



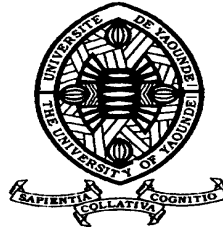
23 23



UNIVERSITE DE YAOUNDE I

ECOLE NATIONALE SUPERIEURE
POLYTECHNIQUE

DEPARTEMENT DE GENIE
INFORMATIQUE



UNIVERSITY OF YAOUNDE I

NATIONAL ADVANCED SCHOOL
OF ENGINEERING

DEPARTMENT OF COMPUTER
ENGINEERING

OPTIMIZED RESOURCE ALLOCATION FOR THE PRIVILEGED DOMAIN IN SERVER VIRTUALIZATION

Applied to the domain 0 in the Xen virtualization system

End of course dissertation/Master of Engineering

Presented and defended by

MVONDO DJOB BARBE THYSTERE

In partial fulfilment of the requirements for the award of a:

Master of Engineering in Computer Science

Under the supervision of:

**DANIEL HAGIMONT, PROFESSOR, NATIONAL POLYTECHNIC INSTITUTE OF
TOULOUSE**

**ALAIN TCHANA, ASSOCIATE PROFESSOR, NATIONAL POLYTECHNIC
INSTITUTE OF TOULOUSE**

**BORIS TEABE, ENGINEER, TOULOUSE COMPUTER SCIENCE RESEARCH
INSTITUTE**

In front of the jury composed of:

President: **BENOÎT NDZANA, ASSOCIATE PROFESSOR, UNIVERSITY OF
YAOUNDE I**

Reporter: **ALAIN TCHANA, ASSOCIATE PROFESSOR, NATIONAL
POLYTECHNIC INSTITUTE OF TOULOUSE**

Examiner: **BERNABÉ BATCHAKUI, SENIOR LECTURER, UNIVERSITY
OF YAOUNDE I**

Academic year 2016-2017
Defended the 08th September 2017



OPTIMIZED RESOURCE ALLOCATION FOR THE
PRIVILEGED DOMAIN IN SERVER VIRTUALIZATION

Applied to the domain 0 in the Xen virtualization system

End of course dissertation/Master of Engineering

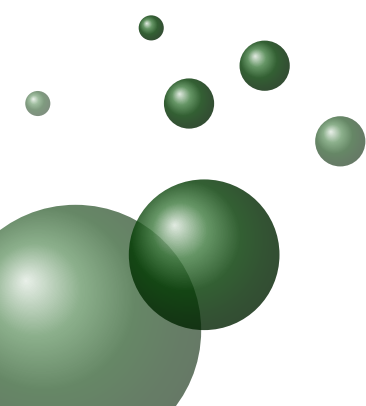
Presented and defended by

MVONDO DJOB BARBE THYSTERE

In partial fulfilment of the requirements for the award of a:

Master of Engineering in Computer Science

Academic year 2016-2017
Defended the 08 September 2017



DEDICATION



To my parents

ACKNOWLEDGEMENTS

This work is the culmination of many efforts and sacrifices and would never have been accomplished without the help and support of:

- **Benoît NDZANA**, Associate Professor at UY1, who has made us the honor of presiding over this jury;
- **Bernabé BATCHAKUI**, Senior Lecturer at UY1, for agreeing to examine this work and for his remarkable dedication to teaching;
- **Alain TCHANA** and **Daniel HAGIMONT**, respectively Associate Professor and Professor at INPT, members of the IRIT laboratory and my supervisors, for their guidance, their availability and the enormous contributions brought to the accomplishment of this work;
- **Boris TEABE**, PhD student in the IRIT laboratory and one of my supervisors, for his advice during the internship;
- **Michel DAYDÉ**, Professor at INPT and Director of the IRIT laboratory, for giving me the opportunity to do an internship in the structure he is responsible for;
- All the teaching staff of ENSPY, especially those of the Computer Science and Engineering Department for the skills I acquired during my curriculum at ENSPY;
- **Zoua MVONDO**, my brother and **Lingom MVONDO**, my sister for their support and kindness;
- **El Mehdi JAZOULI**, **Elleuch TAHA YASSINE**, **Fopa Léon CONSTATIN**, **Grégoire TODESCHI**, **Kevin JIOKENG**, **Lavoisier WAPET**, **Mathieu BACOU**, **Nicolas DURASEI**, **Vlad NITU**, my internship mates, for their help;
- **Joubou LERRICK**, **Kevin ESSELEBO**, **Metuno BLERIoT**, **Gaetan TIBATI**, my friends, for the joy I had with them during these past years;
- My classmates and friends of the 2017 promotion for their collaboration spirit;
- Every other person who contributed from near or afar to the elaboration of this masterpiece.

GLOSSARY

Benchmark In computer science, a benchmark is a test to measure the performance of a system to compare it to others. 3

Bus In computer science, a bus is a communication system that transfers data between components inside a computer, or between computers.. 9

Business recovery plan It defines the process of creating prevention and recovery systems to deal with potential threats to a company. 8

Data center It represents a physical site on which are grouped components of the information system of a company (central computers, servers, storage bays, network and telecommunications equipment, etc.) . vi, 1, 2, 10, 17, 19, 21, 27, 28

Device driver It is a computer program that operates or controls a particular type of device that is attached to a computer. 1, 11, 34

Interrupt An interrupt is a signal from a device attached to a computer or from a program within the computer that requires the operating system to stop and figure out what to do next. 12, 34

Kernel It is a computer program that is the core of a computer's operating system, with complete control over everything in the system. It is the first program loaded on start-up.. 11, 34, 39

Open source The open source designation, or "open source code", applies to software whose license meets criteria precisely established by the Open Source Initiative, that is, the possibilities of free redistribution, access to source code and the creation of derivative works.. 1, 6, 9, 34

ABSTRACT

Server virtualization offers the ability to slice large, underutilized physical servers into smaller, parallel virtual machines, enabling diverse applications to run in isolated environments on a shared hardware platform. A key motivation for applying server virtualization is to improve hardware resource utilization while maintaining a reasonable quality of service. However, such a goal cannot be achieved without efficient resource management. Though most physical resources, such as processor cores and I/O devices, are shared among virtual machines using time slicing and can be scheduled flexibly based on priority, allocating an appropriate amount of resources to the privileged virtual machine is more challenging. Indeed, the latter is responsible to carry out a set of tasks for the good execution of virtual machines. Different virtual machines generate varying load on the privileged one. Even a single virtual machine generates varying load during its execution. An optimal resource management strategy for the privileged virtual machine thus needs to dynamically adjust memory and processor allocation, and in case of non-symmetric multiprocessing architecture as non-uniform memory access, it should adjust the location of these resources. This work presents a design for the privileged virtual machine which ensures the latter possesses what it needs at any moment, by decomposing the latter in special groups called main container and secondary containers with new scheduling and memory allocation algorithms. We have implemented the proposed design for para-virtualized guests in the Xen hypervisor. Compared to the current Xen native implementation, we achieve, on average, 12% gain in performance over a set of tasks and provide a scalable architecture for the privileged virtual machine.

Keywords: Virtualization, Resource, Memory, Processor, Location.

RÉSUMÉ

La virtualisation des serveurs offre la possibilité de découper de gros serveurs physiques sous-utilisés en petites machines virtuelles parallèles, permettant à diverses applications de s'exécuter dans des environnements isolés sur une plate-forme matérielle partagée. Une motivation clé pour appliquer la virtualisation des serveurs est d'améliorer l'utilisation des ressources matérielles tout en maintenant une qualité de service raisonnable. Cependant, un tel objectif ne peut être atteint sans une gestion efficace des ressources. Bien que la plupart des ressources physiques, telles que les processeurs et les périphériques d'E/S, soient partagées entre les machines virtuelles à l'aide du découpage temporel et peuvent être programmées avec souplesse en fonction de la priorité, l'allocation d'une quantité appropriée de ressources à la machine virtuelle privilégiée s'avère plus délicate. En effet, ce dernier est chargé d'exécuter un ensemble de tâches pour le bon fonctionnement des machines virtuelles. Différentes machines virtuelles génèrent une charge variable sur la machine virtuelle privilégiée. Même une seule machine virtuelle génère une charge variable pendant son exécution. Une stratégie optimale de gestion des ressources pour la machine virtuelle privilégiée doit permettre d'ajuster dynamiquement la mémoire et l'allocation du processeur de ce dernier et dans le cas particulier d'une architecture multiprocesseur non symétrique, comme dans une architecture à accès mémoire non uniforme, permettre d'ajuster l'emplacement de ses ressources. Ce travail présente une approche pour l'allocation des ressources à la machine virtuelle privilégiée qui garantit que ce dernier possède ce dont il a besoin à tout moment, en le décomposant en des groupes spéciaux appelés conteneur principal et conteneurs secondaires avec de nouveaux algorithmes d'ordonnancement et d'allocation de mémoire. Nous avons mis en œuvre l'approche proposée pour les hôtes para-virtualisés dans l'hyperviseur Xen. Par rapport à l'implémentation native de Xen, nous réalisons en moyenne 12% de gain de performance sur un ensemble de tâches et fournissons une architecture évolutive pour la machine virtuelle privilégiée.

Mot clés: Virtualisation, Mémoire, Ressource, Processeur, Emplacement.

LIST OF TABLES

TABLE	Page
1.1 Comparison of virtualization tools based on different factors	9
3.1 Dom0 processes organization	27
3.2 Data center minimalistic architecture consumption	28


LIST OF FIGURES

FIGURE	Page
0.1 Simplified Xen Architecture	2
1.1 Full virtualization architecture	5
1.2 OS level virtualization architecture	6
1.3 Paravirtualization architecture	7
1.4 An example of a numa architecture	10
1.5 Split driver model	11
1.6 Network I/O transmit mechanism between front-end and back-end	13
1.7 Zero copy mechanism	14
2.1 Experimental setup to show the variation of resources required by the tasks from the second group.	18
2.2 Experimental setup to show the impact of dom0's resources on user's application. . .	19
2.3 Experimental setup to show the impact of dom0's resources on management tasks. . .	20
2.4 Experimental setup to show the impact of dom0's resources on the data center management services.	20
2.5 Experimental setup to show the impact of dom0's resources location on domU IO applications.	22
2.6 Experimental setup to show the impact of dom0's resources location on domU migration.	22
2.7 Experimental results, first group	23
2.8 Experimental results, second group	23
2.9 Experimental results, third group	24
3.1 Our dom0 design	26
3.2 OpenStack Minimalistic set-up consumption	28
3.3 Minimal OpenStack architecture set-up	29
3.4 Scheduling Algorithm for secondary dom0 tasks	32
4.1 First group results	38
4.2 Second group results	38

TABLE OF CONTENTS

Dedication	i
Acknowledgements	ii
Abbreviations	iii
Glossary	v
Abstract	vi
Résumé	vii
List of Tables	viii
List of Figures	ix
Table of Contents	xi
Introduction	1
1 Background	4
1.1 Server virtualization	5
1.1.1 Types of server virtualization systems	5
1.1.2 Advantages of server virtualization	7
1.1.3 Some server virtualization systems	9
1.2 NUMA architecture	9
1.3 I/O mechanism in Xen	11
1.3.1 Network I/O mechanism	12
1.3.2 Packet reception	12
1.3.3 Block I/O mechanism	14
2 Problem statement and assessment	16

TABLE OF CONTENTS



2.1	(Q_1) Dom0 sizing	17
2.1.1	Consequences of an undersized dom0	18
2.1.2	Consequence of an oversized dom0	21
2.2	(Q_2) Dom0 location and Review	21
2.3	Review	24
3	Contribution	25
3.1	Container composition	27
3.2	Resource allocation to the main container	27
3.3	Resource allocation to secondary containers	29
3.3.1	Memory allocation	29
3.3.2	Processor allocation	30
3.4	Our approach advantages	31
4	Implementation and Evaluation	33
4.1	Implementation	34
4.1.1	The hypervisor	34
4.1.2	The guest OS	34
4.1.3	Development tool	34
4.2	Evaluation	35
4.2.1	Benchmark 1: Load generated on the dom0 due to domUs activities	35
4.2.2	Benchmark 2: Migration time under high load conditions	36
4.2.3	Benchmark 3: I/O performance as a result of locality	36
4.2.4	Benchmark 4: Migration time as a result to locality	37
4.3	Review	37
	Conclusion	39
	Annex	40



INTRODUCTION

CONTEXT

Server virtualization allows the multiplexing of hardware resources on a physical computer for virtual machines¹ (VM) that can then be purchased by companies in order to achieve power saving costs, machine acquirement savings, avoid licensing costs and servers maintenance. The server virtualization market is gaining demand as organizations seek to improve their business growth by shifting from on-premise to cloud based. Cloud providers such as Amazon, Microsoft, IBM, Google must ensure the isolation of virtual machines between them and make sure that a virtual machine possesses the resources for which a customer paid for the SLA to be met. To achieve this, they use server virtualization systems commonly known as hypervisors or virtual machine monitors (VMMs). Several hypervisors exist but our work relies on the Xen virtualization system firstly due to its open source nature, secondly because it is the hypervisor used by the Amazon Web Service [?], who is the world leader cloud infrastructure provider [?].

