are violated. If so, there is no need to further develop the tree, because the current node and all its descendants will violate the constraints. The node is then marked and we move on to evaluation of another node. Second phase of the evaluation is the generation of all the possible descendants, that are proposed assignments for the decisional variables of the

next group (at level i of the three, we have the values of the decisional variables for the i-th group). If there are no leafs to be evaluated, the enumeration process is done, and we have found (one of) the optimal solution for the problem.

This algorithm uses a simple method to cut the developing of the tree, that could be improved implementing a branch and bound technique.

To do so, we consider a linear relaxation of original problem, where the constraints 4.4 and 4.5 are relaxed by these:

$$\sum_{j=1}^{g_i} x_m^{ij} = 1, \ \forall m \tag{4.9}$$

$$\sum x_m^{ij} D^{ij} \leq R^m, \ \forall m \tag{4.10}$$

where decisional variables x_m^{ij} are real numbers in the range [0..1].

For each leaf of the decision tree, we have some of these variables that are fixed, and others that are free. We find the optimal solution that maximizes 4.3 via the Simplex Method. This solution is an upper bound of the solution for the original (integer) problem, because allowing for decisional variables to take fractional values potentially lead to a better use of the available knapsacks capacities. This upper bound will be evalu-

ated against the current optimal found for the integer problem, and the corresponding sub-tree will no further expanded if the upper bound is less than the optimal found insofar. Otherwise, the sub-tree is promising, and we could afford to expand it.

The Simplex Method could grow exponentially in the time it takes to find an optimal solution for a given set of free variables, and the number of trees to be expanded and evaluated will be exponential in the number of decisional variables. This confirms that the MMMKP problem is an NP-hard problem, and that an optimal solution couldn't searched but for problems of limited size.

4.5 Approximate solutions for the mapping problem

As the mapping problem is NP-hard, we are forced to develop algorithms to find approximate solutions of it. We both developed an heuristic to deal with it [44] and a genetic algorithm [45].

4.5.1 A packing oriented heuristic

As discussed, the MMMKP problem is very similar to the bin packing problem, so we could define some heuristics that resemble the ones used for the bin packing problem. In all of the proposed heuristics, we start searching for a basic solution, that later we try to improve.

A basic solution is when we consider, for each group i, only the item $i1$ - with associated resource demand vector D^{i1} - to be mapped: we start solving the mapping problem by reducing it to a multi-dimensional multi-knapsack problem. If we find a solution for it, we try to improve the solution, considering if we could map $D^{i(j+1)}$ instead of the currently D^{ij} in solution. In the following, we adopt naming conventions from operational research, so item ij, will be indicateb by its size D^{ij}.

Next Fit For each item D^{i1} we search for a knapsack with sufficient available space. If there is one, we put D^{i1} into it, and we correspondingly reduce the available space. We start with all the knapsacks open, and we close one when the available space is insufficient for the D^{i1} item. When a knapsack is closed, we no further inspect it to see if has sufficient available space for an item. We have a basic solution if each D^{i1} item has been put into a knapsack (some knapsacks will be open, other closed,

and it's possible that there are knapsack completely unused), otherwise the heuristic fails.

First Fit In the First Fit heuristic, we search for a destination of the item D^{i1} by inspecting all available knapsacks, i.e. we no longer have open or closed knapsacks. We find a basic solution in the same sense of Next Fit, i.e. when each item D^{i1} has been put into a knapsack.

Best Fit The Best Fit heuristic searches between all the available knapsacks the best where to put the item D^{ij}, usually defining best with a metric that tries to minimize the unused resources. A monodimensional example is where we have two knapsack, with available space respectively $R^1 = 3$ and $R^2 = 4$ and we need to insert the item $i1$ such that$D^{i1} = 3$. Both knapsacks could host it, but the heuristic choose the first, to make it completely used and leave a little unused capacity in the second. In a multi-dimensional bin-packing problem, ties are resolved in favor of lower (resource) index ore more complex evaluation of relative scarcity of each resource, as we'll do. We find a basic solution in the same sense of Next Fit, i.e. when each item D^{ij} has been put into a knapsack.

Improving the basic solution We improve the solution found insofar in an iterative way.

At the generic step, we have chosen a specific item from each group. Assume that for the group i we have item j chosen on machine m, i.e. $x_m^{ij} = 1$ and $x_m^{i(j+1)} = 0$. If $j = g_i$ we already have the most profitable item from group i so we move on another group to check for possible increases.

Otherwise, we could remove item j from group i from solution, releasing associated resources on knapsack m, and we see if and where we could put in solution item $j+1$ of the same group. This requires considering all available knapsacks, finding the most suitable one to contain such item. If a generic knapsack k as enough free resources for the item, we evaluate the goodness of the mapping by this formula:

$$Goodness(i,k) = \frac{\|R\|}{P^{i(j+1)} - P^{ij} - (1-u_k) * C} \tag{4.11}$$

In eq. 4.11:

- the numerator is the vector norm of the residual amount of resources available on knapsack k after we put item $i(j+1)$ in it;

- the denominator is the increase in profit we have ($P^{i(j+1)} - P^{ij}$) minus the possibility that we may end up using a knapsack k that was not yet used ($1 - u_k * C$);

Eq. 4.11 makes sense only if the denominator is bigger than zero, i.e. only if we have some profit gain. In each phase of the improvement process, we calculate $Goodness(i, k)$ for each acceptable value of i (groups with more valuable items) and k (knapsacks with available resources). Lowest values of $Goodness(i, k)$ are better, so we choose the minimum positive one, and we perform the necessary corrections on the solution we are working on (i.e., we set $x_m^{ij} = 0$ and $x_k^{i(j+1)} = 1$). We repeat this process as long as we have made improvements on the current solution.

Each pass has a computational complexity of $\theta(G * M)$.

Randomization of data The proposed heuristics are strongly based on the order by which groups and knapsacks are defined. We cannot stipulate that it exists an order of these variable such that the proposed heuristic could always find the optimal solution, but we are confident that if we permute the groups and the machines before actually building up a basic solution we could increase the final profit as a result of the application of the basic heuristics shown before. We observe that there is not a general criterion to discriminate between good permutations that lead

us to find better solutions and bad permutations, and also these good ones are less than statistically rare, unless that P=NP.

Other bin packing heuristics In most of the scientific literature for the bin-packing problem ([50]) it is assumed that all the resources are dimensionally homogeneous. As an example, if the problem is to put cans into a container maximizing used space, the cans could be rotated, so elements of vectors D^{ij} could be interchanged. This is not the case of the mapping problem. Also, heuristics are evaluated with a predetermined set of elements, that are used as a comparative basis. These elements are defined as outcomes of some random variables, assumed independent of each other, while in our problem this is generally not true, for two distinct reasons. First, there is some degree of intra-dependencies, i.e. if a virtual machines requires a lot of CPU power it will requires (on average) more memory than a machine that requires less CPU power. Second, there is a degree of inter-dependencies, as all machines belonging to the same tier will share their resource demand vector.

4.5.2 A genetic algorithm

A genetic algorithm could be seen as an intelligent probabilistic search in the space of solutions for an hard problem.

Starting from the name itself, the terminology of the genetic algorithms is derived from the evolutionary biology, where individuals stem from a population by a recombination of genetic characteristics of their parents, plus a small probability of some random genetic mutation. Some mutations are for the better, giving the individual an higher chance to become a parent of a new individual (that could inherit this advantageous mutation), other mutations are for the worst, and the individual carrying them will have a smaller chance to become parent.

Genetic algorithms have been widely considered as an optimization strategy for hard optimization problems, where it's easy to find some solutions but it's very difficult to find the optimal, as these initial solutions could be the initial population from which start the search for the optimal one.

In the field of integer programming, the mapping between an individual and a solution is usually really simple, as he i-th chromosome of the individual is 0 (or 1) if and only if the i-th decision variable of the portrayed solution is 0 (or 1). Although more complex representations are possible [52] we choose to stick with this.

Genetic algorithm are not a free lunch in the field of optimization when they are applied to a constrained optimization problem, as the result of recombination and mutation of two fea-

sible element (i.e., individuals that represent feasible solutions) could not be feasible. The mapping problem is particularly complex from this point of view. In fact we have two different set of constraints, the first that requires we choose only one element from each group, the other that we don't overfill a knapsack. As these two set of constraints must be enforced together, we cannot adapt a simple 'repair' operator to deal with unfeasible individual (i.e., individual representing unfeasible solutions), as has been done in [48] where, should a knapsack be overfilled, elements are removed from it until the violation is fixed: we cannot do that as we *must* allocate exactly one element from each group. The approach we have adopted is to consider our constraints as belonging to two different sets: easy and hard. An easy constraint is a constraint that, should an individual violate it, we could easily fix, while hard constraints require a complementary approach, based on the use of penalty function.

Formally speaking, if we have this optimization problem:

$$\begin{cases} \max f(\mathbf{x}) \\ \mathbf{x} \in E \\ \mathbf{x} \in H \end{cases} \tag{4.12}$$

where E and H represents respectively easy and hard con-

straints, we transform problem 4.12 into this one:

$$\begin{cases} \max\ f(\mathbf{x}) - p(d(\mathbf{x}, H)) \\ \mathbf{x} \in E \end{cases} \tag{4.13}$$

where $d(x, H)$ is a metric function describing the distance of solution x from the set H of feasible solutions, and $p(\cdot)$ is a monotonically non-decreasing function such that $p(0) = 0$. Penalty functions are surveyed in [34]. For our model, constraint 4.4 is easy, so we define a repair operator for individuals that violate it, while constraint 4.5 is hard, and it will be handled via a penalty function.

Outline Our genetic algorithm starts with a population that is made up of individuals representing basic solution for the problem, i.e. solutions where only the lowest SLA of each virtual machine has been chosen to be allocated. We generate these solutions by using the first-fit, best-fit and next-fit heuristic from the bin-packing problem, with a randomization of the data to generate initial different solutions.

We must take care that we don't insert into the population an element that is already in, to avoid that we unnecessarily reduce the initial population size. After this initialization step,

we do an iterative process, each cycle of it called a generation, where we:

1. Choose the two parents of the new individual, by a tournament process;
2. Create the new individual by a crossover operator;
3. Mutate some variables of the new individual with a mutation operator;
4. Calculate the fitness of the individual, taking care of unfeasibility due to overfilling;
5. Fix the easy constraint with the repair operator;
6. Insert this individual into the population, and remove the individual with the lowest fitness.

These steps are all tunable by some parameters, resulting in different instances of the same genetic algorithm. We discuss each of these steps in the following paragraphs.

Tournament Process To choose the two parents that will generate a new individual, we randomly define two different pools of all different elements from the population. From each pool, we choose the element with the highest fitness as one of

the two parents. A larger pool will increase the competitive pressure.

Crossover Operator After the selection of the two parents, the new individual will be defined as the crossover of them. Instead of adopting a random crossover we do an uniform crossover [37], where the probability that the i-th variable of the new individual is equal to the i-th variable of the first or second parent is proportional to the fitness of the first or second parent.

Mutation Operator Mutation rate is fixed. A more complex approach would be a dynamic mutation rate, with an higher rate for the initial generations (when we are probably away from the optimal solution, so we can change a lot of variables) and a lower rate as the generations pass away. This is a critical parameter, as an high rate could destroy the stability of the genetic algorithm, and a low rate could end up in being trapped in a local minimum.

Fitness and penalty function At a first glance, one should be tempted to consider the objective function 4.3 as the fitness function, but this will result in even completely different individuals with the same fitness, when we want to differentiate as

much as possible in order to pick up, from the tournament process, the potentially best individuals by looking at their fitness and not only by chance.

We have also to include the penalty function in the fitness computation, so we are already considering a different problem than the original one, but we must define the fitness function so individuals with better values of the fitness are, on average, better solutions for the original problem.

We observe that, if we have two different and feasible solutions x and x' with the same value of the objective function 4.3 and the same number of physical hosts used, we can still say that x is better than x' if x packs more virtual machines in the same physical host, while x' allocate virtual machines more evenly; this because it's more probable that, from the solution x, we have more unused resources in some physical hosts and we can use these resources to allocate some others virtual machines, without changing the number of physical hosts used; while for solution x' unused resources are not aggregated together. Formally speaking, for a solution x of the formal problem, we define the relative amount of unused resources for each physical hosts as:

$$rel_k^m = \frac{r_k^m - \sum_{i=1}^{G} \sum_{j=1}^{g_i} x_k^{ij} * d_k^{ij}}{r_k^m} \tag{4.14}$$

From this definition, we have that rel_k^m is not negative when the knapsack l has some unused resource k, and it's less than zero when we have overfilled it with respect to that resource. We can leverage on this property of rel_k^m using it both for rewarding feasible individuals and for penalizing unfeasible individuals.