Xen is a server virtualization system that replaces the traditional operating system and runs directly on the hardware. It runs virtual machines in environments known as **domains**, which encapsulate a complete running virtual environment [?]. For the sake of simplicity and maintainability, the original operating system is embedded in a particular domain called **dom0** which stands for **domain 0** (the other domains are called **domU** meaning **domain unprivileged**). The dom0 embeds administration tools for managing virtual machines (monitoring and configuration) in the overall data center. The most obvious task performed by the dom0 is to handle devices on behalf of the domU guests. It includes **device drivers** which provide access to the hardware as shown on Figure 0.1. The dom0 is therefore granted privileged access rights and can be seen as an extension of the hypervisor.

¹A virtual machine is the billing unit in the cloud. One consumes virtual machines possessing computing resources

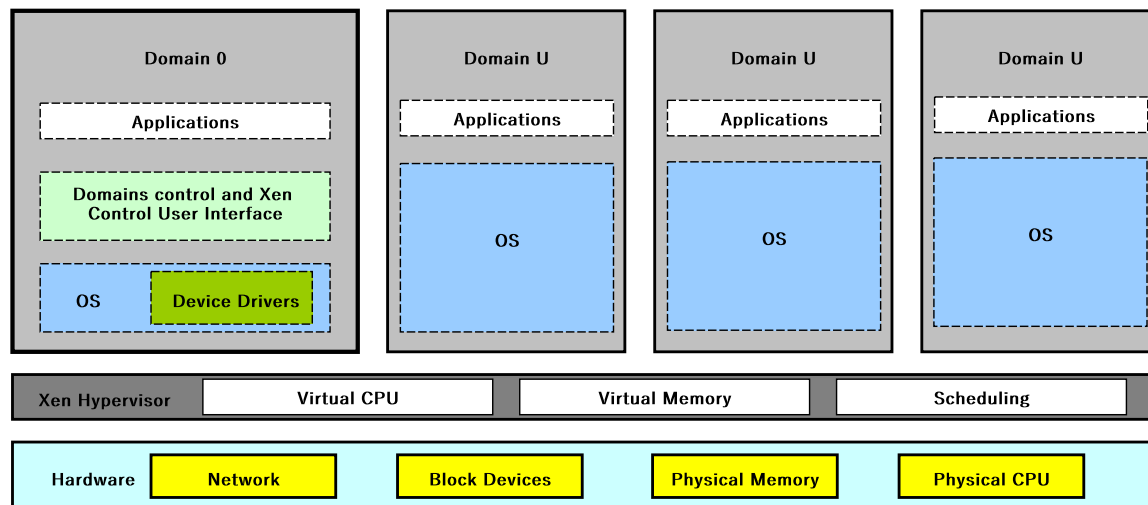


Figure 0.1: Simplified Xen Architecture

PROBLEMATIC

In order to perform all these tasks, the dom0 requires computing resources. As computing resources, we refer to CPU and memory since the other resources such as network and disk are not significantly involved in these tasks. Resource allocation to the dom0 is a serious problem, especially in NUMA architectures which are commonly used in today's data centers [?]. Concretely, cloud providers are faced with this essential question:

Questions

How should resources be allocated to the dom0 and organised in aNUMA architecture ?

OBJECTIVES

The main goal of this work is to establish a **resource allocation strategy for the dom0** that will take into consideration the two fundamental questions mentioned above. Our resource allocation strategy should enable:

1. the dom0 to have exactly the amount of resources it needs at every moment,
2. the dom0 processes that work on behalf of a virtual machine use the resources of this virtual machine and
3. those processes must be executed as close as possible to the virtual machine for which they work.





To achieve this, we explored the issues associated with these two questions, analyzed the limitation of previous solutions addressed to this problem and compared our resulting model to the existing solutions on a series of benchmarks well thought out for the problem.

ROAD MAP

The rest of the document is structured as follows:

- **Background** in which we enrich our vocabulary and present some mechanisms in the Xen architecture for a better understanding of the work and what follows;
- **Problem statement and assessment** where we discuss about the questions raised in the problematic and prove it is relevant;
- **Contribution** in which we describe our model, with a view on the overall architecture and a detailed presentation of the different model components;
- **Implementation and Evaluation** where we describe how our model has been developed, the different software tools used and how our solution performs compared to the existing solutions in order to verify if our solution meets the objectives defined.



BACKGROUND

In this chapter, we present basic concepts such as server virtualization, the types of server virtualization systems, NUMA architectures and I/O mechanism in virtualized environments such as Xen for a better apprehension of the work done. The plan of this chapter is as follows:

Contents

1.1	Server virtualization	5
1.1.1	Types of server virtualization systems	5
1.1.2	Advantages of server virtualization	7
1.1.3	Some server virtualization systems	9
1.2	NUMA architecture	9
1.3	I/O mechanism in Xen	11
1.3.1	Network I/O mechanism	12
1.3.2	Packet reception	12
1.3.3	Block I/O mechanism	14



1.1 Server virtualization

Most servers use less than 15% of their resources [?]. Server virtualization addresses these inefficiencies by allowing multiple operating systems to run on a single physical server as virtual machines, each with access to the underlying server's computing resources. To achieve server virtualization, we need a server virtualization system (hypervisor). There are mainly 4 types of server virtualization systems which are: **full virtualization**, **para-virtualization**, **OS level virtualization** and **hardware assisted virtualization**. Depending on the needs, one type may be better than another. In the next section, we discuss the different types of server virtualization systems and give some examples.

1.1.1 Types of server virtualization systems

1.1.1.1 Full virtualization

Here, the hypervisor is a software that runs on top of the host¹ OS as shown on Figure 1.1. The latter virtualizes/emulates the hardware for the guest OS (virtual machine), and is believed to communicate directly with the hardware. This solution is very similar to an **emulator**, and sometimes even mistaken for the latter. However, the CPU, the RAM, as well as the storage memory, are directly accessible to the virtual machines, while on an emulator the CPU is emulated, and the performance is considerably reduced compared to virtualization.

This type of server virtualization is offered by software such as **Virtualbox**, **VMWare Workstation**. Hypervisors enabling this type of server virtualization are said to be of **type 2**.

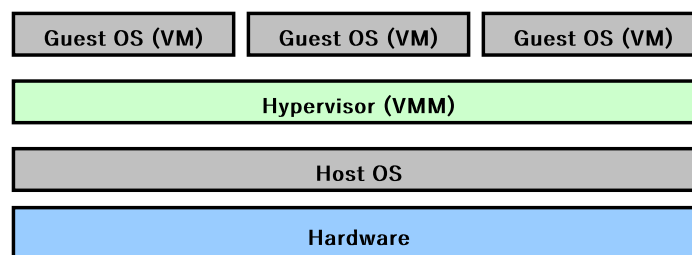


Figure 1.1: Full virtualization architecture

1.1.1.2 OS level virtualization

Here, the hypervisor is called an **isolator**. An isolator is software that isolates the execution of applications in **contexts**, or **execution areas** as shown on Figure 1.2 . The isolator thus makes it possible to run the same application several times in a multi-instance mode (several instances

¹Here we refer to the physical machine





of execution) even if it was not designed for that. This solution is very efficient, because of the **small overhead**, but the virtualized environments are not completely isolated.

There is a visible performance. However, we can not really talk about virtualization of operating systems. Only connected to Linux systems, isolators are actually made up of several elements and can take many forms. This type of virtualization is offered by software as **Linux-Vserver**, **Chroot**, **BSD Jail**, **OpenVZ** and **LXC (Linux Container)**.

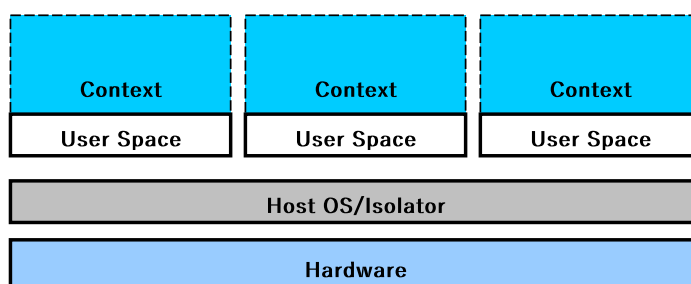


Figure 1.2: OS level virtualization architecture

1.1.1.3 Hardware assisted virtualization

This is a particular case of server virtualization where the hardware is designed to ease virtualization. Here the virtual machines are called **Hardware Virtual Machines (HVM)**. The CPU designers such as **Intel** and **AMD** modify the hardware to enable technologies such as **Intel VT** and **AMD-V** which enables material assisted virtualization. Hypervisors such as **VMWare vCenter**, **Xen** and **KVM** supports material assisted virtualization.

1.1.1.4 Paravirtualization

Here the hypervisor replaces the host OS and acts as an intermediate between the guest and the hardware. The host OS is considered a privileged virtual machine and used by the hypervisor to complete a set of tasks as shown in Figure 1.3. However, this type of server virtualization requires modification of the guest OS to be virtualized for it to support **system calls** from the hypervisor. Paravirtualization offers better performance than the other types of server virtualization even though the modification of the guest OS is not always possible. For example, it is not possible to modify the **Windows** kernel for its designer does not allow it since it is not open source.

Hypervisors enabling this type of virtualization are said to be **type 1**. Some examples of these hypervisors are **Xen**, **VMWare ESX**.



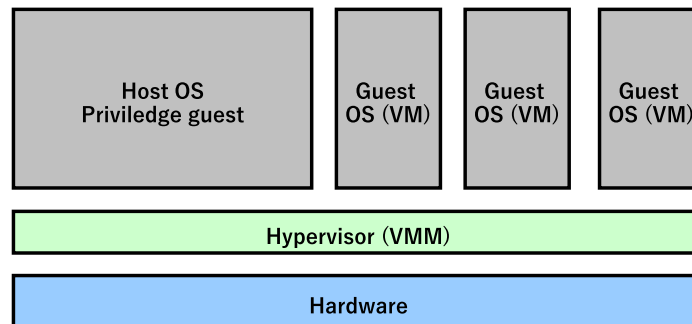


Figure 1.3: Paravirtualization architecture

1.1.2 Advantages of server virtualization

Server virtualization is not just for big companies. SMEs can also take advantage of this technology, the advantages of which are presented here.

Reduced physical servers. The first benefit of server virtualization is that since many virtual machines can run on a single physical server, the number of servers to buy and maintain will be reduced. With a traditional IT infrastructure, many servers are oversized in order to cope with possible peak loads. Combining multiple virtual machines in one place, a virtualized server is better exploited. The total number of machines required for the smooth functioning of the IS can thus be revised downwards.

Better availability. Server virtualization solutions now allow the migration of virtual machines which is the process to move a virtual machine from one physical server to another without even needing to stop it [? ?]. This feature is a key element in improving the availability of services. The use of two physical servers thus makes it possible to easily double a virtualized infrastructure (redundancy). In case of a failure of one of the two servers, the virtual machines will be automatically moved to the second.

Better performance. Another advantage of hot-migrating virtual machines between physical servers is that they can be used to distribute the workload between servers [? ?]. When a virtual machine rises to extreme load, others will be able to move to a less demanding physical server. Critical tasks will also work within a virtual machine that has more virtual CPU, virtual RAM and virtual hard disk cores than the others. It is thus possible to modulate the size of the virtual machines according to the tasks they will have to perform.

Security enhanced. In a traditional VSE/SME IT infrastructure, ERP, file sharing, the mailing system, or even the web server, all run on the same server. However, if the mailing system





is infected with a malware, all applications hosted on the machine are endangered. Server virtualization can be used to separate the different tasks of a physical server into many separate virtual machines, which will then be isolated from each other, thus dividing the services.

Anti-obsolescence warranty. A part of an IS, for example, an ERP, is likely to work on a dedicated server, due to a specific configuration. Ensuring the renewal of this dedicated server, which is used by only one application, is often not very profitable. P2V tools make it possible to transform most physical servers into virtual machines. Once this is done, the virtualized ERP server will be able to switch from an end-of-life machine to a new server by simply migrating the virtual machines.

Gain on licensing costs. One operating system license per server is required. However, virtualization sometimes allows you to take advantage of license packs covering the OS of the physical server and its virtual machines. For existing virtual machines, it will not be necessary to repay a license when the physical server running them is changed. A plus, which also applies to applications running within these virtual machines.

Simplified backups. With a virtualized infrastructure, the physical server is the only one physically present in the engine room. Virtual machines are pure software. This aspect greatly simplifies data backup operations. It is indeed possible to directly perform a backup of the contents of the virtual hard disk of a virtual machine. And even during its operation, by creating a snapshot of the virtual machine and its data. In case of a problem, this snapshot will restart the virtual machine in a previous state.