To do so, we need also to define the portion of the profit on a per physical hosts basis, i.e. the profit we earn for the virtual machines allocated on a specific physical hosts:

$$Gain(m) = \sum x_m^{ij} * P^{ij} \tag{4.15}$$

we need this value to deal with the unfeasibility that arises after overfilling a physical host: in such a case, we cannot say which virtual machines is 'guilty', and we have to decrease the total profit for the portion of the profit we earn from all the virtual machines allocated over this overfilled physical host (the inability to say which virtual machines is guilty is what makes difficult to define a repair operator and forces use to search a suitable penalty function).

Conversely, when the host is not overfilled, we could increase

the fitness, and the more resources are relatively free (hosts by hosts), the more we increase the fitness. Putting all together, we define the fitness function as:

$$F(\mathbf{x}) = P(\mathbf{x}) + \sum_{i=1}^{K} \sum_{m=1}^{M} Gain(m) * \frac{\alpha}{K} * rel_k^m \tag{4.16}$$

The quantity α is used as a static multiplier: if $\frac{\alpha}{K}$ we reward and penalize individuals more aggressively.

The Repair Operator We define a repair operator to ensure that eq. 4.4 holds for each individual. This equation requires that, for each group i, we have exactly one element set to 1, all others being 0. We could express in a different way by stating that, for each group i: 1) there is at least an element different than 0; 2) there is no more than one element different than 0. By such separation, we can define two specialized operators. Each individual is an ordered collection of groups, and eq. 4.4 could hold for some groups and not for others. So we scan all the groups comprising the individual to check for property 1, we somehow fix the groups that don't verify it, and then we rescan all the groups to check and possibly fix for property 2.

To describe the process, consider an individual made up of 3 groups (which means that we are searching for the optimal al-

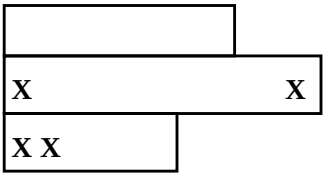

Figure 4.2: An individual for a problem with 3 groups. Each X marks a variable set to 1.

location of 3 virtual machines) where first group has no element set to 1, and second and third group both have 2 elements set to 1 (see figure 4.2 for a pictorial representation).

To ensure that each group has at least one non-zero element, we need to fix the first group. We could this randomly, by choosing one element of the first group and setting it to 1, or we could start a neighborhood search. In this search, we generate g_1 new individuals, each one of them completely equal to the individual we are fixing, but with the i-th variable of the first group set to 1. (see figure 4.3).

For each of these individuals, we evaluate the fitness (our fitness function takes care of unfeasibility, so we can safely use it) and we choose the individual with the highest fitness as the repaired individual. If we have more than one group that needs this fixing, we perform it in an iterative way, group after group.

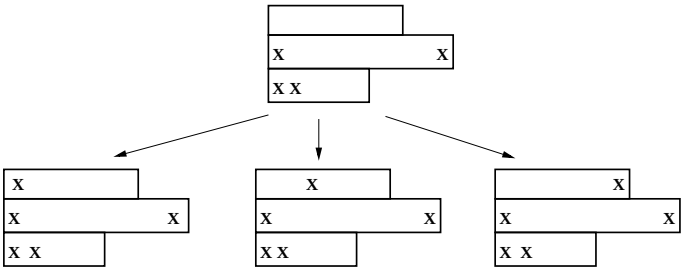

Figure 4.3: Fixing the first group with different individuals.

Now we have to ensure that each group has no more than one non-zero element, so we need to fix the second group. We generate two individuals, where the i-th individual has set to 1 only the i-th non-zero variable of the second group, and again we choose the best among them. Then we repeat the process for the third group (see figure 4.4).

We stress that, when we create the neighborhood list, we have partially unfeasible individuals in it, but we can cope with this as the fitness function is robust enough. We could have been put in place a more complex research when we consider all the possible combinations (see figure 4.5), but we have chosen not to do this for this version of the genetic algorithm, as efficiency should be carefully evaluated, especially for the size of

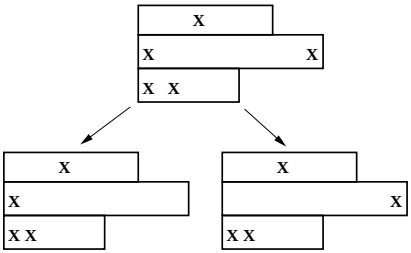

Figure 4.4: Fixing the second group by generating two different individuals.

the explored set of neighborhood elements.

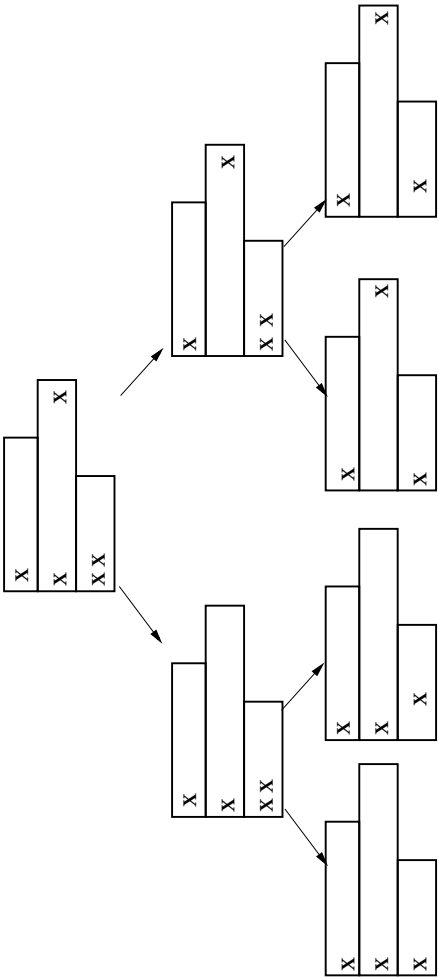

Figure 4.5: Fixing the second and third group by generating all possible feasible individuals.

5

Simulations results

In this chapter we analyze the performances of the heuristics and the genetic algorithm we have proposed in Chapter 4, considering four different problem sets. In the field of the operational research and integer programming, there are some datasets for the most classical problems that one could use to test against a new algorithm, so it's possible to compare the relative performances between different researchers. At the time being, we don't have the same for virtualized architectures, and we were unable to find publicly known data depicting a distributed system that has been virtualized: we were forced to consider some-

what arbitrary models of distributed systems, and we are also aware that the performances of our solvers (particularly for the genetic algorithm) cannot be fully understood and determined without knowing the class and the structure of the problems.

From an implementation stand-point, all the programs we have made are written in C, and they extensively use the GNU Scientific Library (GSL) [7] as we need to to deal with random numbers. We don't stress too much in the sense of their efficiency, as we are more interested in their robustness: nevertheless, each dataset is solved within few minutes, on a commodity computer.

5.1 Implementation of the bin packing heuristics

All the proposed bin packing heuristics find a basic solution, which is a solution when only the lowest SLA of each virtual machine is instantiated over a physical host. We define $mapping[i]$ has such physical host, so if we have $mapping[i] = j$ we have, in our formal model, that $x_{i1}^{j} = 1$. We have implemented the Next Fit, First Fit and Basic Fit bin packing heuristics.

Next Fit In the Next Fit (algorithm 1) we start considering all physical host as initially available (open). Then, for each virtual machine, we consider each open physical host: if it could contain the virtual machine, we put the latter in the former, thus reducing the available resources, and setting $mapping[i]$ properly; otherwise, we close the physical host, so we'll no longer consider it in further iterations. These closings reduce the set of available physical host over iterations. It's easy to show that computational complexity of this heuristic is $\Theta(G * M)$.

First Fit In the first fit (algorithm 2), we consider all physical hosts as possible destinations for the lowest SLA of each virtual machine, in a strict order starting from the first physical host and then moving on. The first host we find that has sufficient available resources, will be the chosen host. As in the Next Fit, computational complexity is $\Theta(G * M)$.

Best Fit In the Best Fit heuristic (algorithm 3), we have first to define which resources of the K we have in our model is the scarcest. The scarcest resource is the one where the ratio between the grand total of it, as provided by the physical hosts, and the grand total of requested by the lowest SLA of all virtual machines, is lowest. The scarcest resource r is the resource

Algorithm 1 Next Fit Algorithm

```
for i = 1 to M do
  open[i] ← true
end for
for i = 1 to G do
  mapping[i] ← −1
end for
for each virtual machine i do
  for each host j do
    if open[j] then
      if host j has sufficient resources for lowest SLA of virtual machine i then
        mapping[i] ← j
        reduce available resources for host j
        move to next value of i
      else
        open[j] ← false
      end if
    end if
  end for
end for
if ∃i s. t. mapping[i] = −1 then
  print Unable to find a basic solution
end if
```

Algorithm 2 First Fit Algorithm

for $i = 1$ *to* G **do**
  $mapping[i] \leftarrow -1$
end for
for each virtual machine i **do**
  for each host j **do**
    if host j has sufficient resources for lowest SLA of virtual machine i **then**
      $mapping[i] \leftarrow j$
      reduce available resources for host j
      move to next value of i
    end if
  end for host j
end for virtual machine i
if $\exists i \, s.t. \, mapping[i] = -1$ **then**
  print Unable to find a basic solution
end if

which should drive our mapping process, as there is not so much of it.

After this determination, for the lowest SLA of each virtual machine, we first determine the set of physical hosts that have sufficient available resources (defining $has\_space[j] = true$ if host j has this property). From all these hosts, we associate the virtual machine and the best host. The best host is the host where mapping of the virtual machine will result in the minimization of the residual resource r. This heuristic has a computational complexity of $\Theta(G * M)$ but the hidden proportionality factor is the highest.

Randomization of data The three heuristics process data in a strict order, while the mapping problem does not change if data are reordered (as an example, by swapping elements of group i with elements of group j). So we decide to allow for a permutation of the D vector (and the associated P) vector. Permutation is defined randomly by using the `gsl_ran_shuffle()` function of the GSL library.

The random number generator used is the Mersenne Twister [77] implemented by GSL, and we consider 10,000 runs of each of the heuristics, each time changing the random number seed. For each of these runs, we improve the solution by the heuristic

Algorithm 3 Best Fit Algorithm

for $i = 1$ *to* G **do**
  $mapping[i] \leftarrow -1$
end for
for each virtual machine i **do**
  for each hosts j **do**
    $has\_space[j] \leftarrow$ **false**
  end for
  for each host j **do**
    if host j has sufficient resources for lowest SLA of virtual machine i **then**
      $has\_space[j] \leftarrow$ **true**
    end if
  end for
  for each host j s.t. $has\_space[j]$ is **true do**
    choose the *best* host, b
    reduce available resources for host b
    $mapping[i] \leftarrow b$
  end for
end for
if $\exists i \; s.t. \; mapping[i] = -1$ **then**
  print Unable to find a basic solution
end if

defined by eq. 4.11 discussed before.

5.2 Implementation of the genetic algorithm

As is for each genetic algorithm, the tuning of the parameters is particularly complex, and more an art than a science. We believe that we cannot find the best parameters without a deep understanding and analysis of real models; as our models are somewhat arbitrary, we choose to make the more simplistic assumptions:

1. The size of the tournament process is 5, so we draw a pool of all different 5 individuals to find each parent: this means a very high competitive pressure (usually each pool is made up of 2 individuals);

2. The mutation rate is proportional to 3 times the number of decisional variables set to 1, and it's fixed all along the simulation. It's a rather high value;

3. The value of α from eq. 4.16 is set to 2.4; being $K = 2$ this means that the multiplier $\frac{\alpha}{K}$ is bigger than 1;

4. The initial population consists of 300 individuals, 100 for each of the bin-packing basic heuristics seen before; as we remove the individual with the lowest fitness at each generation, the population size remains stable throughout the simulation;

5. We run the genetic algorithm for 2,000 generations.

5.3 Models dataset

We consider two models to test the heuristic, plus two different models to test the genetic algorithm. For all but the smallest of them, the time it takes to find the optimal solutions via a tool like GLPK [6] is so long that we didn't see the linear programming solver coming to an end.

Tables 5.1 and 5.2 show the characterization for the first and second model, the ones on which the heuristic is tested. Both models depict a three-tier; as an example for the first model (table 5.1) the first tier (the web tier) is made up of 2 nodes, each one having three different services of levels: the lowest level of service requires 1 CPU core, 2 GiB of RAM and gives a profit of 2 units, where the intermediate level requires 2 CPU cores and 4 GiB of RAM, and the profit goes up to 4. The highest level

returns us a profit of 8, but it requires 3 CPU cores and 4 GiB of RAM. All the fourth models are 2-dimensional, as it will be difficult to find reasonable characterization of other resources.

Table 5.3 reports the grand total of physical resources for all the four models. Note that not all hosts are equal in the amount of resources they provide.

Although the second model is not so much bigger than the first, the increase in its size (both in virtual machines number, SLAs and number of physical hosts available) makes difficult to find the optimal solution: the linear programming solver takes some seconds to find the optimal solution for the first model, while on the second we have only the range where the profit of the optimal solution lies after hours of computation (on an Intel Xeon 1.86 GHz).