Business recovery plan easier to manage [?]. Virtualization can simplify the business recovery plan, making it easy to implement complex recovery plans. For example, launching a database server before the ERP server that accesses it.

Test without paying. Virtualization allows you to create a blank virtual machine in minutes. As long as your physical server does not display full, it will be possible to add new virtual machines to manage. Developers or system administrators will be able to exploit this feature to try new services without spending a single coin. Where a test server was previously needed, a simple virtual machine on an existing server of the company is enough today.

A stepping stone to the private cloud [?]. Server virtualization allows you to deploy new services within your computer system in the form of virtual machines. But also to dimension these virtual machines according to the criticality and the expected use. From there, why not offer a catalog of services and sizes of virtual machines to your business teams? And turn your IT into a service center.





1.1.3 Some server virtualization systems

In the table 1.1, we compare some server virtualization systems based on their characteristics.

Table 1.1: Comparison of virtualization tools based on different factors

Virtualization tools	License	Mode supported
Bosch	Open source	OS level
KVM	Open source	Full
LXC	Open source	OS level
Microsoft Virtual PC	Commercial	Full
Parallels	Commercial	Full
QEMU	Open source	Full
VMWare Vcenter	Commercial	Full
VMWare Workstation	Free (not open source)	Full
VNUML (Virtual Network User Model Linux)	Open source	Full
VServer	Open source	Para
Xen	Open source	Para

1.2 NUMA architecture

When using a server virtualization tool on a physical machine, for better configurations, it is essential to understand and master its architecture. Here we will try to give an outline of NUMA architectures and present its impact on the performance of applications.

For the past decade, processor clock speed has increased dramatically. A multi-gigahertz CPU, however, needs to be supplied with a large amount of memory bandwidth to use its processing power effectively. Even a single CPU running a memory-intensive workload, such as a scientific computing application, can be constrained by memory bandwidth. This problem is amplified on SMP systems, where many processors must compete for bandwidth on the same system bus. Some high-end systems often try to solve this problem by building a high-speed data bus. However, such a solution is expensive and limited in scalability.

NUMA is an alternative approach that links several small, cost-effective nodes using a high-performance connection [?]. Each **node** contains processors² and memory, much like a small SMP system. However, an advanced memory controller allows a node to use memory on all other nodes, creating a single system image as shown on Figure 1.4. When a processor accesses memory

²The group of processors in a node is usually called **socket**



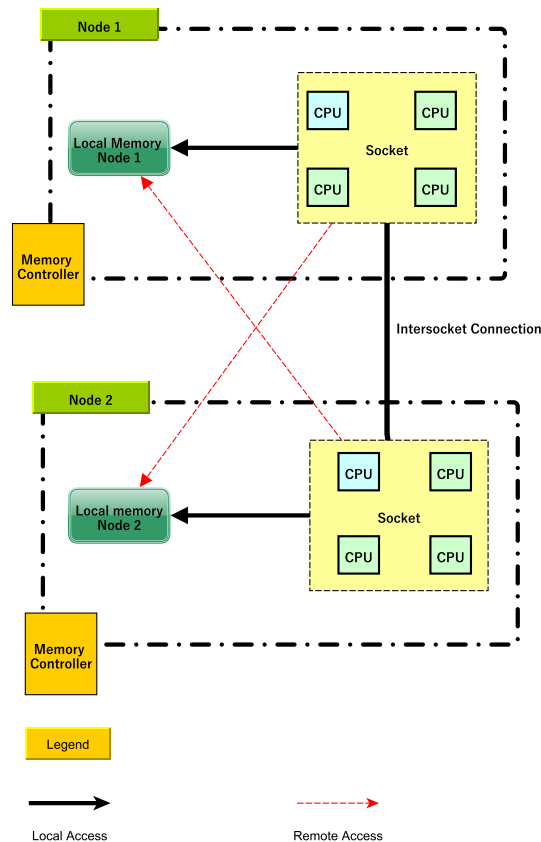


Figure 1.4: An example of a numa architecture

that does not lie within its own node (remote memory), the data must be transferred over the NUMA connection, which is slower than accessing local memory. Memory access times are not uniform and depend on the location of the memory and the node from which it is accessed, as the technology's name implies.

A lot of work was done to optimize existing algorithms to take into account NUMA [? ? ?], this work led to many changes in existing OS to take into account NUMA. NUMA is relevant to multiple processors and means that memory can be accessed *quicker if it is closer*. This means that memory is commonly **partitioned** at the hardware level in order to provide each processor in a multi-CPU system with its own memory. The idea is to avoid an argument when processors attempt to access the same memory. This is a good thing and means that NUMA has the potential to be more scalable than a **UMA** (multiple sockets share the same bus) design, particularly when it comes to environments with a large number of logical cores.

Now that we have a good idea of what is concretely server virtualization, and the predominant architecture in data centers, let us present how our server virtualization system handle I/O tasks

for the virtual machines to help us better understand the role played by the privileged domain (here the dom0) in these crucial tasks.

1.3 I/O mechanism in Xen

I/O tasks are handled by physical devices on a native OS. There are two major types of I/O activities, **block and network I/O** which use the block devices and network card by the means of device drivers. How the virtualization system Xen manages virtual devices will be the point of interest in this section.

Virtual devices under Xen are provided by a **split device driver model**. The illusion of the virtual device is provided by two co-operating drivers: the **front-end**, which runs in the unprivileged domain and the **back-end**, which runs in the privileged domain (here the domain0) as shown on Figure 1.5. The front-end driver appears to the unprivileged guest as if it was a real device, for instance, a block or network device. It receives I/O requests from its kernel, as usual. However, since it does not have access to the physical hardware of the system it must then issue requests to the back-end.

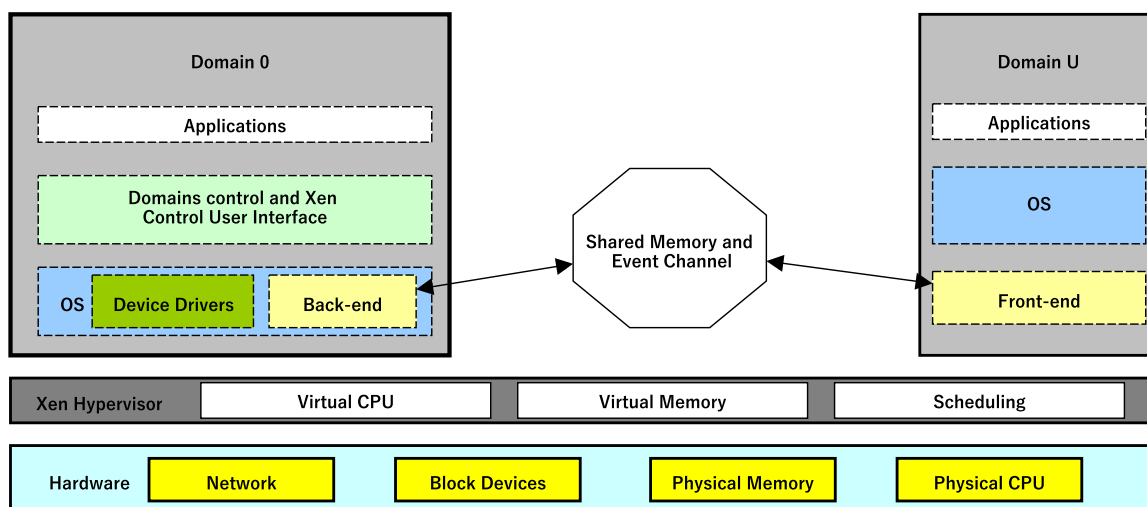


Figure 1.5: Split driver model

The backend driver is responsible for receiving these I/O requests, **verifying** that they are safe and then issuing them to the **real device** hardware. The back-end driver appears to its kernel as a normal user of in-kernel I/O functionality. When the I/O completes the back-end notifies the front-end that the data is ready for use, the front-end is then able to report I/O completion to its own kernel. Split drivers exchange requests and responses in **shared memory**, with an **event channel** for asynchronous notifications of activity [?]. When the front-end driver comes



up, it uses the **Xenstore** to set up a shared memory frame and an interdomain event channel for communications with the backend. Once this connection is established, the two can communicate directly by placing requests/responses into shared memory and then sending notifications on the event channel. In the sections below, we will detail what happens in the case of block I/O and network I/O.

1.3.1 Network I/O mechanism

From the point of view of the back-end domain itself, the network back-end driver consists of a number of ethernet devices. Each of these has a logical direct connection to a virtual network device in another domain. This allows the back-end domain to route, bridge, firewall, etc the traffic to/from the other domains using normal operating system mechanisms [?]. Corresponding to each virtual interface in a guest domain, a back-end interface is created in the driver domain, which acts as the proxy for that virtual interface in the driver domain (dom0). Each virtual interface uses two **descriptor rings** (two shared memory rings), one for transmitting, the other for receiving. Each descriptor identifies a block of contiguous machine memory allocated to the domain. Each back-end interface has a number of **queues** to handle requests from a virtual interface.

1.3.1.1 Packet transmission

As shown on Figure 1.6, the transmit ring carries packets to transmit from the guest to the back-end domain. By doing so,

- it writes/copies the packet in the transmit ring (which is a shared memory between the driver domain and the virtual guest),
- then generates a virtual interrupt request (VIRQ) to notify the driver domain (dom0)
- and the return path of the transmit ring carries messages indicating that the contents have been physically transmitted and the backend no longer requires the associated memory pages.

Each transmit request is followed by a transmit response at some later date. This is part of the shared-memory communication protocol and allows the guest to (potentially) retrieve internal structures related to the request.

1.3.2 Packet reception

To receive packets, the guest places descriptors of unused pages on the receive ring. The back-end will return received packets by exchanging these pages in the domain's memory with new pages containing the received data and passing back descriptors regarding the new packets on the ring.



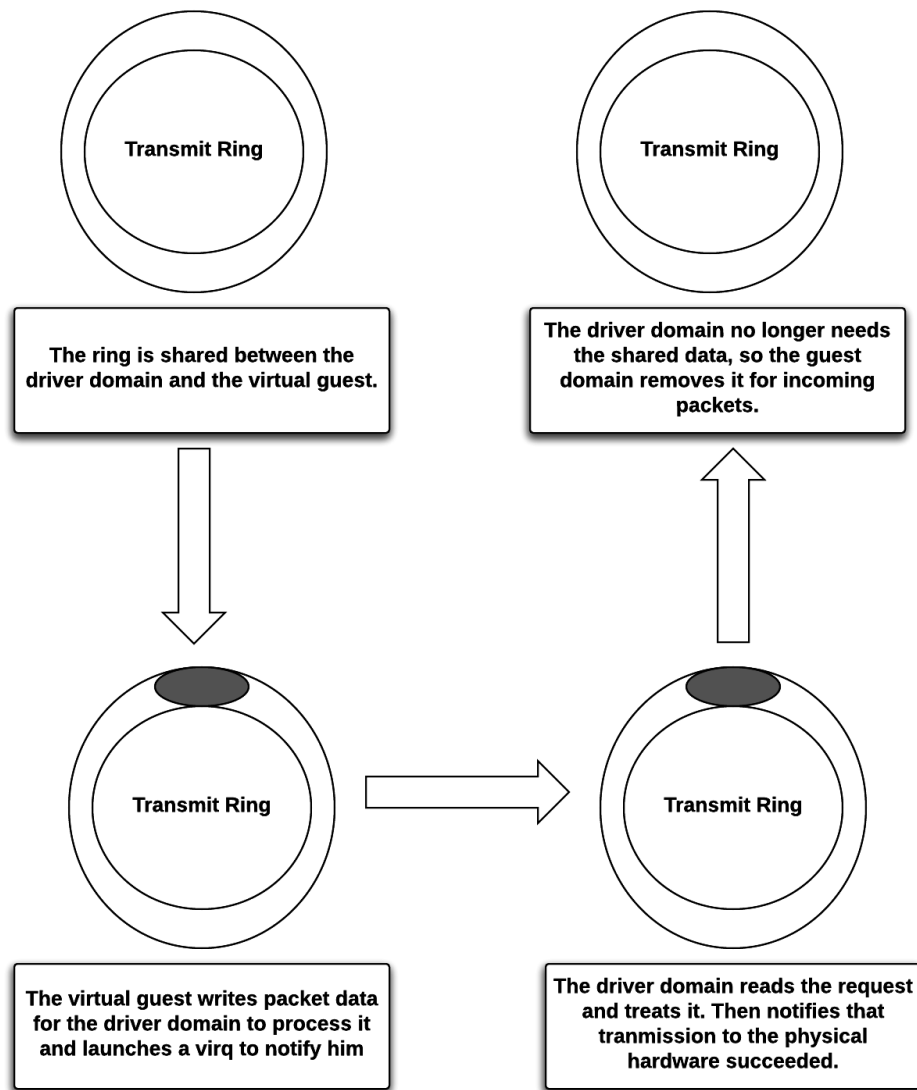


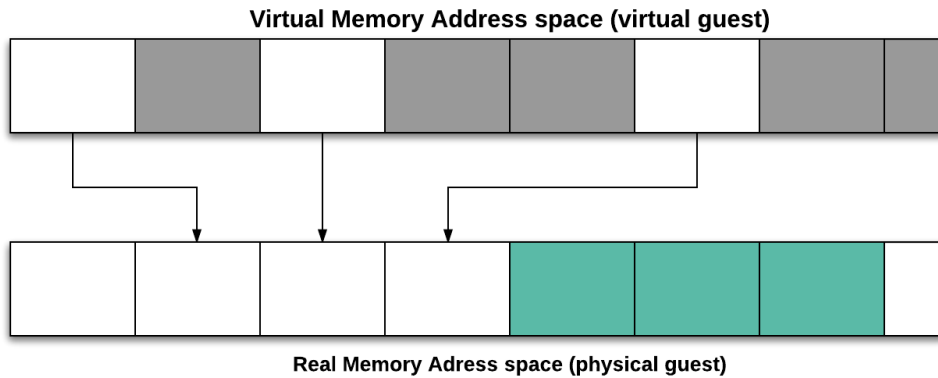
Figure 1.6: Network I/O transmit mechanism between front-end and back-end

This zero-copy approach as shown on Figure 1.7, allows the back-end to maintain a pool of free pages to receive packets into, and then deliver them to appropriate domains after examining their headers. Receive requests must be queued by the front-end, accompanied by a donation of page-frames to the back-end. The back-end transfers page frames full of data back to the guest. Receive response descriptors are queued for each received frame.