So, while in table 5.4 we compare the profit for the optimal solution and the best profit found by the heuristic, on table 5.5 we cannot do better than compare the range of profit with the best profit found by the heuristic. We consider, for this two different models, different values of C.

There is a common result we can see: for small value of C the heuristic performs extremely well, being capable to find the optimal solution or a solution really close to it. When C increases, the heuristic is no more able to find an optimal solution. This

Tier	Size	CPU cores	RAM (GiB)	Profit
Web	2	1/2/3	2/4/4	2/4/8
Application	6	2/2	2/4	2/6
Database	2	2/4	2/4	2/4

Table 5.1: First model SLAs and profits.

Tier	Size	CPU cores	RAM (GiB)	Profit
Web	2	1/2/4	2/4/6	2/4/8
Application	6	2/4/4	2/4/6	2/4/6
Database	2	2/2	4/6	2/6

Table 5.2: Second model SLAs and profits.

may indicate that the value of $Goodness(i, k)$ as computed by eq. 4.11 is too much sensitive to the value of C.

Tables 5.6 and 5.7 describe the third and fourth model, inputs for the genetic algorithm. These models have a structure analogue to the first two models, with some minor variations. In both cases, the value of C is set to 1.

Table 5.8 reports the fitness of the best individual of the initial population and the fitness of the best individual at the end of the simulation.

The increase in individual's fitness is clearly evident. On figure 5.1 the average fitness of the population over the simula-

Model	Hosts	CPU cores	RAM size (GiB)
First	6	48	48
Second	8	64	88
Third	8	64	88
Fourth	12	96	192

Table 5.3: Physical hosts characterizations for all models.

C	Optimal	Heuristic
0	60	60
1	55	53
2	50	46
3	45	39

Table 5.4: Comparisons of profits for the optimal solution and the approximate solution for the first model, for different values of C.

C	Range of optimal profit	Heuristics
0	$87 \div 96$	84
1	$79 \div 95$	69
2	$71 \div 93$	54
3	$63 \div 89$	39

Table 5.5: Second model: range of profit and approximate solution profit, for different values of C.

Tier	Size	CPU Cores	RAM (GiB)	Profit
Web	4	1/2/4	2/4/6	2/4/8
Application	8	2/4/4	2/4/6	2/4/6
Database	3	2/2	4/6	2/6

Table 5.6: Third model SLAs and profits.

Tier	Size	CPU cores	RAM (GIB)	Profit
Web	6	1/2	2/4	2/4
Application	12	1/4	2/4	2/4
Database	4	2/2	4/6	2/6

Table 5.7: Fourth model SLAs and profits.

Model	Initial best fitness	Final best fitness
Third	33	109
Fourth	39	166

Table 5.8: Initial and final fitness of best individual for third and fourth model, as seen by the genetic algorithm.

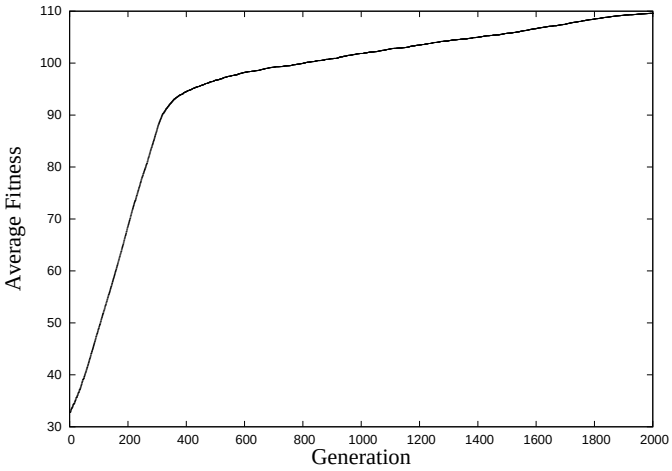

Figure 5.1: Average fitness of population for the third model.

tion for the third model is depicted, while figure 5.2 is for the fourth model. The average fitness value is higher than the fitness shown in table 5.8, as the average is all over the population, including the unfeasible individuals.

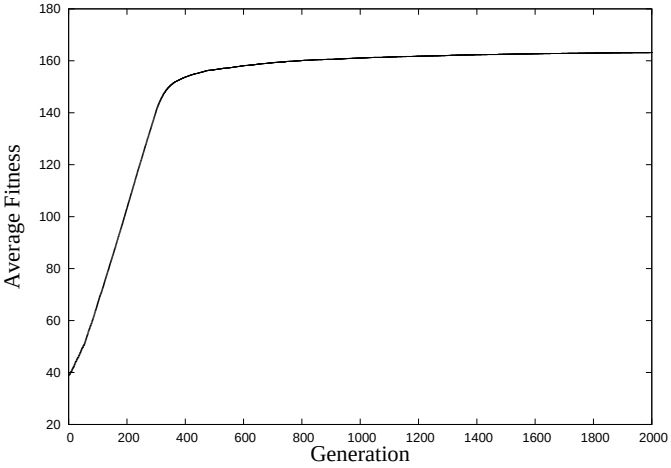

Figure 5.2: Average fitness of population for the fourth model.

6

Conclusions

> It is the fate of every voyager, when he has just discovered what object in any place is more particularly worthy of his attention, to be hurried from it. (Charles Darwin, *Voyage of the Beagle*)

Operating system level virtualization will be fundamental in designing and deploying computing architectures int the next following years.

The increasing concerns for environment protection and the rising prices of electricity are some of the drivers for maximize computing efficiency. Virtualization will be helpful in achieving these goals, and has also some more technological benefits, as it's capable of isolating systems to allow for better security and makes easier to implement disaster recovery solutions.

These key benefits are possible as virtualization is another layer connecting to and interposing between hardware and operating system, thus hiding physical heterogeneity and boundaries. As virtualization is progressively made in hardware, reducing at a fraction the penalty due to the layer itself, its adoption will further increase, and this will in turn lead to a new generation of problems.

These problems begins to appear in the field of autonomic systems, where the ability to self-adapt the system to the changing workload condition is of the utmost importance, but other fields with similar resources allocation problems are the emerging cloud computing and, to some extent, the grid computing.

In this dissertation we have formalized the mapping problem, which is a generalization of the classical $0/1$ knapsack problem, unknown before this work. The mapping problem requires to find, for a given set of virtual machines, each one characterized by a multi-dimensional resource demand vector, one physical

machine that has sufficient resources to host it. Physical machines are from a given set, and are also characterized by an analogous multi-dimensional vector, describing their allowable resources. Each one of the virtual machines has possibly some different service levels, with increasing resources and increasing profits that are earned when the allocation is made. The objective function is represented by a sum of profits, that could be mitigated by the number of physical machines that are used.

We have determined the complexity of the mapping problem, showing that is a NP-hard problem, and we have proposed an heuristic and a genetic algorithm to deal with it.

Both the proposed approaches have been valuable in finding an approximate solution for the mapping problem. We believe that they would both perform well and complement each other in a real scenario.

Operating system level virtualization is, in some sense, a new territory to explore, that could lead us to better, more efficient and more resilient distributed systems. Some of the assumptions we have made in the past about how to build a multi-tier distributed systems must be rediscussed, and to do so limits and benefits of virtualization must be clearly understood, properly formalized and methodologically analyzed.

This dissertation aims to be a first step in that direction.

Bibliography

[1] *Amazon Elastic Computing Cloud*, http://www.amazon.com/gp/browse.html?node=201590011

[2] *Amazon Simple Storage Service*, http://en.wikipedia.org/wiki/Amazon_S3

[3] *AMD64 Architecture Tech Docs Volume 2*, http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/24593.pdf

[4] *AMD I/O Virtualization Technology (IOMMU) Specification*, http://www.amd.com/us-en/assets/content-type/white_papers_and_tech_docs/34434.pdf

[5] *AMD Nested Page Table performance benchmark*, http://www.redhatmagazine.com/2007/11/20/red-hat-enterprise-linux-51-utilizes-nested-paging-on-amd-barcelona-processor-to-improve-performance-of-virtualized-guests/

[6] *GNU Linear Programming Kit*, http://www.gnu.org/software/glpk

[7] *GNU Scientific Library*, http://www.gnu.org/software/gsl/

[8] IBM S/360, *http://en.wikipedia.org/IBM_360*

[9] *Intel Extended Page Tables*, http://www.intel.com/technology/itj/2006/v10i3/1-hardware/8-virtualization-future.htm

[10] *Intel Virtualization Directed I/O*, http://www.intel.com/technology/itj/2006/v10i3/2-io/7conclusion.htm

[11] *Intel Virtual Machine Device Queues*, http://www.intel.com/technology/platform-technology/virtualization/VMDq_whitepaper.pdf

[12] *Intel Virtualization Architecture*,
http://www.intel.com/technology/itj/2006/v10i3/1-hardware/5-architecture.htm

[13] ISCSI,
http://en.wikipedia.org/ISCSI

[14] *Lguest site*,
http://lguest.ozlabs.org

[15] *Linux-VServer site*,
http://linux-vserver.org

[16] *QEMU site*,
http://fabrice.bellard.free.fr/qemu/

[17] *Qumranet, Linux Kernel Based Virtual Machine*,
http://kvm.qumranet.com

[18] *OpenVZ site*,
http://openvz.org

[19] *Self-Service, Prorated Super Computing Fun*,
http://open.blogs.nytimes.com/2007/11/01/self-service-prorated-super-computing-fun

[20] *Solaris Containers (Zone)*,
http://www.sun.com/bigadmin/content/zones

[21] *The UC Berkley/Stanford Recovery-Oriented Computing (ROC) Project*,
http://roc.cs.berkeley.edu

[22] *Virtualbox Site*,
http://www.virtualbox.org

[23] Virtual Local Area Network,
http://www.cs.wustl.edu/~jain/cis788-97/ftp/virtual_lans/index.htm

[24] *VMWare site*,
http://www.vmware.com

[25] *Citrix Xen Server*,
http://www.citrixxenserver.com

[26] *Xen University of Cambridge Computer Lab site*,
http://www.cl.cam.ac.uk/research/srg/netos/xen

[27] *Xen Assign Hardware to DomU*,
http://wiki.xensource.com/xenwiki/
Assign_Hardware_to_DomU_with
_PCIBack_as_module

[28] *Xen Network architecture*,
http://wiki.xensource.com/xenwiki/XenNetworking

[29] *Xen Network Paravirtualized Driver performances*, https://bugzilla.redhat.com/show_bug.cgi?id=431898

[30] K. Adams, O. Agesen, *A Comparison of Software and Hardware Techniques for x86 Virtualization*, Proceedings of the Twelfth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS-XII), October 2006, San Jose, California, USA

[31] J. Almeida, D. Ardagna, M. Trubian, *Resource Management in the Autonomic Service-Oriented Architecture*, in Proceedings of the 2006 International Conference on Autonomic Computing ICAC'06

[32] M. M. Akbar, E. G. Manning, G. C. Shoja, S. Khan, *Heuristic Solutions for the Multiple-Choice Multidimension Knapsack Problem*, Lecture Notes in Computer Science, LNCS 2074, Springer Verlag, 2001

[33] M. M. Akbar, M. Sohel Rahman, M. Kaykobad, E. G. Manning, G. C. Shoja, *Solving the Multidimensional Multiple-choice Knapsack Problem by Constructing Convex Hulls*, Computers and Operation Research, vol. 33 issue 5, May 2006

[34] T. Baeck, D. Fogel, Z. Michalewicz, Eds., *Handbook of Evolutionary Computing*, A joint Publication of Oxford University Press and Institute of Physics Publishing, 1995

[35] R. Bhargava, B. Serebrin, F. Spadini, S. Manne, *Accelerating Two-Dimensional Page Walks for Virtualized Systems*, Proceedings of the 13th international conference on Architectural support for programming languages and operating systems (ASPLOS-XIII), Seattle, Washington, USA 2008

[36] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, A. Warfield, *Xen and the Art of Virtualization*, in Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, Bolton Landing, NY, USA, 2003

[37] J. E. Beasley, P. Chu, *A Genetic Algorithm for the Set Covering Problem*, European Journal of Operational Research, vol. 94 1996

[38] F. Bellard, *QEMU, a Fast and Portable Dynamic Translator*, Proceedings of the USENIX Annual Technical Conference, 2005

[39] M. N. Bennani, D. A. Menascé, *Assessing the Robustness of Self-Managing Computer Systems Under Highly Variable Workloads*, in Proceedings of the International Conference on Autonomic Computing (ICAC'04)

[40] M. N. Bennani, D. A. Menascè, *Resource Allocation for Autonomic Data Centers using Analytic Performance Models*, in Proceedings of the Second International Conference on Autonomic Computing (ICAC'05), Seattle, WA, USA

[41] D. Bertsimas, R. Demir, *An Approximate Dynamic Programming Approach to Multi-dimensional Knapsack Problems*, Management Science, vol. 48 issue 4, 2002