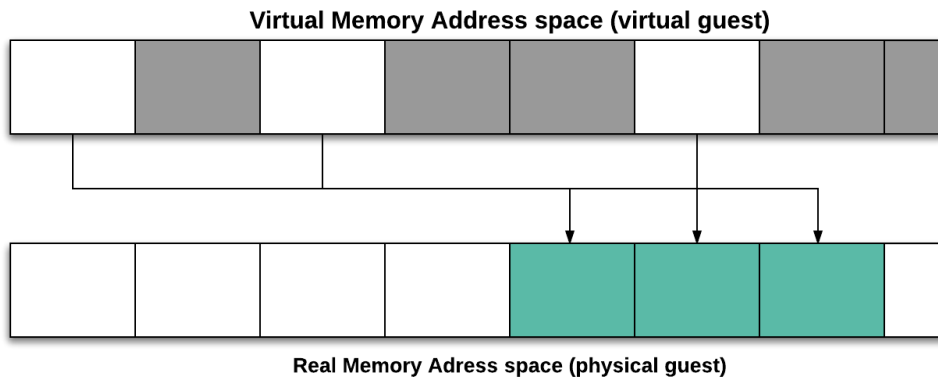
Guests are expected to return memory to the hypervisor in order to use the network interface. They must do this or they will exceed their maximum memory reservation and will not be able to receive incoming frame transfers. When necessary, the back-end is able to replenish its pool of free network buffers by claiming some of this free memory from the hypervisor.



- (1) The virtual guest grants a set of free pages to be remapped to the real pages containing the packet destined to him.



- (2) The exchange now takes place, the virtual pages granted are remapped to the pages containing the packet data and the initial mapping is given to the driver domain to handle incoming requests.



- (3) Now the virtual guest can access the data. But in case the new real address is on a different physical node as the virtual guest, the virtual guest will have to access remote content.

Legend



Figure 1.7: Zero copy mechanism

1.3.3 Block I/O mechanism

All guest OS disk access goes through the **virtual block device** (VBD) interface. This interface allows domains access to portions of block storage devices visible to the block back-end device. The VBD interface is a split driver, similar to the network interface described above. A single shared memory ring is used between the front-end and back-end drivers for each virtual device, across which I/O requests and responses are sent. Any block device accessible to the back-end





domain can be exported as a VBD. Each VBD is mapped to a device node in the guest, specified in the guest's startup configuration.

The per-(virtual)-device ring between the guest and the block back-end supports two messages:

- **READ:** Read data from the specified block device. The front end identifies the device and location to read from and attaches pages for the data to be copied to (typically via DMA from the device). The back-end acknowledges completed read requests as they finish.
- **WRITE:** Write data to the specified block device. This functions essentially as READ, except that the data moves to the device instead of from it.

In the next chapter, we will discuss about the questions raised in the problematic and its assessment.



PROBLEM STATEMENT AND ASSESSMENT

This chapter detail the issues associated with the two questions described in the introduction and present its assessment. The plan of this chapter is as follows:

Contents

2.1	(Q_1) Dom0 sizing	17
2.1.1	Consequences of an undersized dom0	18
2.1.2	Consequence of an oversized dom0	21
2.2	(Q_2) Dom0 location and Review	21
2.3	Review	24



The privileged domain, the dom0 in Xen, is a crucial element for a good yield of applications running inside virtual machines and for the data center management tasks using its resources. Here we detail the issues associated with the problematic in order to prove the relevance of this work. Let's recall the questions raised in the problematic :

- (Q₁) how should resources be allocated to the dom0?
- (Q₂) how should these resources be organised in a NUMA architecture ?

2.1 (Q₁) Dom0 sizing

The commonly used strategy is to statically allocate a fixed amount of resources to the dom0. A static resource allocation strategy consists in the allocation to the dom0, at server startup, of a given amount of resource which stays unchanged during its life time. Most data centers follow this strategy, including the research works that we studied [?????]. For instance, we can observe such a static dom0 allocation policy in Amazon EC2 [?], an allocated *c3 xlarge* is supposed to receive 20 processors, but only 16 are available for virtual machine instances. Other processors are reserved for the dom0.

More generally;

There isn't any defined method to estimate the amount of resources which is required for the dom0 to perform correctly.

This allocation is totally arbitrary. Server virtualization systems providers do not provide any recommendation regarding this issue. The only recommendation that we found for the dom0 resource allocation comes from **Oracle** [?], which proposes a dom0 memory allocation based on the rule given by the following rule. **E stands for whole part.**

$$\text{dom0_mem} = 502 + E(\text{physical_mem} \times 0.0205)$$

The problem with such a static allocation is that the amount of resources required by the dom0 is not constant, as it depends on the **dom0 workload**. The tasks executed by the dom0 can be organized into two groups:

- those that do not depend on the domUs (these tasks are related to the management of the data center) and
- those that depend on the domUs (most of these tasks are involved when the domUs are requesting device drivers).



The amount of resources required by tasks from the second group is **variable** as it depends on the activity of the domUs. To demonstrate it, we carried out an experiment (whose experimental setup is shown on Figure 2.1) consisting of measuring the variation of the dom0 CPU load according to a number of domU executing an **IOZONE** [?] benchmark.

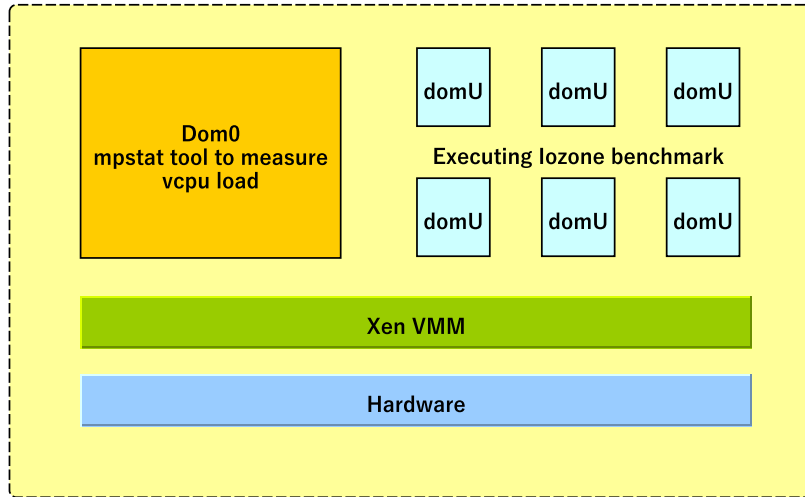


Figure 2.1: Experimental setup to show the variation of resources required by the tasks from the second group.

As shown on Figure ??, we observe that the dom0's load heavily depends on the domU. The consequence is that a static allocation can lead to two situations: either **the dom0 lacks resources**, or **the resources are overbooked by the dom0**. These two situations can be harmful both for the cloud provider and for the user's applications.

2.1.1 Consequences of an undersized dom0

A lack of resource in the dom0 can have significant impacts both for applications executed in the domUs and for administrative services executed in the dom0.

Impact on user's applications. To prove it, we conducted an experiment (whose setup is shown on figure 2.2), where we varied the number of co-located domU guests which executes I/O intensive workloads with a domU guest executing a Wordpress application. The dom0 was allocated two processors while each domU was allocated one processor. In order to prevent any contention (e.g QPI link contention), all processors (from dom0 or domUs) were allocated on the same socket. The low-level cache is not a limiting factor for I/O intensive applications [?].

Figure 2.8a presents the dom0 CPU load and the domU web application performance. The first observation is that the dom0 load varies according to the I/O activity of Domus. This variation is explained by the fact the dom0 embeds the back-end (from the split driver) and the driver



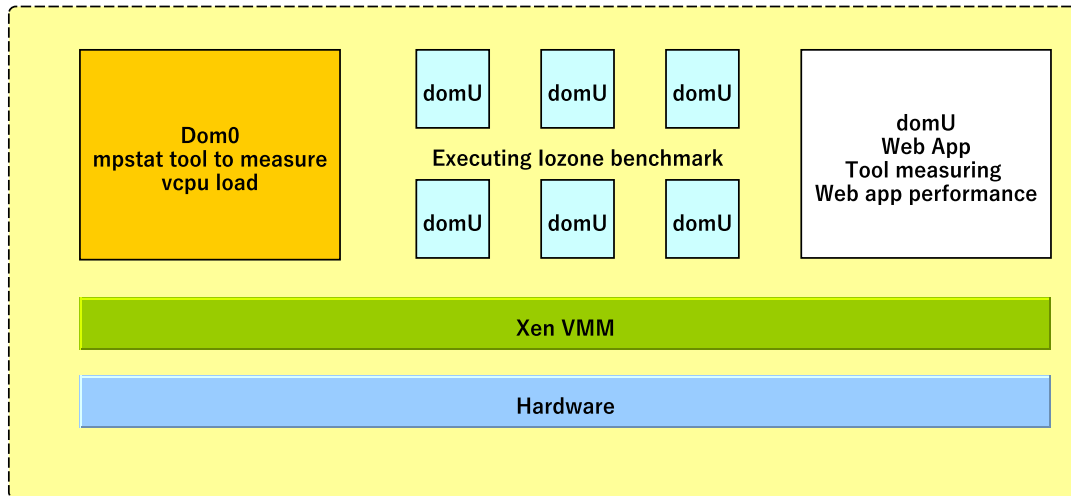


Figure 2.2: Experimental setup to show the impact of dom0's resources on user's application.

is responsible for accessing the hardware. The second observation is that the web application performance decreases when the dom0 lacks CPU resources since CPU resources are statically allocated to the dom0. Therefore, **performance predictability** is compromised, which is one of the main issues in cloud computing environments [? ? ? ? ?].

Impact on virtual machine management tasks. The dom0 hosts the execution of virtual machine administration commands, the most important commands being: **create**, **destruction**, **migration** of virtual machines and modification of resources allocated to the virtual machines. The limitation of dom0 allocated resources leads to a variation of the execution time of these commands since they require a significant amount of resources. For instance, virtual machine migration requires a lot of CPU to detect dirty pages from the virtual machine being migration [? ? ?]. We conducted an experiment (whose setup is shown on Figure 2.3) to show the impact of dom0's resources on these tasks. On Figure 2.8b and 2.7b, we observe that the dom0 load has a significant impact on these execution times, and may dramatically influence higher level services such as auto-scaling (which relies on virtual machine creations and destructions) or consolidation (which relies on virtual machine migrations).

Impact on data center management services. An overload of the dom0 (due to an overload of hosted virtual machines, which may be a consequence of a **denial service attack**) can significantly impact the management system of a data center. If we consider an **OpenStack** system, each server executes (in its dom0) a set of tasks which enable its management in the data center. If these tasks are no longer reachable from the central controller (Nova), the server is considered out of order, which may trigger a recovery procedure. We experimented with such a situation in a small OpenStack cluster composed of one compute node and one controller node (as shown



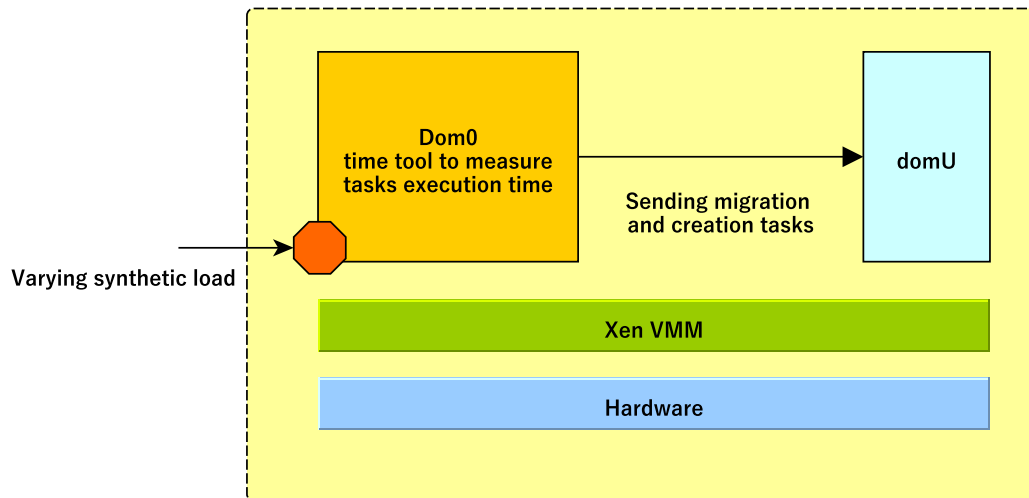


Figure 2.3: Experimental setup to show the impact of dom0's resources on management tasks.

on figure 2.4. OpenStack components were configured with default values (a timeout of 300s). We allocated to the dom0 of the compute node two vCPUs and created several domUs on this node. Then, we executed in these virtual machines the Iozone benchmark in order to overload the dom0. We observed that when reaching a given load (75%) in the dom0, the controller considered that the compute node was failing.

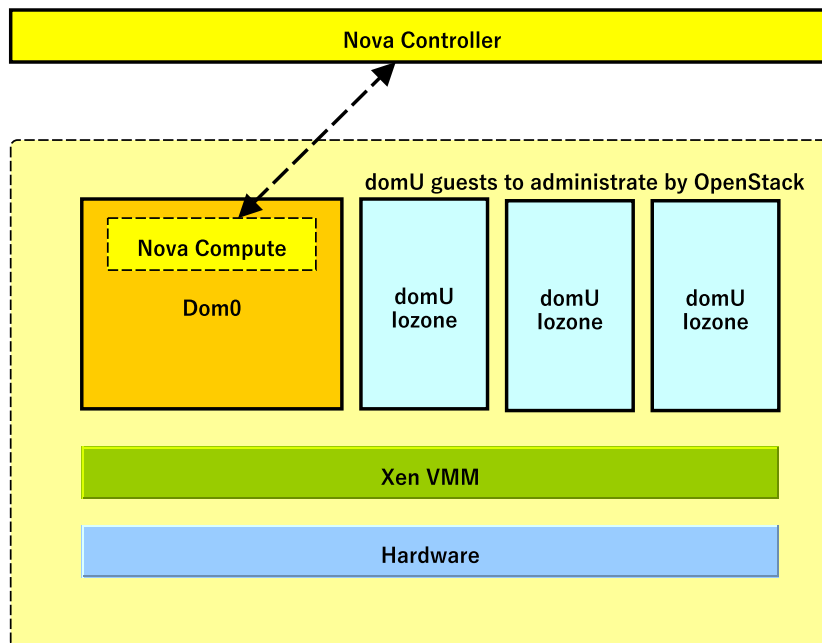


Figure 2.4: Experimental setup to show the impact of dom0's resources on the data center management services.



2.1.2 Consequence of an oversized dom0

Overprovisioning is a source of resource waste as shown by many works. Most previous works studied domU overprovisioning and its associated waste [? ? ?]. We are here interested in dom0 over-provisioning. In order to evaluate it, we relied on **Google traces**, considering that container workloads were executed in Domus. We statically allocated to the dom0 one socket from a machine. Assuming that domUs are executing **Hadoop** tasks, we experimentally determined the relationship between resources consumed by an Hadoop task and resources consumed by the dom0. Then, we evaluated on a period of traces (29 days) the amount of unused (wasted) resources in the dom0. These resources could have been used to host additional domUs or to increase the consolidation ratio. If we consider the smallest task in Google's data center, wasted resources could have satisfied 1440 tasks.

2.2 (Q_2) Dom0 location and Review

The location of resources allocated to the dom0 may significantly influence virtual machine's application performance.

Performance of I/O intensive applications. We performed an experiment (in a NUMA architecture as shown on Figure 2.5) to demonstrate the influence of dom0 resources location on the performance of I/O intensive applications. We executed a web application in a virtual machine whose CPU and memory resources are located on one socket. Then, we varied the location of the resources (CPU and memory) allocated to the dom0. Figure 2.9b presents the performance of the web application in different situations. We observe that the best performance is obtained when resources from the virtual machine and the dom0 are co-located on the same socket (socket 8). Indeed, co-location prevents **remote memory accesses** between sockets in the NUMA architecture.

Virtual machine migration. Figure 2.9a presents the virtual machine migration time according to the location of the dom0 VCPUs on the machine's sockets (as shown on figure 2.6. When the dom0 VCPUs are co-located with the virtual machine to migrate (on socket 8), migration is much faster since remote memory accesses between sockets are avoided. We observed the same behavior for the removal of a virtual machine (since the scrub operation is faster when memory is accessed locally), which has a strong impact on the autoscaling mechanism [?].



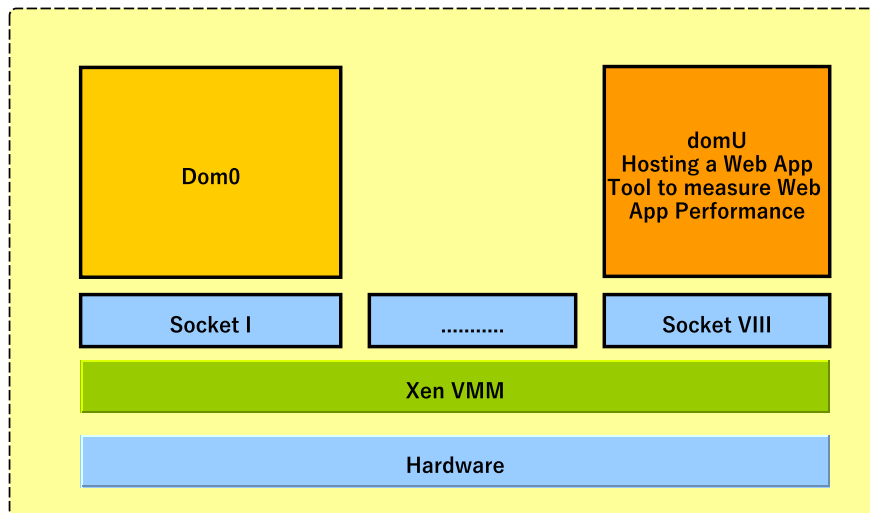


Figure 2.5: Experimental setup to show the impact of dom0's resources location on domU IO applications.

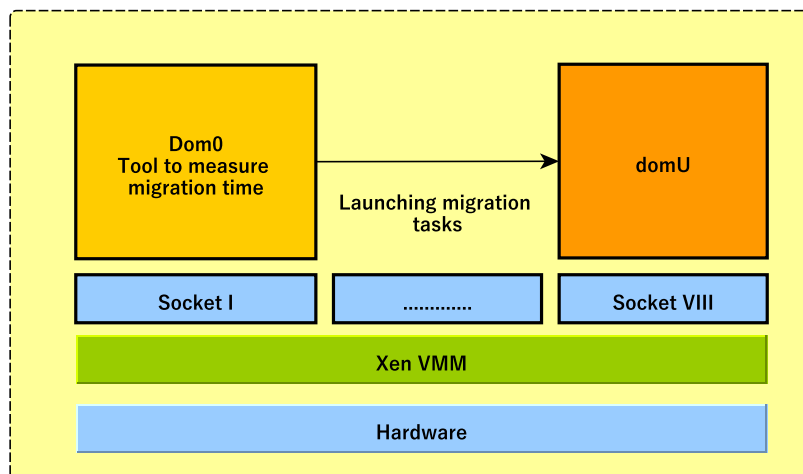


Figure 2.6: Experimental setup to show the impact of dom0's resources location on domU migration.

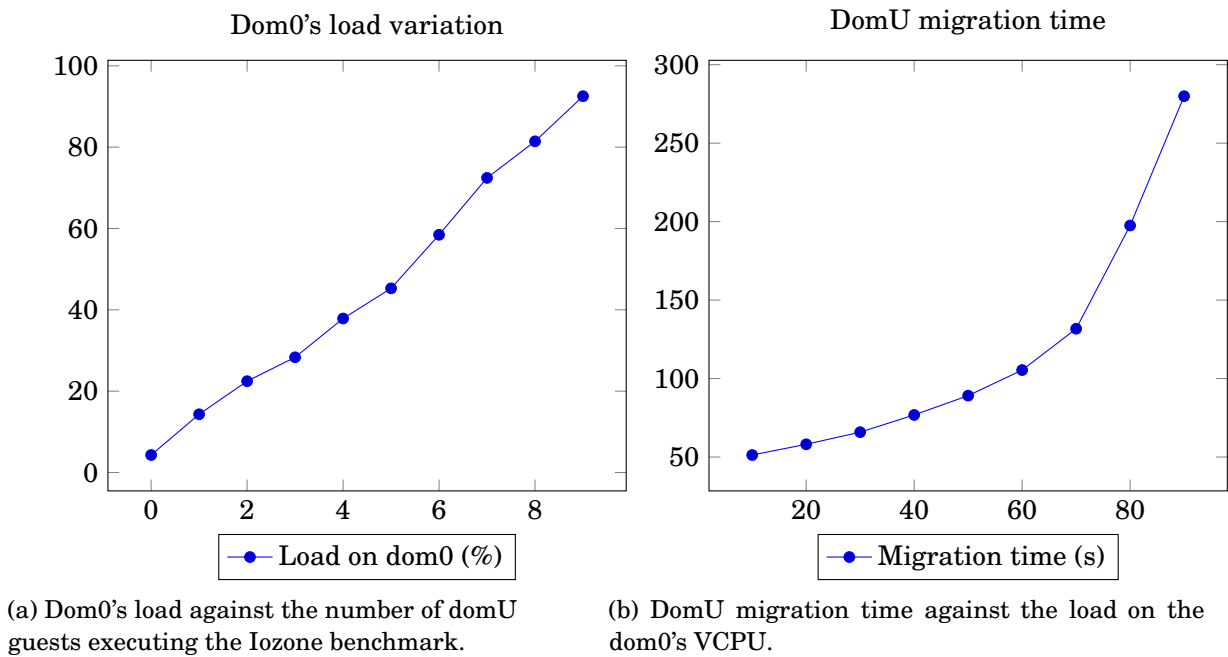


Figure 2.7: Experimental results, first group

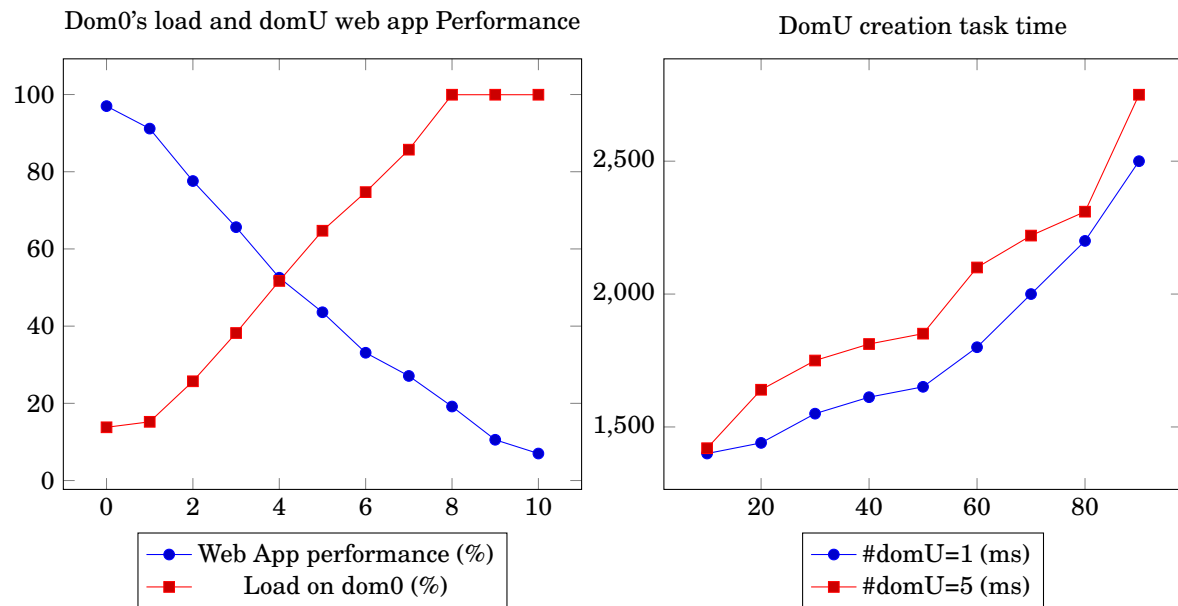


Figure 2.8: Experimental results, second group



2.3 Review

From all the experiments we conducted, it is clear that a non-intelligent allocation for the dom0 will lead to adverse consequences for the provider and its customers. We identified the main problem and in the next chapter, we present our approach to this problem.