[42] S. Biemueller, *AMD ASID Implementation in Xen AMD-V*, Xen Summit Spring 2007
http://xen.org/files/xensummit_4/2007XenSummit-AMD-ASIDS-Biemueller.pdf

[43] R. Bradford, E. Kotsovinos, A. Feldmann, H. Schioberg, *Live Wide-Area Migration of Virtual Machines Including Local Persistent State*, in Proceedings of the 3rd International Conference on Virtual Execution Environments (VIEE'03), San Diego, California, USA, 2007

[44] P. Campegiani. F. Lo Presti, *A General Model for Virtual Machines Resources Allocation in Multi-Tier Distributed Systems*, in Proceedings of the International Conference on Autonomous and Autonomic Computing 2009 (ICAC'09), Valencia, Spain, 2009

[45] P. Campegiani, *A Genetic Algorithm to Solve the Virtual Machines Resources Allocation Problem in Multi-tier Distributed Systems*, submitted to the Second International Workshop on Virtualization Performance: Analysis, Characterization, and Tools (VPACT'09), Boston, Mass. USA, 2009

[46] C. Chekuri, S. Khanna, *A PTAS for the Multiple Knapsack Problem*, in Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithm, San Francisco, CA, USA, 2000

[47] L. Cherkasova, D. Gupta, A. Vahdat, *Comparison of the Three CPU Schedulers in Xen*, Xen Summit Spring 2007, http://xen.org/files/xensummit_4/3schedulers-xen-summit_Cherkosova.pdf

[48] P. C. Chu, J. E. Beasley, A Genetic Algorithm for the Multidimensional Knapsack Problem, Journal of Heuris-

tics, vol. 4 n. 1, Springer Verlag, 1998

[49] C. Clark, K. Fraser. S. Hand, J. Gorm Hansen, *Live Migration of Virtual Machines*, in Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation (NSDI), 2005

[50] E. G. Coffman Jr., M. R. Garey, D. S. Johnson, *Approximation Algorithms for Bin Packing: a Survey*, Approximation Algorithms for NP-hard Problems, PWS, 1996

[51] R. Cohen, L. Katzir, D. Raz, *An Efficient Approximation for the Generalized Assignment Problem*, Information Processing Letters, vol. 100 issue 4, Elzevier, 2006

[52] C. Reeves, *Hybrid Genetic Algorithms for Bin-Packing and Related Problems*, Annals of Operations Research, vol. 63 1996

[53] C. Cotta, J. M. Troya, A Hybrid Genetic Algorithm for the 0-1 Multiple Knapsack Problem, Artifical Neural Nets and Genetic Algorithms 3, Springer Verlag, 1998

[54] B. Cully, A. Warfield, Virtual Machine Checkpointing, Xen Summit Spring 2007,
http://xen.org/files/xensummit_4/talk_ Cully.pdf

[55] W. Curtis Peterson, *Using SANs and NAS*, O'Reilly

[56] S. Dobson, S. Denazis, A. Fernandez, D. Gaiti, E. Gelembe, F. Massacci, P. Nixon, F. Saffre, N. Schmidt, F. Zambonelli, *A Survey on Autonomic Communications*, ACM Transactions on Autonomous and Adaptive Systems, Vol. 1 no. 2, December 2006

[57] U. Drepper, *The Cost of Virtualization*, ACM Queue Volume 6 Number 1, January/Febrary 2008

[58] U. Drepper, *What Every Programmer Should Know About Memory*,
http://people.redhat.com/drepper/cpumemory.pdf

[59] K. J. Duda, D. R. Cheriton, *Borrowed-Virtual-Time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler*, in Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles, Kiawah Island Resort, SC, USA, 1999

[60] C. E. Ferreira, A. Martin, R. Weismantel, Solving Multiple Knapsack Problems by Cutting Planes, SIAM Journal of Optimization, vol. 6 n. 3, 1996

[61] G. Franks, S. Majumdar, J. Neilson, D. Petriu, J. Rolia, M. Woodside, *Performance Analysis of Distributed Server*

Systems, Proceedings of The 6-th International Conference on Software Quality, Ottawa, October 1996

[62] T. Garfinkel, R. Bfaff, J. Chow, M. Rosenblum, D. Boneh, Terra: *A Virtual Machine-Based Platform for Trusted Computing*, Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, Bolton Landing, NY, USA, 2003

[63] E. Van Hensbergen, *P.R.O.S.E. Partitioned Reliable Operating System Environment*, ACM SIGOPS Operating Systems Review, Volume 40, Issue 2, April 2006

[64] E. Van Hensbergen, *The Effect of Virtualization on OS Interference*, http://research.ihost.com/osihpa/osihpa-hensbergen.pdf

[65] M. Hifi, M. Michrafy, A. Sbihi, *Heuristic Algorithms for the Multiple-choice Multidimensional Knapsack Problem*, Journal of Operational Research Society, vol. 55 num. 12, 2004

[66] M. Hifi, M. Michrafy, A. Sbihi, *A Reactive Local Search-Based Algorithm for the Multiple Choice Multi-Dimensional Knapsack Problem*, Computational Opti-

mization and Applications, vol 33. n. 2-3, Springer Verlag, 2006

[67] P. Horn, *Autonomic Computing Vision and Manifesto*, http://www.research.ibm.com /autonomic/manifesto/autonomic_computing.pdf

[68] W. Huang, *Nested Page Table Support*, Xen Summit Spring 2007 http://xen.org/files/xensummit_4/2007XenSummit-AMD-Barcelona_Nested_Paging_WahligHuang.pdf

[69] A. Karve, T. Kimbrel, G. Pacifici, M. Spreitzer, M. Steinder, M. Sviridenko, A. Tantawi, *Dynamic Placement for Clustered Web Applications*, in Proceedings of the 15th international conference on World Wide Web, Edinburgh, Scotland, 2006

[70] J. O. Kephart, D. M. Chess, *The Vision of Autonomic Computing*, Computer, January 2003

[71] L. Kleinrock, Queuing Systems, Volume I: Theory, Wiley Interscience, 1975

[72] L. Kleinrock, *Queueing Systems, Volume 2: Computer Applications*, John Wiley and Sons, Inc., 1976

[73] J. Jann, L. M. Browning, R. S. Burugula, Dynamic Reconfiguration: Basic Building Blocks for Autonomic Computing on IBM pSeries Servers, IBM Systems Journal, 42(1):29-37

[74] T. Lindholm, F. Yellin, The Java Virtual Machine Specification, second edition, Prentice Hall PTR

[75] S. Martello, P. Toth, *Knapsacks Problems: Algorithms and Computer Implementations,* John Wiley and Sons, 1990

[76] H. Matsumoto, *SCSI Support for Xen*, Xen Summit Spring 2007,
http://xen.org/files/xensummit_4/Xen
_Summit_8_Matsumoto.pdf

[77] M. Matsumoto, T. Nishimura, *Mersenne twister: a 623-dimensionally Equidistributed Uniform Pseudo-random Number Generato*r, ACM Transactions on Modeling and Computer Simulation, vol. 8 issue 1, January 1998

[78] D. A. Menasce, V. A. F. Almeida, L. W. Dowdy, *Capacity Planning and Performance Modeling: from mainframes to client-server systems*, Prentice Hall, 1994.