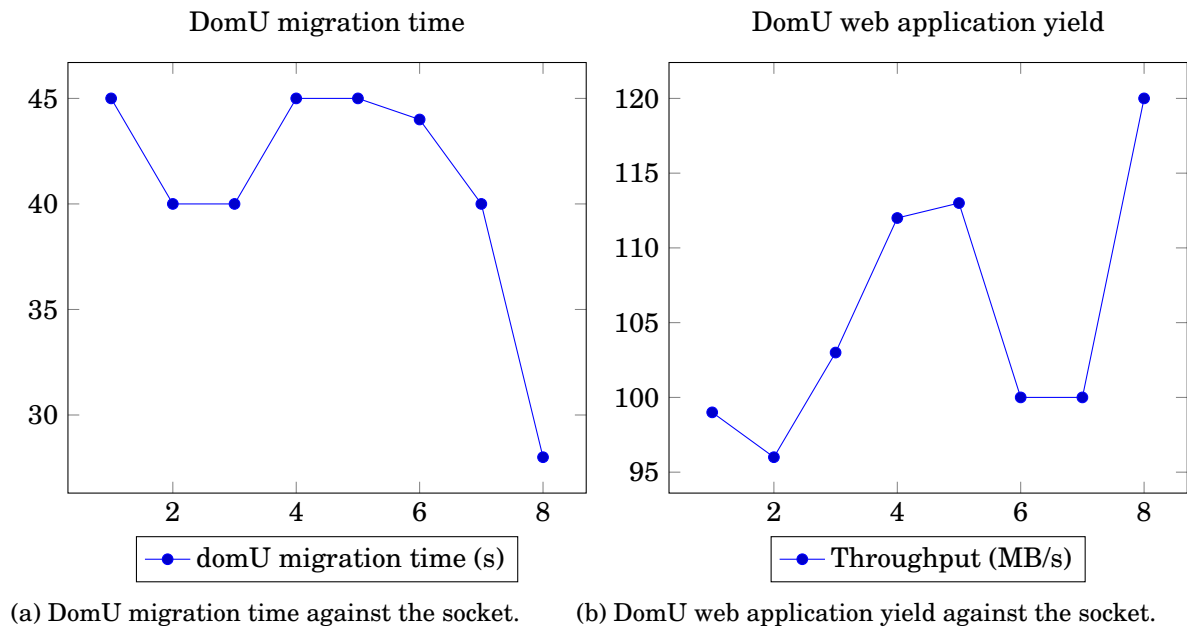


Figure 2.9: Experimental results, third group

CONTRIBUTION

From the experiments conducted in Chapter 2, we identified a set of problems and designed our solution based on the observations made. This chapter presents our architecture and describes the different modules of which it is constituted. The plan of this chapter is as follows:

Contents

3.1	Container composition	27
3.2	Resource allocation to the main container	27
3.3	Resource allocation to secondary containers	29
3.3.1	Memory allocation	29
3.3.2	Processor allocation	30
3.4	Our approach advantages	31



In the previous chapter, we demonstrate the consequences of a non-intelligent allocation of resources to the dom0. To solve this problem, we propose an architecture of dom0 which will respect the following three principles:

1. the dom0 must have exactly the amount of resources it needs at every moment,
2. the processes that work on behalf of virtual machines must use the resources of these virtual machines and
3. these processes must be executed as close as possible to the virtual machines for which they work.

We propose an architecture in which dom0 is a kind of multi-kernel [?], as shown in Figure 3.1. It is organized in containers of two types: a **main container** and **secondary containers**. The following sections describe the contents of each container.

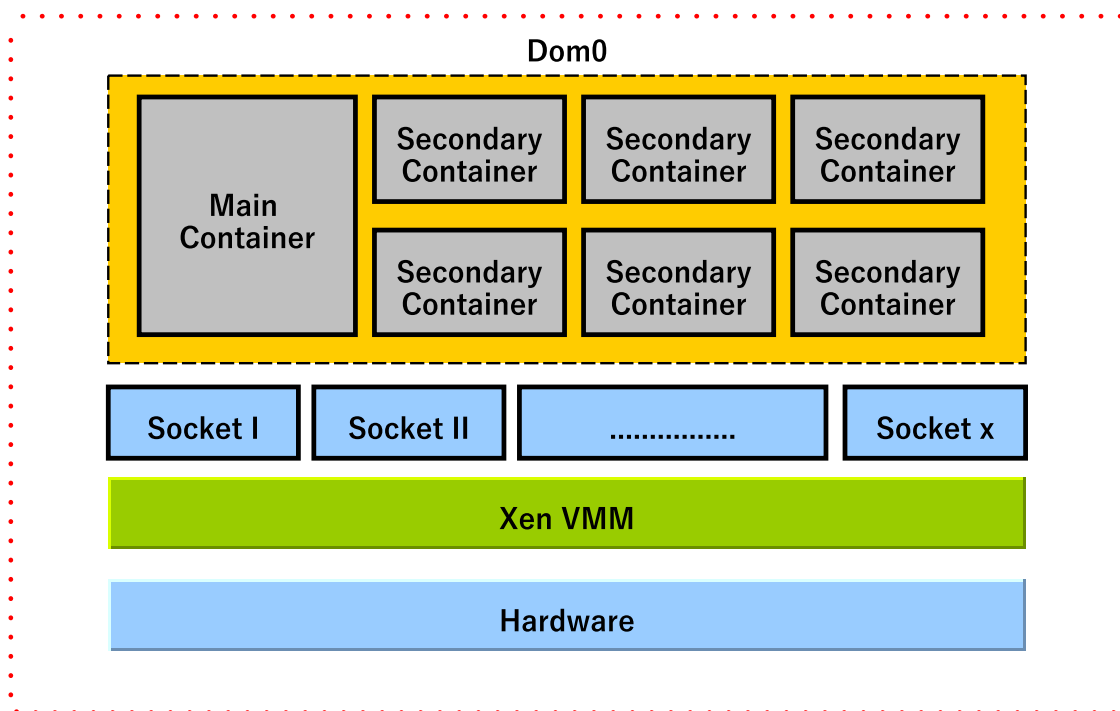


Figure 3.1: Our dom0 design





3.1 Container composition

We have organized the processes of dom0 into four groups, see table 3.1, (T_o) the basic processes of a Linux system, (T_1) management processes for the data center, (T_2) the virtual machine administration processes, and (T_3) the management processes of I/O. The tasks of T_o , T_2 and part of T_2 (excluding the tasks of shutdown and migration of virtual machines) constitute the main container. These are processes whose resource consumption is almost constant (section 3.2). As for the secondary container, it consists of virtual machine migration and shutdown processes and all processes of T_3 . The resource consumption of these processes depends on the target virtual machines. The number of secondary containers corresponds to the number of sockets dedicated to Domus (section 3.3).

The process creation task has been modified to take this classification into account. Each processor is identified by its category and assigned in the right container. The same is true for VCPUs containers. A VCPU is of either of type **main** or **secondary** depending on its container.

Table 3.1: Dom0 processes organization

Group Designation	Processes included
T_o	Basic processes of a Linux system
T_1	Data center management processes
T_2	VM administration processes
T_3	I/O management processes

VM: Virtual Machine.

3.2 Resource allocation to the main container

A number of resources consumed by the main container processes must be supported by the provider, unlike the secondary container processes. Moreover, these resources have no need to be close to the Domus. On the contrary, they would harm each other. The number of resources consumed by the main container is constant. Figure 3.2 shows the consumption of these processes as part of a data center handled by OpenStack (minimalist architecture as on Figure 3.3). We see that regardless of the number of virtual machines on the server, these processes (essentially the **nova-compute**) consume the same amount of memory and CPU. Note also that this consumption does not depend on the number of servers in the data center but on the complexity of the OpenStack architecture set up. In this OpenStack standard configuration, we estimated that the allocation of 2 processes and 2GB of memory to the main container is sufficient. Thus, for the allocation of the resources of the main container, we propose a static allocation estimated by the provider. Table 3.2 gives several configurations for the minimalistic architecture of OpenStack, Eucalyptus, and OpenNebula, three reputed cloud data center frameworks.



Table 3.2: Data center minimalistic architecture consumption

	Number of Virtual machines									
	10		20		30		40		50	
	VCPU	Mem	VCPU	Mem	VCPU	Mem	VCPU	Mem	VCPU	Mem
Euc	1	1.5	1	1.5	1	1.5	1	1.5	1	1.5
OpN	2	1.85	2	1.85	2	1.85	2	1.85	2	1.85
OpS	2	2	2	2	2	2	2	2	2	2

- Euc = Eucalyptus, OpN = OpenNebula, OpS = OpenStack, Mem = Memory.
- VCPU stands for the number of VCPU required.
- All the memory values are in GB.

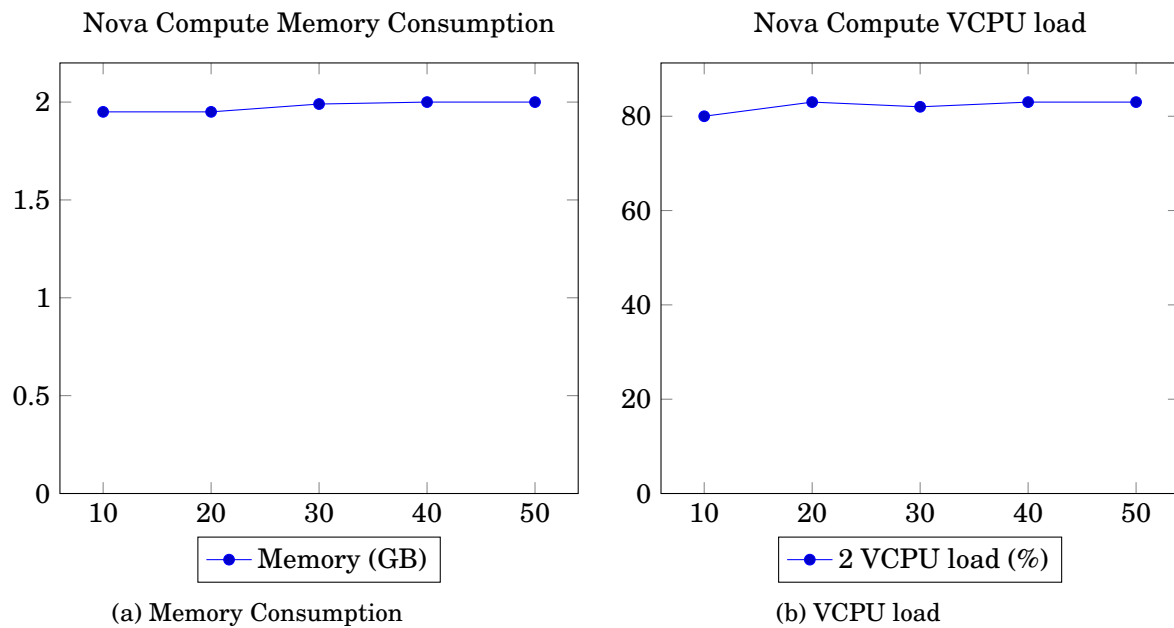


Figure 3.2: OpenStack Minimalistic set-up consumption

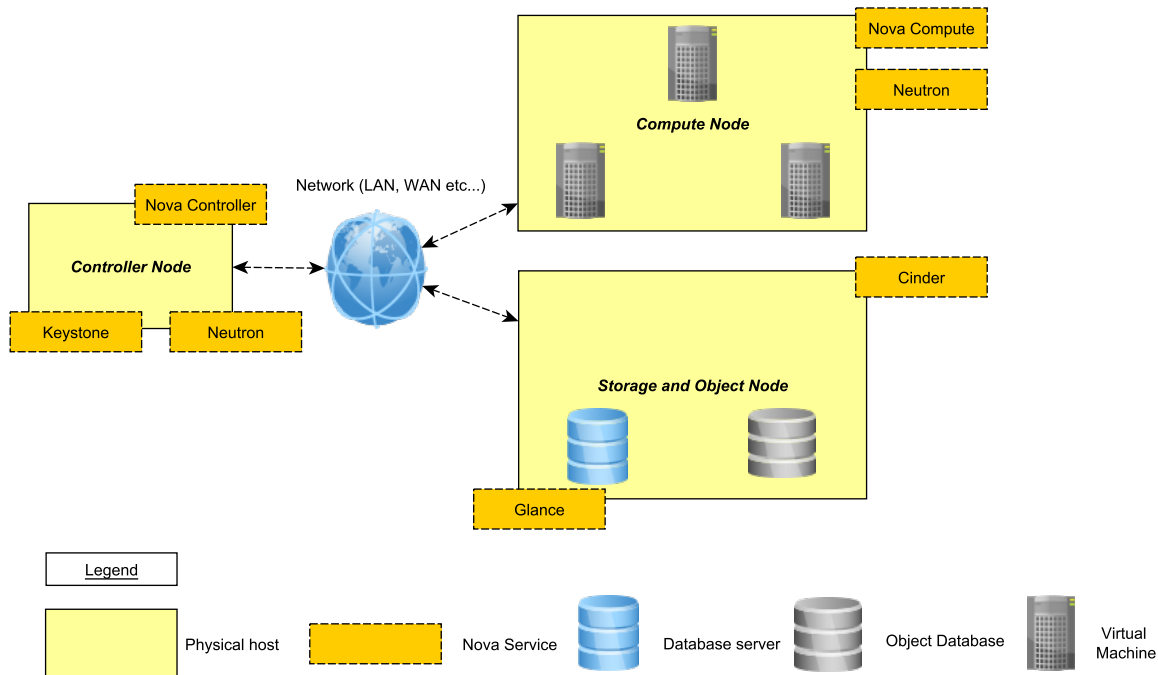


Figure 3.3: Minimal OpenStack architecture set-up

3.3 Resource allocation to secondary containers

Among the tasks that are in a secondary container, we have the tasks of virtual machine administrations and I/O management tasks. The first ones are created only when needed while the seconds are created when the server starts. The allocation of resources to these tasks must be gently done because their activities depend on the domains and their localities can affect the performance of these domains. To do this, we associate a secondary container per NUMA socket so that a virtual machine that runs on the socket x is managed by the secondary container of this socket. The resources of a secondary container are restricted to its socket. In addition, the resources consumed by the secondary containers are taken from the resource quota of the domU. The following sections introduce **our scheduling algorithms** for these tasks.

3.3.1 Memory allocation

After the server starts (no domU), each secondary container contains only the back-end management processes of I/O that do not run. The amount of memory used by this back-end is negligible (44KB in the case of Xen). This amount of memory is assumed by the provider, as well as the memory that will be used by the processes of migration and shutdown of virtual machines. Thus the initial size of a secondary container is 75MB, which is negligible. The size of the memory will be adapted according to the activity of the domU. Let M be the amount of memory requested by a domU that must be started. Let n be the number of sockets on which the domU will be



executed. Thus the amount of memory that will actually be given to the VM at startup will be $M - n \times \epsilon$, ϵ will be the initial contribution of the domU in each secondary container. ϵ corresponds the amount of memory used by dom0. The possible values of ϵ can be calculated beforehand by studying all the types of file system executed by the cloud (on Amazon, for example, we know all types of instances in advance). Once the virtual machine is started, it can be called up (ballooning) according to the intensity of its activities I/O. The claim policy is made in increments of intensity and the memory is always requested in advance for each level passed. Note that when the intensity decreases, the virtual machine will be given back the memory.

Algorithm 1 Memory Allocation for the secondary container in charge of the virtual machine VM using the defined parameter ϵ

```

1: procedure SECMEMALLOC(VM,  $\epsilon$ )
2:    $M \leftarrow$  MemoryRequested (VM)                                ▷ Get the VM memory size at creation
3:    $n \leftarrow$  SocketNumber (VM)
4: initialization:
5:    $secMem = M - (n \times \epsilon)$ 
6: loop:
7:   if VMIOIncreasing(VM) then                                ▷ If the I/O intensity of VM is increasing
8:      $reserve \leftarrow$  BalloonReserve (VM)                        ▷ Increase secondary container's memory
9:      $secMem \leftarrow secMem + reserve$  .                          ▷ by removing from VM's memory
10:    goto loop.
11:  else
12:    if VMIODecreasing (VM) then                                ▷ If the I/O intensity of VM is decreasing
13:      GiveBackMem (VM,  $secMem$ )
14:    goto loop.

```

3.3.2 Processor allocation

At server startup, dom0 is created with as many VCPUs as there are processors on the server. VCPUs allocated to the main container (tagged main VCPU) are pinned to their processors and are never rescheduled. Here we are interested in other VCPU (tagged secondary VCPU), used by secondary containers. At server startup, each of these VCPUs are pinned to a processor (except those used by the VCPU tagged main) but is not scheduled. They will be used whenever necessary, according to the algorithm Figure 3.4.

domU shutdown and migration operations. When one of these operations is requested, the associated process is created and assigned to the secondary container of the socket on which the greatest amount of memory of the target virtual machine is located. Then, the VCPU that will execute the operation is the one that has been pinned to the processor currently used by the target VCPU. In this way, the processes are executed next to the memory of the target virtual machines, which accelerates their execution (these processes work mainly on the memory). Using





one of the processors of the virtual machine can be seen as a problem in the migration, slowing down the execution of the virtual machine. This slowdown accelerates the migration operation because it reduces the frequency of modification of the memory pages by the virtual machine. This allows you to quickly migrate the VM [? ?]. The evaluations confirm this.

Algorithm 2 Task Scheduling for a secondary container in charge of virtual machine VM

```

1: procedure SEC MEM ALLOC(VM)
2:   secCon ← GreatestMemSocket (VM)
3:   vcpu ← pinnedVCPU (VM,secCon)
4:   Schedule(secCon,vcpu)

```

I/O management operations. These operations are of two types: the operations initiated by the domU and the operations not initiated by the domU. In the first type, we have read and write disk operations, and packet sending operations on the network. In the second type, we have the reception of packets. Our policy of allocating processors to these processes in the case of operations initiated by the VM consists in scheduling them on one of the VCPUs of the secondary container and this VCPU will in turn scheduler on the processor which was used by the VCPU of the virtual machine having initiated the operation (in the case of the tasks initiated by the virtual machine). For external operations, the processes of the secondary container responsible for processing the operation are scheduled on one of the processes used by the target VM. VCPUs will be chosen fairly. The implementation of this policy in Xen is quite fast. For operations initiated by the VM, it is easy to know the VCPU that initiated the operation. This is the VCPU that generates the virtual interrupt to signal to the back-end the presence of an operation in the ring buffer (in chapter 1). For external operations, it is sufficient to count the participation of each VCPU of the secondary container and to choose the one that contributed the least. This is the processor, which will be responsible for processing the operation in question. It is another VCPU of the virtual machine that will receive the virtual interrupt saying that a packet is present.

3.4 Our approach advantages

Th new dom0 design we came up with, present a number of advantages which are :

- **Predictability:** The performance of virtual machines will not be disturbed by the activities of neighboring guests, so the yield should be what is expected.
- **Locality:** Each time the dom0 will have to work on behalf of a virtual machine, its resources will be co-located with those of the virtual machine target, thus preventing remote accesses.
- **Accounting/Billing:** Each virtual machine will consume a number of resources requests at startup since the dom0 uses the virtual machine resources to realize their work.



- **Scalability:** This approach allows the dom0 to be scalable by distributing its resources over the entire hardware architecture.

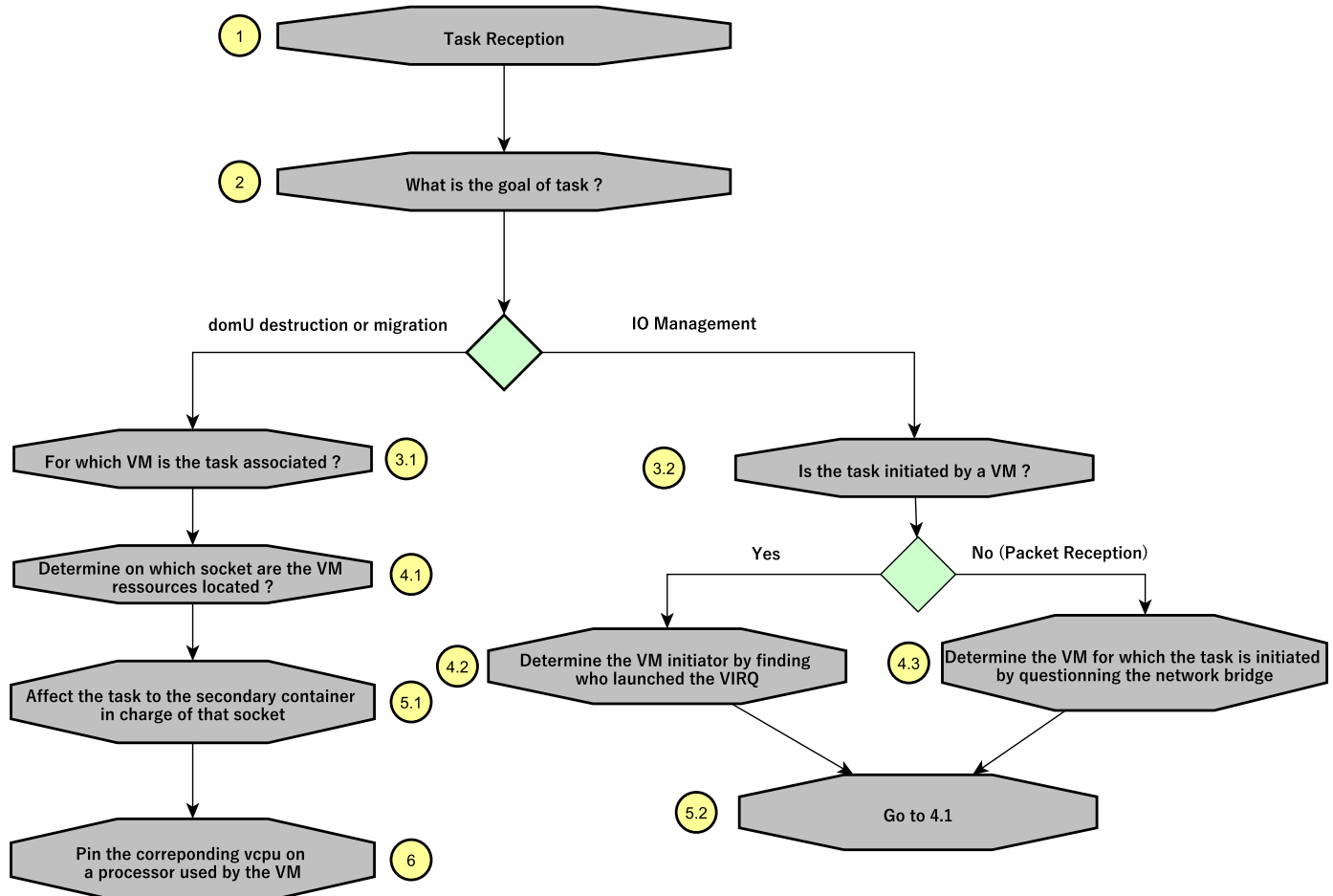


Figure 3.4: Scheduling Algorithm for secondary dom0 tasks

IMPLEMENTATION AND EVALUATION

In this chapter, we will give some details of implementation, the operating systems we used and the development tools we used, then we will present the evaluation of our solution and comparing our results with the actual solutions presented in this document. The plan of this chapter is as follows :

Contents

4.1	Implementation	34
4.1.1	The hypervisor	34
4.1.2	The guest OS	34
4.1.3	Development tool	34
4.2	Evaluation	35
4.2.1	Benchmark 1: Load generated on the dom0 due to domUs activities .	35
4.2.2	Benchmark 2: Migration time under high load conditions	36
4.2.3	Benchmark 3: I/O performance as a result of locality	36
4.2.4	Benchmark 4: Migration time as a result to locality	37
4.3	Review	37



4.1 Implementation

4.1.1 The hypervisor

In order to implement our solution, we had to modify a hypervisor in order to take into account the new design and scheduling algorithms. Since our work is based on the Xen architecture with its privileged domain, we used this latter to implement our solution. Let's note that, our solution is not only specific to Xen but two all hypervisors whose architecture is based on a privileged guest whose role is to execute a number of tasks on behalf of some other guests.

Xen is a hypervisor written in **C**, **Assembly language** and **Python**. Its source code is composed of about 5000 files containing more than a million lines of code. To date, Xen is now at version **4.9** but for stability issues, we worked on its **4.8** version.

4.1.2 The guest OS

It is important to choose an OS to work on, since the device driver code and the interrupt handlers depend on the OS. In this work, we chose **Linux** for:

- its open source nature which permits us to modify the source code,
- its large community of developers, which is active both in native or virtualized domain,
- its ability to integrate paravirtualization offered by **Xen**.

The Linux kernel is written in **C** and **Assembler**. The source code of Linux is composed of about 14000 files, distributed in about 1000 folders containing about 6 million lines of code. To date, Linux is now at version **4.12.9** but for compatibility issues with our Xen version, we worked with version **4.9**.

4.1.3 Development tool

4.1.3.1 The compiler

To compile the Linux kernel, the compiler **GNU Compiler Collection (GCC)** must be used. GCC is a set of compilers created by the project GNU. GCC is a free software enabling compilation of many languages such as C, C++, Objective-C, Java, Ada, and Fortran. In order to use it, we need the tool **make** to call the GCC compiler. We used the most recent version to date, **4.2**.