[79] D. A. Menascé, M. N. Bennani, *Autonomic Virtualized Environments*, in Proceedings of the International Conference on Autonomic and Autonomous Systems, ICAS'06, 2006

[80] D. A. Menascè, M. N. Bennani, *On the Use of Performance Models to Design Self-Managing Computer Systems*, in Proceedings of 2003 Computer Measurement Group Conference, 2003, Dallas, TX, USA

[81] D. A. Menascé, *Virtualization: Concepts, Applications, and Performance Modeling*, in Proceedings of 2005 Computer Measurement Group Conference, Dec. 4-9, 2005, Orlando, FL, USA

[82] M. Nelson, B. Lim, G. Hutchins, *Fast Transparent Migration for Virtual Machines*, in Proceedings of the annual conference on USENIX Annual Technical Conference, Anaheim, CA, USA, 2005

[83] P. Padala, X. Zhu, Z. Wang, S. Singhal, K. G. Shin, *Performance Evaluation of Virtualization Technologies for Server Consolidation*, HP Tech Report HPL-2007-59

[84] P. Padala, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, K. Salem, *Adaptive Control of Virtualized*

Resources in Utility Computing Environments, in ACM SIGOPS Operating Systems Review, vol. 41 issue 3, June 2007

[85] D. Pisinger, A Minimal Algorithm for the Multiple-Choice Knapsack Problem, European Journal of Operational Research, vol. 83 issue 2, Elsevier, 1995

[86] G. J. Popek, R. P. Goldberg, *Formal Requirements for Virtualizable Third Generation Architectures*, Communications of the ACM, Volume 17 Number 1, July 1974

[87] B. Quetier, V. Neri, F. Cappello, *Scalability Comparison of Four Host Virtualization Tools*, in Journal of Grid Computing, 2006

[88] B. Quetier, V. Neri, F. Cappello, *Selecting a Virtualization System for Grid/P2P Large Scale Emulation*, in Proceedings of EXPGRID (HPDC-15's Workshop), Paris, France, 2006

[89] S. Ranjan, J. Rolia, H. Fu, E. Knightly, *QoS-Driven Server Migration for Internet Data Centers*, in Proceedings of Tenth International IEEE Workshop on Quality of Service, 2002.

[90] M. Reiser, S. S. Lavenberg, *Mean-Value Analysis of Closed Multichain Queueing Networks*, Journal of the ACM, vol. 27 issue 2, April 190.

[91] S. Rixner, *Network Virtualization: Breaking the Performance Barrier*, ACM Queue January/February 2008

[92] J.S. Robin, C. E. Irvine, *Analysis of the Intel Pentium's ability to support a secure virtual machine monitor*, in Proceedings of the 9th USENIX Security Symposium, Denver, CO, USA, August 2000

[93] B. Ryu, A. Elwalid, *The importance of long-range dependence of VBR video traffic in ATM traffic engineering: myths and realities*, in Proceedings of the ACM SIGCOMM'96, Palo Alto, CA, USA

[94] S. Shelford, M. M. Akbar, E. G. Manning, G. C. Shoia, *Distributed Optimal Admission Controllers for Service Level Agreements in Interconnected Networks*, in Proceedings of the 21st IASTED International Conference on Applied Informatics, Innsbruck, Austria, 2003

[95] S. Soltesz, H. Potzl, M. E. Fiuczynski, A. Bavier, L. Peterson, *Container-Based Operating System Virtualization: a Scalable, High-performance Alternative to Hypervisors*,

in Proceedings of the EuroSys 2007 Conference, Lisbon, Portugal, 2007

[96] S. J. Vaughan-Nichols, *New Approach to Virtualization is Lightweight*, Computer, IEEE Computer Society, Vol. 39 Number 11, 2006

[97] F. Travostino, P. Daspit, L. Gommans, C. Jog, C. de Laat, J. Mambretti, I. Monga, B. van Oudenaarde, S. Raghunath, P. Yonghui Wang, *Seamless Live Migration of Virtual Machines over the MAN/WAN*, Future Generation Computer Systems, Volume 22 Issue 8, 2006

[98] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, A. Tantawi, *An Analytical Model for Multi-tier Internet Services and Its Applications*, in Proceedings of SIGMETRICS'05, 2005, Canada

[99] D. Vilella, P. Pradhan, D. Rubenstein, *Provisioning Servers in the Application Tier for E-Commerce Systems*, ACM Transactions on Internet Technologies, vol. 7 No. 1, 2007

[100] E. Wahlig, W. Huang, *AMD Barcelona and Nested Paging Support in Xen*, Xen Summit Spring 2007, http://xen.xensource.com/files/xensummit_4/

2007XenSummit-AMD-Barcelona_Nested
_Paging_WahligHuang.pdf

[101] X. Y. Wang, D. Lan, G. Wang, X. Fang, M. Ye, Y. Chen, Q. Wang, A*ppliance-based Autonomic Provisioning Framework for Virtualized Outsourcing Data Center*, in Proceedings of the Fourth International Conference on Autonomic Computing (ICAC'07)

[102] A. Whitaker, M. Shaw, S.D. Gribble, *Denali: Lightweight Virtual Machines for Distributed and Networked Applications*, Technical Report 02-02-01, University of Washington, 2002

[103] S. R. White, J. E. Hanson, I. Whalley, D. M. Chess, J. O. Kephart, *An Architectural Approach to Autonomic Computing*, in Proceedings of the International Conferecne on Autonomic Computing (ICAC'04), 2004

[104] T. Yu, K. Lin, Service Selection Algorithms for Web Services with End-to-End QoS Constraints, Information Systems and E-Business Management, vol. 3 n. 2, Spinger Verlag, 2005

[105] Q. Zhang, L. Cherkasova, E. Smirni, *A Regression-based Analytic Model for Dynamic Resource Provision-*

ing of Multi-Tier Applications, in Proceedings of the Fourth International Conference on Autonomic Computing (ICAC'07), 2007