4.1.3.2 The editor

When working on a project containing millions of lines of code such as Linux or Xen, it is important to use an integrated development environment (IDE). However, the disadvantage of IDEs is that they must use their compiler, which is not obvious when programming in the kernel. We





opted for the **Vim editor** and the **Cscope navigation tool**. Cscope is a Linux tool for navigation and exploration of symbols, definitions of symbols, directory structures, and others. It is an integrated tool in Vim. Initially built to work with C code, it currently supports C ++ and Java. During this work, we used the most recent version to date, **8.0**.

4.2 Evaluation

In order to evaluate our solution, we established a set of synthetic benchmarks to verify that our solution met the objectives for which it was designed and to verify if it performed better than the native implementation. In the section that follows, we will present the different benchmarks we used, their evaluation metrics, the experimental setup and the results obtained comparing it with native solutions.

4.2.1 Benchmark 1: Load generated on the dom0 due to domUs activities

This synthetic benchmark will permit us to evaluate if our solution, isolates completely a virtual machine preventing the latter to disturb other activities (other domU applications and dom0 administrative tasks) due to the charge generated on the dom0. Here the main metric is the **load generated** on the dom0 VCPU.

4.2.1.1 Experimental Setup

The experiment was conducted on a physical computer with the following characteristics :

- 56 processors, 2,5 GHz Intel Xeon,
- 40 GB memory,
- Xen 4.8 on Ubuntu 12.04 Linux Kernel 4.9.

Here we configured the dom0 to have 2 VCPU on the main container and 2 GB main memory. We then increased the number of Domus executing an I/O intensive benchmark Iozone and we measured the load generated on the 2 VCPU. We will then compare with the native solution if the dom0 was configured with 2 VCPU and 2 GB main memory.

4.2.1.2 Experimental Results

We can observe on Figure 4.1a that, in the native implementation, the load on the VCPU increases as the number of Domus increases which can be a cause of performance unpredictability. On the other hand, with our design, the load generated on the 2 VCPU is nearly constant and does not come from the Domus but from the tasks from group T_o, T_1, T_2 .





4.2.2 Benchmark 2: Migration time under high load conditions

This synthetic benchmark will permit us to evaluate if our solution, carries out domU administrative tasks independently on the load on its main container resources. Here the main metric is **migration time**.

4.2.2.1 Experimental Setup

The experiment was conducted between 2 physical computers with the following characteristics :

- 8 processors and 4 processors,
- 16 GB memory and 8 GB memory,
- Xen 4.8 on Ubuntu 12.04 Linux Kernel 4.9 on each ,
- and linked over a Cisco Catalyst 2960G-24TC-L-Switch.

Here the dom0 was configured as in benchmark 1, and the domU to migrate was executing a script whose role was to make random memory accesses in order to simulate memory intensive activities. The migration time was taken and the load on the dom0 main container VCPU increased.

4.2.2.2 Experimental Results

As shown on Figure 4.1b, with the native implementation, the migration time increases as the load increases, compared to our solution where the migration is nearly constant, due to the fact that the migration task is taken in charge by the secondary containers which are scheduled on the virtual machine resources.

4.2.3 Benchmark 3: I/O performance as a result of locality

This synthetic benchmark allows us to evaluate our solution, from a local point of view. Here we compare the performance obtained when a domU guest executing Iozone benchmark and its resources are distant to those of the dom0. We also measure **memory bandwidth** for the two approaches, the native implementation, and our design.

4.2.3.1 Experimental Setup

The experiment was conducted on a physical machine with the following characteristics :

- 8 processors 1,70 GHz Intel Xeon,
- 16 GB memory,





- Xen 4.8 on Ubuntu 12.04 Linux Kernel 4.9 on each.

The dom0 was configured as in benchmark 1, and the domU hosts an Iozone benchmark and the metrics were plotted.

4.2.3.2 Experimental Results

As shown on Figure 4.2a, the native implementation bandwidth is lesser than our design bandwidth, due to the fact that, the processes of the dom0 in charge of the domU I/O requests are co-located with the domU resources, hence avoiding remote accesses.

4.2.4 Benchmark 4: Migration time as a result to locality

The same principle as for benchmark 3, the only difference is that the metric here is the domU migration time.

4.2.4.1 Experimental Setup

The experiment was conducted on the same conditions as Benchmark 2. Here the domU resources located is placed so that the dom0 resources should be distant to this domain and we launch the domU migration.

4.2.4.2 Experiment Results

As shown on figure 4.2b, the native implementation migration time is greater than our solution, due to the fact that the process in charge of the migration was co-located with the domU to migrate, hence avoiding remote accesses.

4.3 Review

These series of benchmarks were realized in order to compare our solution to the native solution over the objectives defined in the introduction. The results obtained in those benchmarks enabled us to conclude that our solution gives a better performance than the native solutions and it meets the objectives for which it was designed. Our solution gives better execution times than native implementations and the execution time is independent of the domUs activities intensity which enables a better billing of the domUs activities.



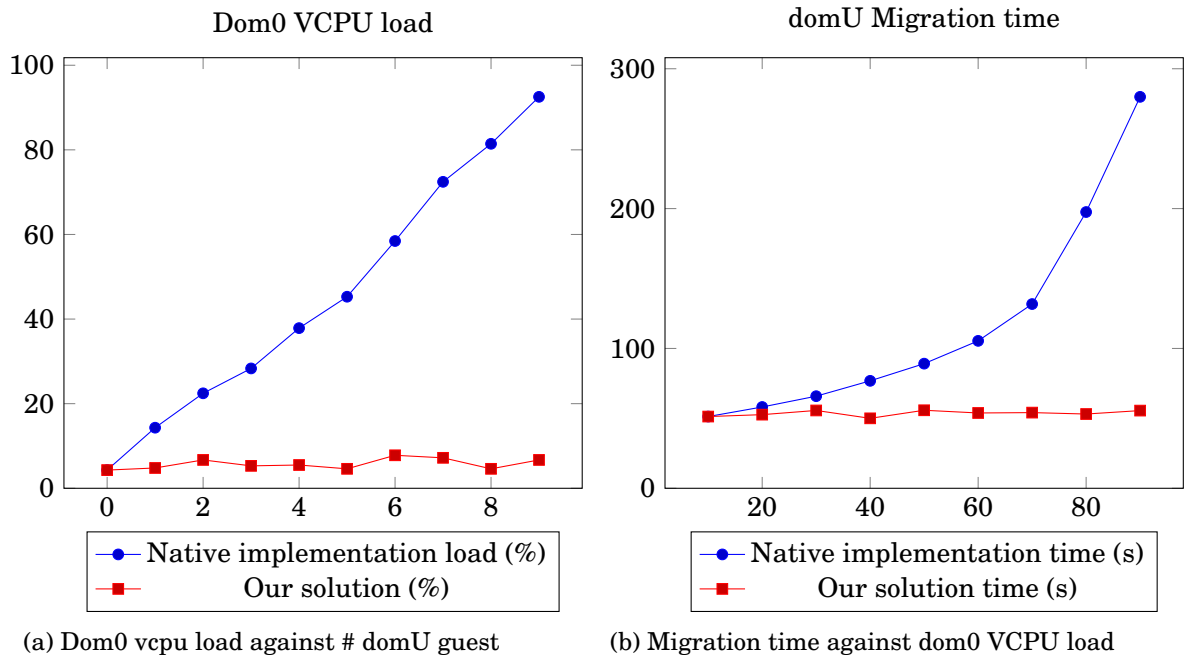


Figure 4.1: First group results

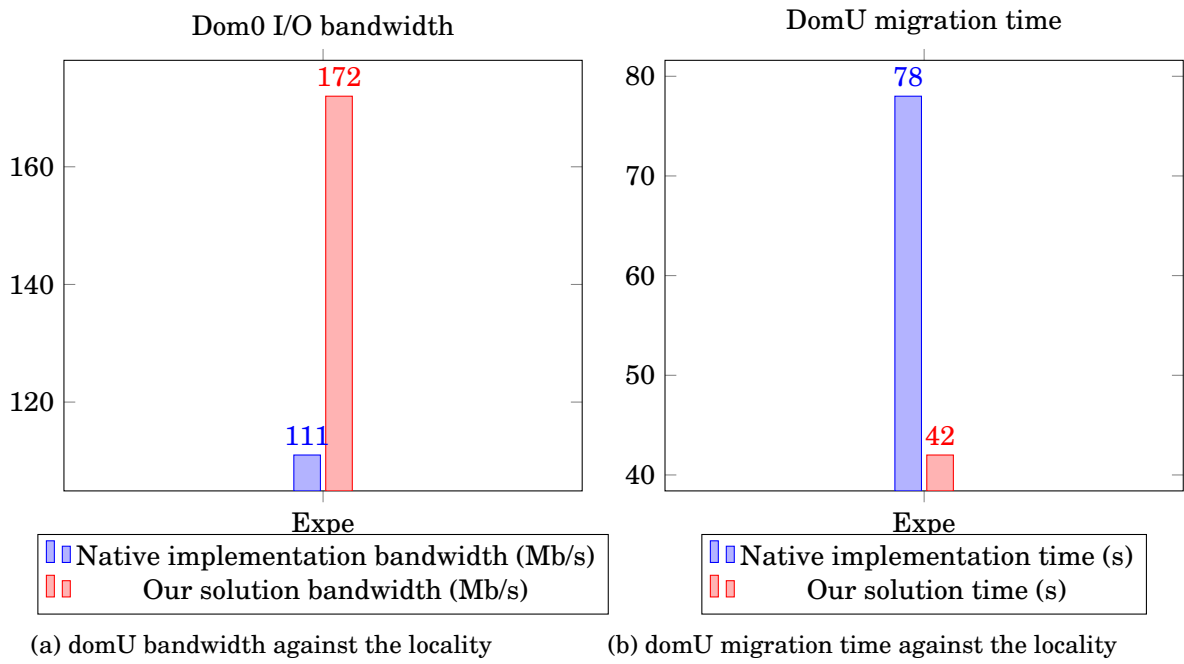


Figure 4.2: Second group results

CONCLUSION

Review

This work reveals that resource allocation to the privileged domain is a serious question due to the consequences that can occur if this is not carefully managed. We explored the server virtualization system Xen to understand its architecture, and how it is implemented. We also learned about the architecture of the Linux kernel and the I/O handlers. This enabled us to design and implement our solution taking into account the limitations of current solutions. Our design in addition to the numerous advantages of its architecture, performs better than the native implementation, on average, 12% gain in performance for a given set of tasks. This work is currently the core of an article named "*Resource management in virtualized systems: the privileged domain should not be overlooked*" and will be submitted to the "*European Conference on Computer Systems 2018*", which is a major conference in the field of virtualization.

Prospects

In order to improve on this work, the evaluations should be increased in order to evaluate every module developed. Furthermore, we can take a look at the network reception mechanism, since this aspect was not taken into account in our design. During the network packet reception, we can implement a hash function which should be able to identify the destined virtual machine and schedule the handlers to take the packet in a secondary container in charge of the virtual machine. Furthermore, we can go beyond and realize a weighting system where a module monitors the network activity of the virtual machines and increases the weight of each virtual machine so as when a packet comes from the exterior, the resources needed to handle this packet should be taken from the virtual machine with the greatest weight. This will permit each virtual machine to contribute to the network reception mechanism and the lesser the network activity of a virtual machine, the lower the probability of its resource being used to handle incoming packets but the higher the network activity of a virtual machine, the greater the probability of its resource being used to handle incoming packets.

Internship host structure description

IRIT, the Toulouse Research Institute of Computer Science, one of the largest joint research units at the national level, is one of the pillars of research in the **Midi-Pyrénées** with its 700 permanent and non-permanent members. Due to its scientific impact and its interactions with other fields, the laboratory is one of the structuring forces of the IT landscape and its applications in the digital world, so regional and national level. The unit is structured into 7 research themes grouping 21 laboratory teams. Its main research axes are:

1. Analysis and synthesis of information.
2. Indexing and retrieving information.
3. Interaction, Cooperation, Adaptation through Experimentation.
4. Reasoning and decision.
5. Modeling, algorithms and high-performance computing.
6. Architecture, systems, and networks.
7. Software development security.

Beyond that, IRIT's influence is reflected in various actions at european and international level, for example, the european laboratory IREP, the french spanish Laboratory for advanced studies in information and perennial cooperation with various countries including the Maghreb, Japan, Armenia, the Unites States of America, etc.

IRIT present in all Toulouse universities as well as the Institute of Technology of Tarbes Castres provides both coverage of the local territory and all the themes of computer science and its interactions, ranging from infrastructure enabling it to contribute to the structuring of regional research. The laboratory works to establish a continuum from research to valorization, despite the difficulties and limitations inherent in existing devices (industrial property, patent), by imagining and developing innovative forms of collaboration around the concepts of a joint laboratory or a consortium.